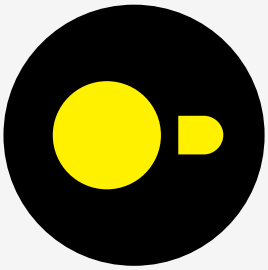
DuckDB Blog

Generated on 2026-10-26 at 12:22 UTC



Contents

Contents i

Summary 1

DuckDB Blog 3

Testing Out DuckDB's Full Text Search Extension 4

Efficient SQL on Pandas with DuckDB 7

Querying Parquet with Precision Using DuckDB 16

Fastest Table Sort in the West – Redesigning DuckDB's Sort 24

Windowing in DuckDB 39

DuckDB-Wasm: Efficient Analytical SQL in the Browser 47

Fast Moving Holistic Aggregates 54

DuckDB – Lord of the Enums: The Fellowship of the Categorical and Factors 59

DuckDB Quacks Arrow: A Zero-Copy Data Integration between Apache Arrow and DuckDB 65

DuckDB Time Zones: Supporting Calendar Extensions 72

Parallel Grouped Aggregation in DuckDB 78

Friendlier SQL with DuckDB 88

Range Joins in DuckDB 95

Persistent Storage of Adaptive Radix Trees (ART) in DuckDB 104

Querying Postgres Tables Directly from DuckDB 120

Modern Data Stack in a Box with DuckDB 126

Lightweight Compression in DuckDB 132

Announcing DuckDB 0.6.0 143

Announcing DuckDB 0.7.0 149

JupySQL Plotting with DuckDB 154

Shredding Deeply Nested JSON, One Vector at a Time 162

The Return of the H2O.ai Database-like Ops Benchmark 174

Introducing DuckDB for Swift 177

PostGEESE? Introducing The DuckDB Spatial Extension **183**

10 000 Stars on GitHub **187**

Announcing DuckDB 0.8.0 **188**

Correlated Subqueries in SQL **192**

From Waddle to Flying: Quickly Expanding DuckDB's Functionality with Scalar Python UDFs **198**

DuckDB ADBC – Zero-Copy Data Transfer via Arrow Database Connectivity **205**

Even Friendlier SQL with DuckDB **209**

DuckDB's AsOf Joins: Fuzzy Temporal Lookups **222**

Announcing DuckDB 0.9.0 **234**

DuckDB's CSV Sniffer: Automatic Detection of Types and Dialects **240**

Updates to the H2O.ai db-benchmark! **249**

Extensions for DuckDB-Wasm **251**

Multi-Database Support in DuckDB **259**

Announcing DuckDB 0.10.0 **264**

SQL Gymnastics: Bending SQL into Flexible New Shapes **272**

Dependency Management in DuckDB Extensions **281**

42.parquet – A Zip Bomb for the Big Data Age **286**

No Memory? No Problem. External Aggregation in DuckDB **288**

duckplyr: dplyr Powered by DuckDB **294**

Vector Similarity Search in DuckDB **302**

Access 150k+ Datasets from Hugging Face with DuckDB **305**

Analyzing Railway Traffic in the Netherlands **310**

Announcing DuckDB 1.0.0 **317**

Native Delta Lake Support in DuckDB **320**

Command Line Data Processing: Using DuckDB as a Unix Tool **328**

20 000 Stars on GitHub **336**

Benchmarking Ourselves over Time at DuckDB **340**

DuckDB Community Extensions **349**

Memory Management in DuckDB **352**

Friendly Lists and Their Buddies, the Lambdas **356**

DuckDB Tricks – Part 1 **363**

Announcing DuckDB 1.1.0 **367**

Changing Data with Confidence and ACID **375**

Creating a SQL-Only Extension for Excel-Style Pivoting in DuckDB **381**

DuckDB in Python in the Browser with Pyodide, PyScript, and JupyterLite **391**

DuckDB User Survey Analysis **395**

Analyzing Open Government Data with duckplyr **400**

DuckDB Tricks – Part 2 **405**

Driving CSV Performance: Benchmarking DuckDB with the NYC Taxi Dataset **410**

What's New in the Vector Similarity Search Extension? **415**

Fast Top N Aggregation and Filtering with DuckDB **418**

Analytics-Optimized Concurrent Transactions **424**

Optimizers: The Low-Key MVP **430**

Runtime-Extensible SQL Parsers Using PEG **438**

DuckDB Tricks – Part 3 **444**

CSV Files: Dethroning Parquet as the Ultimate Storage File Format — or Not? **450**

DuckDB: Running TPC-H SF100 on Mobile Phones **456**

The DuckDB Avro Extension **465**

25 000 Stars on GitHub **468**

DuckDB Node Neo Client **470**

Vertical Stacking as the Relational Model Intended: UNION ALL BY NAME **474**

TPC-H on a Raspberry Pi **480**

Query Engines: Gatekeepers of the Parquet File Format **483**

Announcing DuckDB 1.2.0 **486**

Catching up with Windowing **491**

Flying Through Windows **495**

Planning AsOf Joins **504**

Prefix Aliases in SQL **510**

Reading and Writing Google Sheets in DuckDB **512**

Gems of DuckDB 1.2 **517**

Parquet Bloom Filters in DuckDB **520**

The DuckDB Local UI **524**

Preview: Amazon S3 Tables in DuckDB **527**

Maximizing Your Delta Scan Performance in DuckDB **531**

Using DuckDB in Streamlit **539**

Fully Local Data Transformation with dbt and DuckDB **545**

DuckDB's CSV Reader and the Pollock Robustness Benchmark: Into the CSV Abyss **551**

Temporal Analysis with Stream Windowing Functions in DuckDB **559**

Sorting on Insert for Fast Selective Queries **565**

Machine Learning Prototyping with DuckDB and scikit-learn **570**

The Lost Decade of Small Data? **576**

Announcing DuckDB 1.3.0 **581**

Arrow IPC Support in DuckDB **588**

USING KEY in Recursive CTEs **593**

DuckLake: SQL as a Lakehouse Format **600**

Faster Dashboards with Multi-Column Approximate Sorting **601**

30 000 Stars on GitHub **606**

Lightweight Text Analytics Workflows with DuckDB **608**

Discovering DuckDB Use Cases via GitHub **615**

DuckLake 0.2 **623**

Spatial Joins in DuckDB **624**

Basic Feature Engineering with DuckDB **632**

Big Data on the Move: DuckDB on the Framework Laptop 13 **641**

Solving Letter Scramble Puzzles with DuckDB **645**

Announcing DuckDB 1.4.0 LTS **648**

DuckLake 0.3 with Iceberg Interoperability and Geometry Support **652**

Redesigning DuckDB's Sort, Again **653**

Announcing DuckDB 1.4.1 LTS **659**

Adoption Metrics and Benchmark Results for DuckDB v1.4 LTS **661**

Streaming Patterns with DuckDB **663**

Uncovering Financial Crime with DuckDB and Graph Queries **669**

Frozen DuckLakes for Multi-User, Serverless Data Access **676**

Relational Charades: Turning Movies into Tables **677**

Announcing DuckDB 1.4.2 LTS **685**

Data-at-Rest Encryption in DuckDB **689**

Writes in DuckDB-Iceberg **696**

Announcing DuckDB 1.4.3 LTS **702**

Iceberg in the Browser **705**

DuckDB on LoongArch **709**

Announcing Vortex Support in DuckDB **713**

Announcing DuckDB 1.4.4 LTS **717**

Announcing DuckDB 1.5.0 **719**

Big Data on the Cheapest MacBook **728**

DuckDB.ExtensionKit: Building DuckDB Extensions in C **732**

Announcing DuckDB 1.5.1 **736**

DuckDB Now Speaks Dutch! **738**

Data Inlining in DuckLake: Unlocking Streaming for Data Lakes **740**

Announcing DuckDB 1.5.2 **741**

DuckLake v1.0: The Lakehouse Format Built on SQL Reaches Production-Readiness **743**

The DuckLake Spec Is so Simple, Even a Clanker Can Build One for Dataframes **744**

Delta Grows Up: Writes, Unity Catalog and Time Travel **745**

Announcing the Program of DuckCon #7 Amsterdam **751**

Quack: The DuckDB Client-Server Protocol **752**

DuckDB 1.5.3: Not an Ordinary Patch Release **758**

Test-Driving the Lance Lakehouse Format in DuckDB **760**

New DuckDB-Iceberg Features in v1.5.3 **764**

Announcing DuckDB 1.4.5 LTS (Andium) **769**

Announcing DuckDB 1.5.4 (Variegata) **771**

Announcing DuckDB 1.5.5 **773**

Asynchronous I/O in DuckDB: Work, Thread, Work **775**

Thank You for 40 000 Stars on GitHub **783**

A Preview of DuckDB v2.0 **785**

Reconciling JSON in DuckDB, One Patch at a Time **792**

DuckDB v2.0: Your Database Deserves a Better Parser **798**

Chunked Query Results in the DuckDB Java Driver **806**

How DuckDB Runs Recursive CTEs Faster **809**

DuckDB Table Functions in Java **817**

DuckLabs to Join AWS, Projects to Remain Open Source **822**

Try DuckDB v2.0-dev **823**

DuckDB Skills for Claude Code **825**

Persistent Databases in the Browser with DuckDB-Wasm and OPFS **828**

DuckDB Now Ships inside dbt v2 **832**

DuckDB and Hugging Face: Querying Datasets Directly **836**

Announcing DuckDB 1.5.6 **841**

Jev and DuckDB: Plain-English Conditions in SQL **844**

Faster String Aggregations with Dimension Tables **848**

Importing Data using Java Table Functions **855**

A DuckDB Database with No Data in It **860**

Agent Mode in the DuckDB CLI **863**

What Is Inside an Iceberg Table? **868**

From JSON Strings to Typed Columns **876**

Benchmarking DuckDB v2.0-dev on NVIDIA Vera **880**

Announcing DuckDB 2.0.0 LTS “Cinnamon Teal” **882**

Acknowledgments **887**

Summary

This document contains DuckDB's blog posts in a single-file easy-to-search form. If you find any issues, please report them as a GitHub issue. Contributions are very welcome in the form of pull requests.

Code repositories:

- DuckDB source code: github.com/duckdb/duckdb
- DuckDB blog source code: github.com/duckdb/duckdb-web

DuckDB Blog

Testing Out DuckDB's Full Text Search Extension

Publication date: 2021-01-25

Author: Laurens Kuiper

TL;DR: DuckDB now has full-text search functionality, similar to the FTS5 extension in SQLite. The main difference is that our FTS extension is fully formulated in SQL. We tested it out on TREC disks 4 and 5.

Searching through textual data stored in a database can be cumbersome, as SQL does not provide a good way of formulating questions such as "Give me all the documents about **Mallard Ducks**": string patterns with `LIKE` will only get you so far. Despite SQL's shortcomings here, storing textual data in a database is commonplace. Consider the table `products (id INTEGER, name VARCHAR, description VARCHAR)` – it would be useful to search through the `name` and `description` columns for a website that sells these products.

We expect a search engine to return us results within milliseconds. For a long time databases were unsuitable for this task, because they could not search large inverted indexes at this speed: transactional database systems are not made for this use case. However, analytical database systems, can keep up with state-of-the art information retrieval systems. The company Spinque is a good example of this. At Spinque, MonetDB is used as a computation engine for customized search engines.

DuckDB's FTS implementation follows the paper "Old Dogs Are Great at New Tricks". A keen observation there is that advances made to the database system, such as parallelization, will speed up your search engine "for free"!

Alright, enough about the "why", let's get to the "how".

Preparing the Data

The TREC 2004 Robust Retrieval Track has 250 "topics" (search queries) over TREC disks 4 and 5. The data consist of many text files stored in SGML format, along with a corresponding DTD (document type definition) file. This format is rarely used anymore, but it is similar to XML. We will use OpenSP's command line tool `osx` to convert it to XML. Because there are many files, I wrote a Bash script:

```
mkdir -p latimes/xml
for i in $(seq -w 1 9); do
    cat dtds/la.dtd latimes-$i | osx > latimes/xml/latimes-$i.xml
done
```

This sorts the `latimes` files. Repeat for the `fbis`, `cr`, `fr94`, and `ft` files.

To parse the XML I used BeautifulSoup. Each document has a `docno` identifier, and a `text` field. Because the documents do not come from the same source, they differ in what other fields they have. I chose to take all of the fields.

```
import duckdb
import multiprocessing
import pandas as pd
import re
from bs4 import BeautifulSoup as bs
from tqdm import tqdm

# fill variable 'files' with the path to each .xml file that we created here

def process_file(fpath):
    dict_list = []
    with open(fpath, 'r') as f:
        content = f.read()
        bs_content = bs(content, "html.parser")
        # find all 'doc' nodes
```

```
        for doc in bs_content.findChildren('doc', recursive=True):
            row_dict = {}
            for c in doc.findChildren(recursive=True):
                row_dict[c.name] = ''.join(c.findAll(text=True, recursive=False)).trim()
            dict_list.append(row_dict)
    return dict_list

# process documents (in parallel to speed things up)
pool = multiprocessing.Pool(multiprocessing.cpu_count())
list_of_dict_lists = []
for x in tqdm(pool.imap_unordered(process_file, files), total=len(files)):
    list_of_dict_lists.append(x)
pool.close()
pool.join()

# create pandas dataframe from the parsed data
documents_df = pd.DataFrame([x for sublist in list_of_dict_lists for x in sublist])
```

Now that we have a dataframe, we can register it in DuckDB.

```
# create database connection and register the dataframe
con = duckdb.connect(database='db/trec04_05.db', read_only=False)
con.register('documents_df', documents_df)

# create a table from the dataframe so that it persists
con.execute("CREATE TABLE documents AS (SELECT * FROM documents_df)")
con.close()
```

This is the end of my preparation script, so I closed the database connection.

Building the Search Engine

We can now build the inverted index and the retrieval model using a PRAGMA statement. The extension is documented here. We create an index table on table `documents` or `main.documents` that we created with our script. The column that identifies our documents is called `docno`, and we wish to create an inverted index on the fields supplied. I supplied all fields by using the '*' shortcut.

```
con = duckdb.connect(database='db/trec04_05.db', read_only=False)
con.execute("PRAGMA create_fts_index('documents', 'docno', '*', stopwords='english')")
```

Under the hood, a parameterized SQL script is called. The schema `fts_main_documents` is created, along with tables `docs`, `terms`, `dict`, and `stats`, that make up the inverted index. If you're curious what this look like, take a look at our source code under the `extension` folder in DuckDB's source code!

Running the Benchmark

The data is now fully prepared. Now we want to run the queries in the benchmark, one by one. We load the topics file as follows:

```
# the 'topics' file is not structured nicely, therefore we need parse some of it using regex
def after_tag(s, tag):
    m = re.findall(r'<' + tag + r'>([\s\S]*?)<.*>', s)
    return m[0].replace('\n', '').strip()

topic_dict = {}
with open('../../trec/topics', 'r') as f:
    bs_content = bs(f.read(), "lxml")
    for top in bs_content.findChildren('top'):
        top_content = top.getText()
        # we need the number and title of each topic
        num = after_tag(str(top), 'num').split(' ')[1]
        title = after_tag(str(top), 'title')
        topic_dict[num] = title
```

This gives us a dictionary that has query number as keys, and query strings as values, e.g., `301 -> 'International Organized Crime'`.

We want to store the results in a specific format, so that they can be evaluated by trec eval:

```
# create a prepared statement to make querying our document collection easier
con.execute("""
    PREPARE fts_query AS (
        WITH scored_docs AS (
            SELECT *, fts_main_documents.match_bm25(docno, ?) AS score FROM documents)
        SELECT docno, score
        FROM scored_docs
        WHERE score IS NOT NULL
        ORDER BY score DESC
        LIMIT 1000)
    """)

# enable parallelism
con.execute('PRAGMA threads=32')
results = []
for query in topic_dict:
    q_str = topic_dict[query].replace('\'', ' ')
    con.execute("EXECUTE fts_query('" + q_str + "')")
    for i, row in enumerate(con.fetchall()):
        results.append(query + " Q0 " + row[0].trim() + " " + str(i) + " " + str(row[1]) + " STANDARD")
con.close()

with open('results', 'w+') as f:
    for r in results:
        f.write(r + '\n')
```

Results

Now that we have created our 'results' file, we can compare them to the relevance assessments `qrels` using `trec_eval`.

```
./trec_eval -m P.30 -m map qrels results
```

```
map                     all 0.2324
P_30                    all 0.2948
```

Not bad! While these results are not as high as the reproducible by Anserini, they are definitely acceptable. The difference in performance can be explained by differences in

1. Which stemmer was used (we used 'porter')
2. Which stopwords were used (we used the list of 571 English stopwords used in the SMART system)
3. Pre-processing (removal of accents, punctuation, numbers)
4. BM25 parameters (we used the default k=1.2 and b=0.75, non-conjunctive)
5. Which fields were indexed (we used all columns by supplying '*')

Retrieval time for each query was between 0.5 and 1.3 seconds on our machine, which will be improved with further improvements to DuckDB. I hope you enjoyed reading this blog, and become inspired to test out the extension as well!

Efficient SQL on Pandas with DuckDB

Publication date: 2021-05-14

Authors: Mark Raasveldt, Hannes Mühleisen

TL;DR: DuckDB, a free and open source analytical data management system, can efficiently run SQL queries directly on Pandas DataFrames.

Recently, an article was published advocating for using SQL for Data Analysis. Here at team DuckDB, we are huge fans of SQL. It is a versatile and flexible language that allows the user to efficiently perform a wide variety of data transformations, without having to care about how the data is physically represented or how to do these data transformations in the most optimal way.

While you can very effectively perform aggregations and data transformations in an external database system such as Postgres if your data is stored there, at some point you will need to convert that data back into Pandas and NumPy. These libraries serve as the standard for data exchange between the vast ecosystem of Data Science libraries in Python1 such as scikit-learn or TensorFlow.

1Apache Arrow is gaining significant traction in this domain as well, and DuckDB also quacks Arrow.

If you are reading from a file (e.g., a CSV or Parquet file) often your data will never be loaded into an external database system at all, and will instead be directly loaded into a Pandas DataFrame.

SQL on Pandas

After your data has been converted into a Pandas DataFrame often additional data wrangling and analysis still need to be performed. SQL is a very powerful tool for performing these types of data transformations. Using DuckDB, it is possible to run SQL efficiently right on top of Pandas DataFrames.

As a short teaser, here is a code snippet that allows you to do exactly that: run arbitrary SQL queries directly on Pandas DataFrames using DuckDB.

```
# to install: pip install duckdb
import pandas as pd
import duckdb

mydf = pd.DataFrame({'a' : [1, 2, 3]})
print(duckdb.query("SELECT sum(a) FROM mydf").to_df())
```

In the rest of the article, we will go more in-depth into how this works and how fast it is.

Data Integration & SQL on Pandas

One of the core goals of DuckDB is that accessing data in common formats should be easy. DuckDB is fully capable of running queries in parallel *directly* on top of a Pandas DataFrame (or on a Parquet/CSV file, or on an Arrow table, …). A separate (time-consuming) import step is not necessary.

DuckDB can also write query results directly to any of these formats. You can use DuckDB to process a Pandas DataFrame in parallel using SQL, and convert the result back to a Pandas DataFrame again, so you can then use the result in other Data Science libraries.

When you run a query in SQL, DuckDB will look for Python variables whose name matches the table names in your query and automatically start reading your Pandas DataFrames. Looking back at the previous example we can see this in action:

```
import pandas as pd
import duckdb

mydf = pd.DataFrame({'a' : [1, 2, 3]})
print(duckdb.query("SELECT sum(a) FROM mydf").to_df())
```

The SQL table name `mydf` is interpreted as the local Python variable `mydf` that happens to be a Pandas DataFrame, which DuckDB can read and query directly. The column names and types are also extracted automatically from the DataFrame.

Not only is this process painless, it is highly efficient. For many queries, you can use DuckDB to process data faster than Pandas, and with a much lower total memory usage, *without ever leaving the Pandas DataFrame binary format* ("Pandas-in, Pandas-out"). Unlike when using an external database system such as Postgres, the data transfer time of the input or the output is negligible (see Appendix A for details).

SQL on Pandas Performance

To demonstrate the performance of DuckDB when executing SQL on Pandas DataFrames, we now present a number of benchmarks. The source code for the benchmarks is available for interactive use in Google Colab. In these benchmarks, we operate *purely* on Pandas DataFrames. Both the DuckDB code and the Pandas code operates fully on a `Pandas-in, Pandas-out` basis.

Benchmark Setup and Data Set

We run the benchmark entirely from within the Google Colab environment. For our benchmark dataset, we use the infamous TPC-H data set. Specifically, we focus on the `lineitem` and `orders` tables as these are the largest tables in the benchmark. The total dataset size is around 1 GB in uncompressed CSV format ("scale factor" 1).

As DuckDB is capable of using multiple processors (multi-threading), we include both a single-threaded variant and a variant with two threads. Note that while DuckDB can scale far beyond two threads, Google Colab only supports two.

Setup

First we need to install DuckDB. This is a simple one-liner.

```
pip install duckdb
```

To set up the dataset for processing we download two Parquet files using `wget`. After that, we load the data into a Pandas DataFrame using the built-in Parquet reader of DuckDB. The system automatically infers that we are reading a Parquet file by looking at the `.parquet` extension of the file.

```
lineitem = duckdb.query(
    "SELECT * FROM 'lineitemsf1.snappy.parquet'"
).to_df()
orders = duckdb.query(
    "SELECT * FROM 'orders.parquet'"
).to_df()
```

Ungrouped Aggregates

For our first query, we will run a set of ungrouped aggregates over the Pandas DataFrame. Here is the SQL query:

```
SELECT
    sum(l_extendedprice),
    min(l_extendedprice),
    max(l_extendedprice),
    avg(l_extendedprice)
FROM lineitem;
```

The Pandas code looks similar:

```
lineitem.agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

Name	Time (s)
DuckDB (1 Thread)	0.079
DuckDB (2 Threads)	0.048
Pandas	0.070

This benchmark involves a very simple query, and Pandas performs very well here. These simple queries are where Pandas excels (ha), as it can directly call into the numpy routines that implement these aggregates, which are highly efficient. Nevertheless, we can see that DuckDB performs similar to Pandas in the single-threaded scenario, and benefits from its multi-threading support when enabled.

Grouped Aggregate

For our second query, we will run the same set of aggregates, but this time include a grouping condition. In SQL, we can do this by adding a GROUP BY clause to the query.

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_extendedprice),
    min(l_extendedprice),
    max(l_extendedprice),
    avg(l_extendedprice)
FROM lineitem
GROUP BY
    l_returnflag,
    l_linestatus;
```

In Pandas, we use the groupby function before we perform the aggregation.

```
lineitem.groupby(
  ['l_returnflag', 'l_linestatus']
).agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

Name	Time (s)
DuckDB (1 Thread)	0.43
DuckDB (2 Threads)	0.32
Pandas	0.84

This query is already getting more complex, and while Pandas does a decent job, it is a factor two slower than the single-threaded version of DuckDB. DuckDB has a highly optimized aggregate hash-table implementation that will perform both the grouping and the computation of all the aggregates in a single pass over the data.

Grouped Aggregate with a Filter

Now suppose that we don't want to perform an aggregate over all of the data, but instead only want to select a subset of the data to aggregate. We can do this by adding a filter clause that removes any tuples we are not interested in. In SQL, we can accomplish this through the `WHERE` clause.

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_extendedprice),
    min(l_extendedprice),
    max(l_extendedprice),
    avg(l_extendedprice)
FROM lineitem
WHERE
    l_shipdate <= DATE '1998-09-02'
GROUP BY
    l_returnflag,
    l_linestatus;
```

In Pandas, we can create a filtered variant of the DataFrame by using the selection brackets.

```
# filter out the rows
filtered_df = lineitem[
  lineitem['l_shipdate'] < "1998-09-02"]
# perform the aggregate
result = filtered_df.groupby(
  ['l_returnflag', 'l_linestatus']
).agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

In DuckDB, the query optimizer will combine the filter and aggregation into a single pass over the data, only reading relevant columns. In Pandas, however, we have no such luck. The filter as it is executed will actually subset the entire lineitem table, *including any columns we are not using!* As a result of this, the filter operation is much more time-consuming than it needs to be.

We can manually perform this optimization ("projection pushdown" in database literature). To do this, we first need to select only the columns that are relevant to our query and then subset the lineitem dataframe. We will end up with the following code snippet:

```
# projection pushdown
pushed_down_df = lineitem[
  ['l_shipdate',
   'l_returnflag',
   'l_linestatus',
   'l_extendedprice']
]
# perform the filter
filtered_df = pushed_down_df[
  pushed_down_df['l_shipdate'] < "1998-09-02"]
# perform the aggregate
result = filtered_df.groupby(
  ['l_returnflag', 'l_linestatus']
).agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

Name	Time (s)
DuckDB (1 Thread)	0.60
DuckDB (2 Threads)	0.42
Pandas	3.57
Pandas (manual pushdown)	2.23

While the manual projection pushdown significantly speeds up the query in Pandas, there is still a significant time penalty for the filtered aggregate. To process a filter, Pandas will write a copy of the entire DataFrame (minus the filtered out rows) back into memory. This operation can be time consuming when the filter is not very selective.

Due to its holistic query optimizer and efficient query processor, DuckDB performs significantly better on this query.

Joins

For the final query, we will join (`merge` in Pandas) the lineitem table with the orders table, and apply a filter that only selects orders which have the status we are interested in. This leads us to the following query in SQL:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_extendedprice),
    min(l_extendedprice),
    max(l_extendedprice),
    avg(l_extendedprice)
FROM lineitem
JOIN orders ON (l_orderkey = o_orderkey)
WHERE l_shipdate <= DATE '1998-09-02'
  AND o_orderstatus='O'
GROUP BY
    l_returnflag,
    l_linestatus;
```

For Pandas, we have to add a `merge` step. In a basic approach, we merge lineitem and orders together, then apply the filters, and finally apply the grouping and aggregation. This will give us the following code snippet:

```
# perform the join
merged = lineitem.merge(
  orders,
  left_on='l_orderkey',
  right_on='o_orderkey')
# filter out the rows
filtered_a = merged[
  merged['l_shipdate'] < "1998-09-02"]
filtered_b = filtered_a[
  filtered_a['o_orderstatus'] == "O"]
# perform the aggregate
result = filtered_b.groupby(
  ['l_returnflag', 'l_linestatus']
).agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

Now we have missed two performance opportunities:

- First, we are merging far too many columns, because we are merging columns that are not required for the remainder of the query (projection pushdown).

- Second, we are merging far too many rows. We can apply the filters prior to the merge to reduce the amount of data that we need to merge (filter pushdown).

Applying these two optimizations manually results in the following code snippet:

```
# projection & filter on lineitem table
lineitem_projected = lineitem[
  ['l_shipdate',
   'l_orderkey',
   'l_linestatus',
   'l_returnflag',
   'l_extendedprice']
]
lineitem_filtered = lineitem_projected[
  lineitem_projected['l_shipdate'] < "1998-09-02"]
# projection and filter on order table
orders_projected = orders[
  ['o_orderkey',
   'o_orderstatus']
]
orders_filtered = orders_projected[
  orders_projected['o_orderstatus'] == 'O']
# perform the join
merged = lineitem_filtered.merge(
  orders_filtered,
  left_on='l_orderkey',
  right_on='o_orderkey')
# perform the aggregate
result = merged.groupby(
  ['l_returnflag', 'l_linestatus']
).agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

Both of these optimizations are automatically applied by DuckDB's query optimizer.

Name	Time (s)
DuckDB (1 Thread)	1.05
DuckDB (2 Threads)	0.53
Pandas	15.2
Pandas (manual pushdown)	3.78

We see that the basic approach is extremely time consuming compared to the optimized version. This demonstrates the usefulness of the automatic query optimizer. Even after optimizing, the Pandas code is still significantly slower than DuckDB because it stores intermediate results in memory after the individual filters and joins.

Takeaway

Using DuckDB, you can take advantage of the powerful and expressive SQL language without having to worry about moving your data in – and out – of Pandas. DuckDB is extremely simple to install, and offers many advantages such as a query optimizer, automatic multi-threading and larger-than-memory computation. DuckDB uses the Postgres SQL parser, and offers many of the same SQL features as Postgres, including advanced features such as window functions, correlated subqueries, (recursive) common table expressions, nested types and sampling. If you are missing a feature, please open an issue.

Appendix A: There and Back Again: Transferring Data from Pandas to a SQL Engine and Back

Traditional SQL engines use the Client-Server paradigm, which means that a client program connects through a socket to a server. Queries are run on the server, and results are sent back down to the client afterwards. This is the same when using for example Postgres from Python. Unfortunately, this transfer is a serious bottleneck. In-process engines such as SQLite or DuckDB do not run into this problem.

To showcase how costly this data transfer over a socket is, we have run a benchmark involving Postgres, SQLite and DuckDB. The source code for the benchmark can be found on GitHub.

In this benchmark we copy a (fairly small) Pandas data frame consisting of 10M 4-byte integers (40 MB) from Python to the PostgreSQL, SQLite and DuckDB databases. Since the default Pandas `to_sql` was rather slow, we added a separate optimization in which we tell Pandas to write the data frame to a temporary CSV file, and then tell PostgreSQL to directly copy data from that file into a newly created table. This of course will only work if the database server is running on the same machine as Python.

Name	Time (s)
Pandas to Postgres using to_sql	111.25
Pandas to Postgres using temporary CSV file	5.57
Pandas to SQLite using to_sql	6.80
Pandas to DuckDB	0.03

While SQLite performs significantly better than Postgres here, it is still rather slow. That is because the `to_sql` function in Pandas runs a large number of `INSERT INTO` statements, which involves transforming all the individual values of the Pandas DataFrame into a row-wise representation of Python objects which are then passed onto the system. DuckDB on the other hand directly reads the underlying array from Pandas, which makes this operation almost instant.

Transferring query results or tables back from the SQL system into Pandas is another potential bottleneck. Using the built-in `read_sql_query` is extremely slow, but even the more optimized CSV route still takes at least a second for this tiny data set. DuckDB, on the other hand, also performs this transformation almost instantaneously.

Name	Time (s)
PostgreSQL to Pandas using read_sql_query	7.08
PostgreSQL to Pandas using temporary CSV file	1.29
SQLite to Pandas using read_sql_query	5.20
DuckDB to Pandas	0.04

Appendix B: Comparison to PandaSQL

There is a package called PandaSQL that also provides the facilities of running SQL directly on top of Pandas. However, it is built using the to_sql and from_sql infrastructure that we have seen is extremely slow in Appendix A.

Nevertheless, for good measure we have run the first Ungrouped Aggregate query in PandaSQL to time it. When we first tried to run the query on the original dataset, however, we ran into an out-of-memory error that crashed our colab session. For that reason, we have decided to run the benchmark again for PandaSQL using a sample of 10% of the original data set size (600K rows). Here are the results:

Name	Time (s)
DuckDB (1 Thread)	0.023
DuckDB (2 Threads)	0.014

Name	Time (s)
Pandas	0.017
PandaSQL	24.43

We can see that PandaSQL (powered by SQLite) is around 1000× slower than either Pandas or DuckDB on this straightforward benchmark. The performance difference was so large we have opted not to run the other benchmarks for PandaSQL.

Appendix C: Query on Parquet Directly

In the benchmarks above, we fully read the Parquet files into Pandas. However, DuckDB also has the capability of directly running queries on top of Parquet files (in parallel!). In this appendix, we show the performance of this compared to loading the file into Python first.

For the benchmark, we will run two queries: the simplest query (the ungrouped aggregate) and the most complex query (the final join) and compare the cost of running this query directly on the Parquet file, compared to loading it into Pandas using the `read_parquet` function.

Setup

In DuckDB, we can create a view over the Parquet file using the following query. This allows us to run queries over the Parquet file as if it was a regular table. Note that we do not need to worry about projection pushdown at all: we can just do a `SELECT *` and DuckDB's optimizer will take care of only projecting the required columns at query time.

```
CREATE VIEW lineitem_parquet AS
    SELECT * FROM 'lineitemsf1.snappy.parquet';
CREATE VIEW orders_parquet AS
    SELECT * FROM 'orders.parquet';
```

Ungrouped Aggregate

After we have set up this view, we can run the same queries we ran before, but this time against the `lineitem_parquet` table.

```
SELECT sum(l_extendedprice), min(l_extendedprice), max(l_extendedprice), avg(l_extendedprice) FROM
lineitem_parquet;
```

For Pandas, we will first need to run `read_parquet` to load the data into Pandas. To do this, we use the Parquet reader powered by Apache Arrow. After that, we can run the query as we did before.

```
lineitem_pandas_parquet = pd.read_parquet('lineitemsf1.snappy.parquet')
result = lineitem_pandas_parquet.agg(Sum=('l_extendedprice', 'sum'), Min=('l_extendedprice', 'min'),
Max=('l_extendedprice', 'max'), Avg=('l_extendedprice', 'mean'))
```

However, we now again run into the problem where Pandas will read the Parquet file in its entirety. In order to circumvent this, we will need to perform projection pushdown manually again by providing the `read_parquet` method with the set of columns that we want to read.

The optimizer in DuckDB will figure this out by itself by looking at the query you are executing.

```
lineitem_pandas_parquet = pd.read_parquet('lineitemsf1.snappy.parquet', columns=['l_extendedprice'])
result = lineitem_pandas_parquet.agg(Sum=('l_extendedprice', 'sum'), Min=('l_extendedprice', 'min'),
Max=('l_extendedprice', 'max'), Avg=('l_extendedprice', 'mean'))
```

Name	Time (s)
DuckDB (1 Thread)	0.16

Name	Time (s)
DuckDB (2 Threads)	0.14
Pandas	7.87
Pandas (manual pushdown)	0.17

We can see that the performance difference between doing the pushdown and not doing the pushdown is dramatic. When we perform the pushdown, Pandas has performance in the same ballpark as DuckDB. Without the pushdown, however, it is loading the entire file from disk, including the other 15 columns that are not required to answer the query.

Joins

Now for the final query that we saw in the join section previously. To recap:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_extendedprice),
    min(l_extendedprice),
    max(l_extendedprice),
    avg(l_extendedprice)
FROM lineitem
JOIN orders ON (l_orderkey = o_orderkey)
WHERE l_shipdate <= DATE '1998-09-02'
  AND o_orderstatus='O'
GROUP BY
    l_returnflag,
    l_linestatus;
```

For Pandas we again create two versions. A naive version, and a manually optimized version. The exact code used can be found in Google Colab.

Name	Time (s)
DuckDB (1 Thread)	1.04
DuckDB (2 Threads)	0.89
Pandas	20.4
Pandas (manual pushdown)	3.95

We see that for this more complex query the slight difference in performance between running over a Pandas DataFrame and a Parquet file vanishes, and the DuckDB timings become extremely similar to the timings we saw before. The added Parquet read again increases the necessity of manually performing optimizations on the Pandas code, which is not required at all when running SQL in DuckDB.

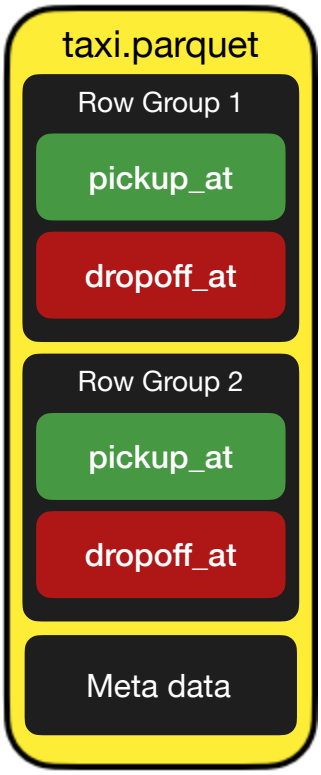
Querying Parquet with Precision Using DuckDB

Publication date: 2021-06-25

Authors: Hannes Mühleisen, Mark Raasveldt

TL;DR: DuckDB, a free and open source analytical data management system, can run SQL queries directly on Parquet files and automatically take advantage of the advanced features of the Parquet format.

Apache Parquet is the most common "Big Data" storage format for analytics. In Parquet files, data is stored in a columnar-compressed binary format. Each Parquet file stores a single table. The table is partitioned into row groups, which each contain a subset of the rows of the table. Within a row group, the table data is stored in a columnar fashion.



The Parquet format has a number of properties that make it suitable for analytical use cases:

1. The columnar representation means that individual columns can be (efficiently) read. No need to always read the entire file!
2. The file contains per-column statistics in every row group (min/max value, and the number of NULL values). These statistics allow the reader to skip row groups if they are not required.
3. The columnar compression significantly reduces the file size of the format, which in turn reduces the storage requirement of data sets. This can often turn Big Data into Medium Data.

DuckDB and Parquet

DuckDB's zero-dependency Parquet reader is able to directly execute SQL queries on Parquet files without any import or analysis step. Because of the natural columnar format of Parquet, this is very fast!

DuckDB will read the Parquet files in a streaming fashion, which means you can perform queries on large Parquet files that do not fit in your main memory.

DuckDB is able to automatically detect which columns and rows are required for any given query. This allows users to analyze much larger and more complex Parquet files without needing to perform manual optimizations or investing in more hardware.

And as an added bonus, DuckDB is able to do all of this using parallel processing and over multiple Parquet files at the same time using the glob syntax.

As a short teaser, here is a code snippet that allows you to directly run a SQL query on top of a Parquet file.

To install the DuckDB package:

```
pip install duckdb
```

To download the Parquet file:

```
wget https://blobs.duckdb.org/data/taxi_2019_04.parquet
```

Then, run the following Python script:

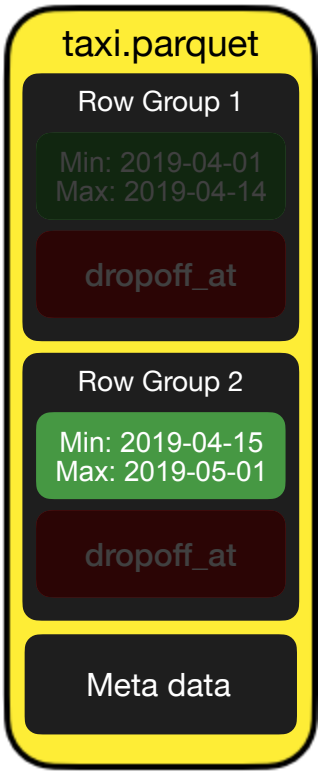
```
import duckdb

print(duckdb.query('''
   SELECT count(*)
   FROM 'taxi_2019_04.parquet'
   WHERE pickup_at BETWEEN '2019-04-15' AND '2019-04-20'
   ''').fetchall())
```

Automatic Filter & Projection Pushdown

Let us dive into the previous query to better understand the power of the Parquet format when combined with DuckDB's query optimizer.

```
SELECT count(*)
FROM 'taxi_2019_04.parquet'
WHERE pickup_at BETWEEN '2019-04-15' AND '2019-04-20';
```

In this query, we read a single column from our Parquet file (`pickup_at`). Any other columns stored in the Parquet file can be entirely skipped, as we do not need them to answer our query.



In addition, only rows that have a `pickup_at` between the 15th and the 20th of April 2019 influence the result of the query. Any rows that do not satisfy this predicate can be skipped.

We can use the statistics inside the Parquet file to great advantage here. Any row groups that have a max value of `pickup_at` lower than `2019-04-15`, or a min value higher than `2019-04-20`, can be skipped. In some cases, that allows us to skip reading entire files.

DuckDB versus Pandas

To illustrate how effective these automatic optimizations are, we will run a number of queries on top of Parquet files using both Pandas and DuckDB.

In these queries, we use a part of the infamous New York Taxi dataset stored as Parquet files, specifically data from April, May and June 2019. These files are ca. 360 MB in size together and contain around 21 million rows of 18 columns each. The three files are placed into the `taxi/` folder.

The examples are available as an interactive notebook over at Google Colab. The timings reported here are from this environment for reproducibility.

Reading Multiple Parquet Files

First we look at some rows in the dataset. There are three Parquet files in the `taxi/` folder. DuckDB supports the globbing syntax, which allows it to query all three files simultaneously.

```
con.execute("""
   SELECT *
   FROM 'taxi/*.parquet'
   LIMIT 5""").df()
```

pickup_at	dropoff_at	passenger_count	trip_distance	rate_code_id
2019-04-01 00:04:09	2019-04-01 00:06:35	1	0.5	1

pickup_at	dropoff_at	passenger_count	trip_distance	rate_code_id
2019-04-01 00:22:45	2019-04-01 00:25:43	1	0.7	1
2019-04-01 00:39:48	2019-04-01 01:19:39	1	10.9	1
2019-04-01 00:35:32	2019-04-01 00:37:11	1	0.2	1
2019-04-01 00:44:05	2019-04-01 00:57:58	1	4.8	1

Despite the query selecting all columns from three (rather large) Parquet files, the query completes instantly. This is because DuckDB processes the Parquet file in a streaming fashion, and will stop reading the Parquet file after the first few rows are read as that is all required to satisfy the query.

If we try to do the same in Pandas, we realize it is not so straightforward, as Pandas cannot read multiple Parquet files in one call. We will first have to use `pandas.concat` to concatenate the three Parquet files together:

```
import pandas
import glob
df = pandas.concat(
    [pandas.read_parquet(file)
     for file
     in glob.glob('taxi/*.parquet')])
print(df.head(5))
```

Below are the timings for both of these queries.

System	Time (s)
DuckDB	0.015
Pandas	12.300

Pandas takes significantly longer to complete this query. That is because Pandas not only needs to read each of the three Parquet files in their entirety, it has to concatenate these three separate Pandas DataFrames together.

Concatenate into a Single File

We can address the concatenation issue by creating a single big Parquet file from the three smaller parts. We can use the `pyarrow` library for this, which has support for reading multiple Parquet files and streaming them into a single large file. Note that the `pyarrow` Parquet reader is the very same Parquet reader that is used by Pandas internally.

```
import pyarrow.parquet as pq

# concatenate all three Parquet files
pq.write_table(pq.ParquetDataset('taxi/').read(), 'alltaxi.parquet', row_group_size=100000)
```

Note that DuckDB also has support for writing Parquet files using the COPY statement.

Querying the Large File

Now let us repeat the previous experiment, but using the single file instead.

```
# DuckDB
con.execute("""
   SELECT *
   FROM 'alltaxi.parquet'
   LIMIT 5""").df()
```

```
# Pandas
pandas.read_parquet('alltaxi.parquet')
      .head(5)
```

System	Time (s)
DuckDB	0.02
Pandas	7.50

We can see that Pandas performs better than before, as the concatenation is avoided. However, the entire file still needs to be read into memory, which takes both a significant amount of time and memory.

For DuckDB it does not really matter how many Parquet files need to be read in a query.

Counting Rows

Now suppose we want to figure out how many rows are in our data set. We can do that using the following code:

```
# DuckDB
con.execute("""
   SELECT count(*)
   FROM 'alltaxi.parquet'
""").df()

# Pandas
len(pandas.read_parquet('alltaxi.parquet'))
```

System	Time (s)
DuckDB	0.015
Pandas	7.500

DuckDB completes the query very quickly, as it automatically recognizes what needs to be read from the Parquet file and minimizes the required reads. Pandas has to read the entire file again, which causes it to take the same amount of time as the previous query.

For this query, we can improve Pandas' time through manual optimization. In order to get a count, we only need a single column from the file. By manually specifying a single column to be read in the `read_parquet` command, we can get the same result but much faster.

```
len(pandas.read_parquet('alltaxi.parquet', columns=['vendor_id']))
```

System	Time (s)
DuckDB	0.015
Pandas	7.500
Pandas (optimized)	1.200

While this is much faster, this still takes more than a second as the entire `vendor_id` column has to be read into memory as a Pandas column only to count the number of rows.

Filtering Rows

It is common to use some sort of filtering predicate to only look at the interesting parts of a data set. For example, imagine we want to know how many taxi rides occur after the 30th of June 2019. We can do that using the following query in DuckDB:

```
con.execute("""
   SELECT count(*)
   FROM 'alltaxi.parquet'
   WHERE pickup_at > '2019-06-30'
""").df()
```

The query completes in 45ms and yields the following result:

count
167022

In Pandas, we can perform the same operation using a naive approach.

```
# pandas naive
len(pandas.read_parquet('alltaxi.parquet')
          .query("pickup_at > '2019-06-30'"))
```

This again reads the entire file into memory, however, causing this query to take 7.5 s. With the manual projection pushdown we can bring this down to 0.9 s. Still significantly higher than DuckDB.

```
# pandas projection pushdown
len(pandas.read_parquet('alltaxi.parquet', columns=['pickup_at'])
          .query("pickup_at > '2019-06-30'"))
```

The pyarrow Parquet reader also allows us to perform filter pushdown into the scan, however. Once we add this we end up with a much more competitive 70ms to complete the query.

```
len(pandas.read_parquet('alltaxi.parquet', columns=['pickup_at'], filters=[('pickup_at', '>',
'2019-06-30')]))
```

System	Time (s)
DuckDB	0.05
Pandas	7.50
Pandas (projection pushdown)	0.90
Pandas (projection & filter pushdown)	0.07

This shows that the results here are not due to DuckDB's Parquet reader being faster than the pyarrow Parquet reader. The reason that DuckDB performs better on these queries is because its optimizers automatically extract all required columns and filters from the SQL query, which then get automatically utilized in the Parquet reader with no manual effort required.

Interestingly, both the pyarrow Parquet reader and DuckDB are significantly faster than performing this operation natively in Pandas on a materialized DataFrame.

```
# read the entire Parquet file into Pandas
df = pandas.read_parquet('alltaxi.parquet')
# run the query natively in Pandas
# note: we only time this part
print(len(df[['pickup_at']].query("pickup_at > '2019-06-30'")))
```

System	Time (s)
DuckDB	0.05
Pandas	7.50
Pandas (projection pushdown)	0.90
Pandas (projection & filter pushdown)	0.07
Pandas (native)	0.26

Aggregates

Finally lets look at a more complex aggregation. Say we want to compute the number of rides per passenger. With DuckDB and SQL, it looks like this:

```
con.execute("""
    SELECT passenger_count, count(*)
    FROM 'alltaxi.parquet'
    GROUP BY passenger_count""").df()
```

The query completes in `220ms` and yields the following result:

passenger_count	count
0	408742
1	15356631
2	3332927
3	944833
4	439066
5	910516
6	546467
7	106
8	72
9	64

For the SQL-averse and as a teaser for a future blog post, DuckDB also has a "Relational API" that allows for a more Python-esque declaration of queries. Here's the equivalent to the above SQL query, that provides the exact same result and performance:

```
con.from_parquet('alltaxi.parquet')
   .aggregate('passenger_count, count(*)')
   .df()
```

Now as a comparison, let's run the same query in Pandas in the same way we did previously.

```
# naive
pandas.read_parquet('alltaxi.parquet')
      .groupby('passenger_count')
      .agg({'passenger_count' : 'count'})

# projection pushdown
pandas.read_parquet('alltaxi.parquet', columns=['passenger_count'])
      .groupby('passenger_count')
      .agg({'passenger_count' : 'count'})
```

```
# native (parquet file pre-loaded into memory)
df.groupby('passenger_count')
  .agg({'passenger_count' : 'count'})
```

System	Time (s)
DuckDB	0.22
Pandas	7.50
Pandas (projection pushdown)	0.58
Pandas (native)	0.51

We can see that DuckDB is faster than Pandas in all three scenarios, without needing to perform any manual optimizations and without needing to load the Parquet file into memory in its entirety.

Conclusion

DuckDB can efficiently run queries directly on top of Parquet files without requiring an initial loading phase. The system will automatically take advantage of all of Parquet's advanced features to speed up query execution.

DuckDB is a free and open source database management system (MIT licensed). It aims to be the SQLite for Analytics, and provides a fast and efficient database system with zero external dependencies. It is available not just for Python, but also for C/C++, R, Java, and more.

Fastest Table Sort in the West – Redesigning DuckDB's Sort

Publication date: 2021-08-27

Author: Laurens Kuiper

TL;DR: DuckDB, a free and open-source analytical data management system, has a new highly efficient parallel sorting implementation that can sort much more data than fits in main memory.

Database systems use sorting for many purposes, the most obvious purpose being when a user adds an ORDER BY clause to their query. Sorting is also used within operators, such as window functions. DuckDB recently improved its sorting implementation, which is now able to sort data in parallel and sort more data than fits in memory. In this post, we will take a look at how DuckDB sorts, and how this compares to other data management systems.

Not interested in the implementation? Jump straight to the experiments!

Sorting Relational Data

Sorting is one of the most well-studied problems in computer science, and it is an important aspect of data management. There are entire communities dedicated to who sorts fastest. Research into sorting algorithms tends to focus on sorting large arrays or key/value pairs. While important, this does not cover how to implement sorting in a database system. There is a lot more to sorting tables than just sorting a large array of integers!

Consider the following example query on a snippet of a TPC-DS table:

```
SELECT c_customer_sk, c_birth_country, c_birth_year
FROM customer
ORDER BY c_birth_country DESC,
         c_birth_year    ASC NULLS LAST;
```

Which yields:

c_customer_sk	c_birth_country	c_birth_year
64760	NETHERLANDS	1991
75011	NETHERLANDS	1992
89949	NETHERLANDS	1992
90766	NETHERLANDS	NULL
42927	GERMANY	1924

In other words: `c_birth_country` is ordered descendingly, and where `c_birth_country` is equal, we sort on `c_birth_year` ascendingly. By specifying NULLS LAST, null values are treated as the lowest value in the `c_birth_year` column. Whole rows are thus reordered, not just the columns in the ORDER BY clause. The columns that are not in the ORDER BY clause we call "payload columns". Therefore, payload column `c_customer_sk` has to be reordered too.

It is easy to implement something that can evaluate the example query using any sorting implementation, for instance, **C++**'s `std::sort`. While `std::sort` is excellent algorithmically, it is still a single-threaded approach that is unable to efficiently sort by multiple columns because function call overhead would quickly dominate sorting time. Below we will discuss why that is.

To achieve good performance when sorting tables, a custom sorting implementation is needed. We are – of course – not the first to implement relational sorting, so we dove into the literature to look for guidance.

In 2006 the famous Goetz Graefe wrote a survey on implementing sorting in database systems. In this survey, he collected many sorting techniques that are known to the community. This is a great guideline if you are about to start implementing sorting for tables.

The cost of sorting is dominated by comparing values and moving data around. Anything that makes these two operations cheaper will have a big impact on the total runtime.

There are two obvious ways to go about implementing a comparator when we have multiple ORDER BY clauses:

1. Loop through the clauses: Compare columns until we find one that is not equal, or until we have compared all columns. This is fairly complex already, as this requires a loop with an if/else inside of it for every single row of data. If we have columnar storage, this comparator has to jump between columns, causing random access in memory.
2. Entirely sort the data by the first clause, then sort by the second clause, but only where the first clause was equal, and so on. This approach is especially inefficient when there are many duplicate values, as it requires multiple passes over the data.

Binary String Comparison

The binary string comparison technique improves sorting performance by simplifying the comparator. It encodes *all* columns in the ORDER BY clause into a single binary sequence that, when compared using `memcmp` will yield the correct overall sorting order. Encoding the data is not free, but since we are using the comparator so much during sorting, it will pay off. Let us take another look at 3 rows of the example:

c_birth_country	c_birth_year
NETHERLANDS	1991
NETHERLANDS	1992
GERMANY	1924

On little-endian hardware, the bytes that represent these values look like this in memory, assuming 32-bit integer representation for the year:

```
c_birth_country
-- NETHERLANDS
01001110 01000101 01010100 01001000 01000101 01010010 01001100 01000001 01001110 01000100 01010011 00000000
-- GERMANY
01000111 01000101 01010010 01001101 01000001 01001110 01011001 00000000

c_birth_year
-- 1991
11000111 00000111 00000000 00000000
-- 1992
11001000 00000111 00000000 00000000
-- 1924
10000100 00000111 00000000 00000000
```

The trick is to convert these to a binary string that encodes the sorting order:

```
-- NETHERLANDS | 1991
10110001 10111010 10101011 10110111 10111010 10101101 10110011 10111110 10110001 10111011 10101100 11111111
10000000 00000000 00000111 11000111
-- NETHERLANDS | 1992
10110001 10111010 10101011 10110111 10111010 10101101 10110011 10111110 10110001 10111011 10101100 11111111
10000000 00000000 00000111 11001000
-- GERMANY     | 1924
10111000 10111010 10101101 10110010 10111110 10110001 10100110 11111111 11111111 11111111 11111111 11111111
10000000 00000000 00000111 10000100
```

The binary string is fixed-size because this makes it much easier to move it around during sorting.

The string “GERMANY” is shorter than “NETHERLANDS”, therefore it is padded with `00000000`'s. All bits in column `c_birth_country` are subsequently inverted because this column is sorted descendingly. If a string is too long we encode its prefix and only look at the whole string if the prefixes are equal.

The bytes in `c_birth_year` are swapped because we need the big-endian representation to encode the sorting order. The first bit is also flipped, to preserve order between positive and negative integers for signed integers. If there are `NULL` values, these must be encoded using an additional byte (not shown in the example).

With this binary string, we can now compare both columns at the same time by comparing only the binary string representation. This can be done with a single `memcmp` in **C++**! The compiler will emit efficient assembly for single function call, even auto-generating SIMD instructions.

This technique solves one of the problems mentioned above, namely the function call overhead when using complex comparators.

Radix Sort

Now that we have a cheap comparator, we have to choose our sorting algorithm. Every computer science student learns about comparison-based sorting algorithms like Quicksort and Merge sort, which have $O(n \log n)$ time complexity, where n is the number of records being sorted.

However, there are also distribution-based sorting algorithms, which typically have a time complexity of $O(n k)$, where k is the width of the sorting key. This class of sorting algorithms scales much better with a larger n because k is constant, whereas $\log n$ is not.

One such algorithm is Radix sort. This algorithm sorts the data by computing the data distribution with Counting sort, multiple times until all digits have been counted.

It may sound counter-intuitive to encode the sorting key columns such that we have a cheap comparator, and then choose a sorting algorithm that does not compare records. However, the encoding is necessary for Radix sort: Binary strings that produce a correct order with `memcmp` will produce a correct order if we do a byte-by-byte Radix sort.

Two-Phase Parallel Sorting

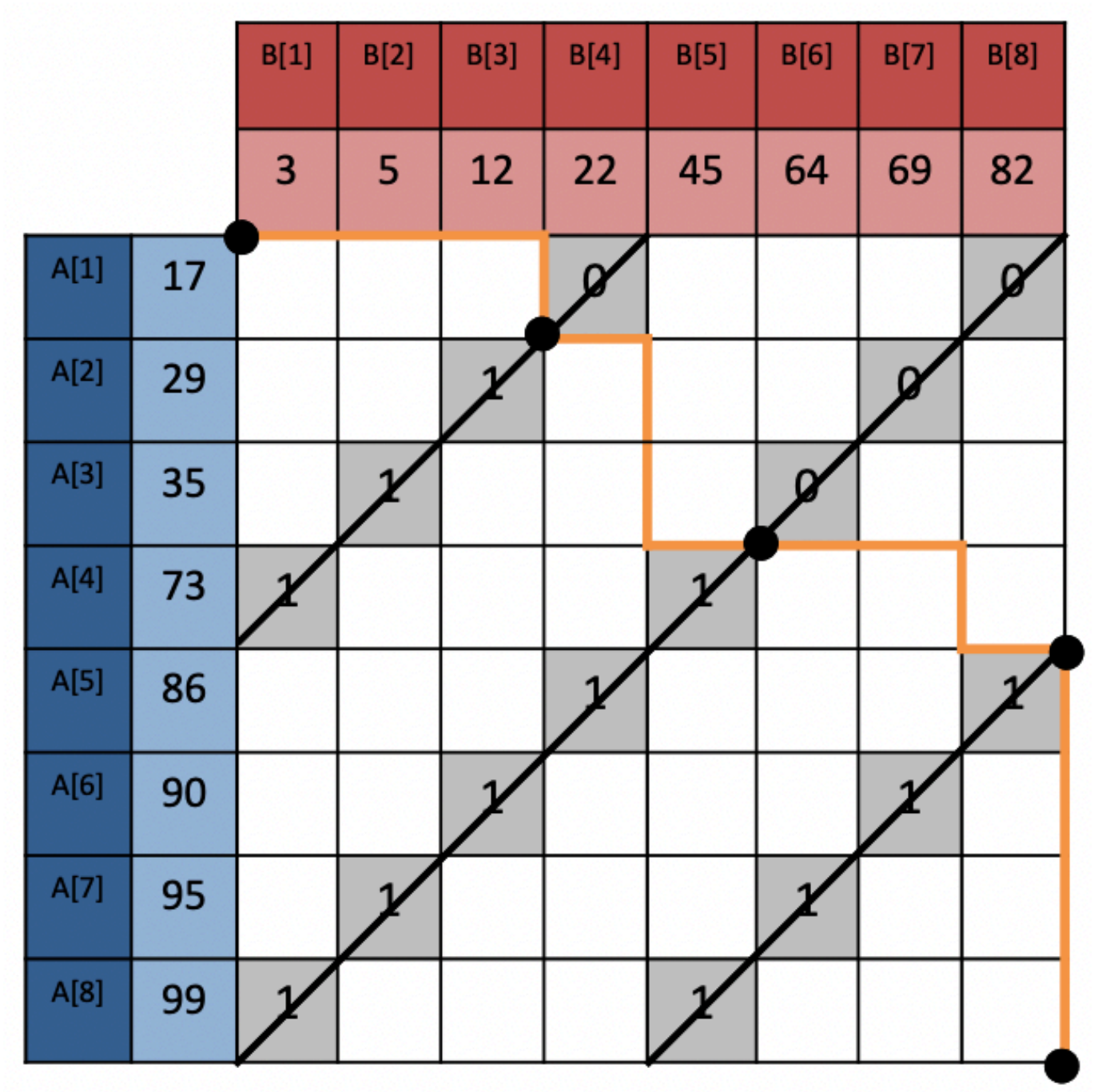
DuckDB uses Morsel-Driven Parallelism, a framework for parallel query execution. For the sorting operator, this means that multiple threads collect roughly an equal amount of data, in parallel, from the table.

We use this parallelism for sorting by first having each thread sort the data it collects using our Radix sort. After this first sorting phase, each thread has one or more sorted blocks of data, which must be combined into the final sorted result. Merge sort is the algorithm of choice for this task. There are two main ways of implementing merge sort: K-way merge and Cascade merge.

K-way merge merges K lists into one sorted list in one pass, and is traditionally used for external sorting (sorting more data than fits in memory) because it minimizes I/O. Cascade merge merges two lists of sorted data at a time until only one sorted list remains, and is used for in-memory sorting because it is more efficient than K-way merge. We aim to have an implementation that has high in-memory performance, which gracefully degrades as we go over the limit of available memory. Therefore, we choose cascade merge.

In a cascade merge sort, we merge two blocks of sorted data at a time until only one sorted block remains. Naturally, we want to use all available threads to compute the merge. If we have many more sorted blocks than threads, we can assign each thread to merge two blocks. However, as the blocks get merged, we will not have enough blocks to keep all threads busy. This is especially slow when the final two blocks are merged: One thread has to process all the data.

To fully parallelize this phase, we have implemented Merge Path by Oded Green et al. Merge Path pre-computes *where* the sorted lists will intersect while merging, shown in the image below (taken from the paper).



The intersections along the merge path can be efficiently computed using Binary Search. If we know where the intersections are, we can merge partitions of the sorted data independently in parallel. This allows us to use all available threads effectively for the entire merge phase. For another trick to improve merge sort, see the appendix.

Columns or Rows?

Besides comparisons, the other big cost of sorting is moving data around. DuckDB has a vectorized execution engine. Data is stored in a columnar layout, which is processed in batches (called chunks) at a time. This layout is great for analytical query processing because the chunks fit in the CPU cache, and it gives a lot of opportunities for the compiler to generate SIMD instructions. However, when the table is sorted, entire rows are shuffled around, rather than columns.

We could stick to the columnar layout while sorting: Sort the key columns, then re-order the payload columns one by one. However, re-ordering will cause a random access pattern in memory for each column. If there are many payload columns, this will be slow. Converting

the columns to rows will make re-ordering rows much easier. This conversion is of course not free: Columns need to be copied to rows, and back from rows to columns again after sorting.

Because we want to support external sorting, we have to store data in buffer-managed blocks that can be offloaded to disk. Because we have to copy the input data to these blocks anyway, converting the rows to columns is effectively free.

There are a few operators that are inherently row-based, such as joins and aggregations. DuckDB has a unified internal row layout for these operators, and we decided to use it for the sorting operator as well. This layout has only been used in memory so far. In the next section, we will explain how we got it to work on disk as well. We should note that we will only write sorting data to disk if main memory is not able to hold it.

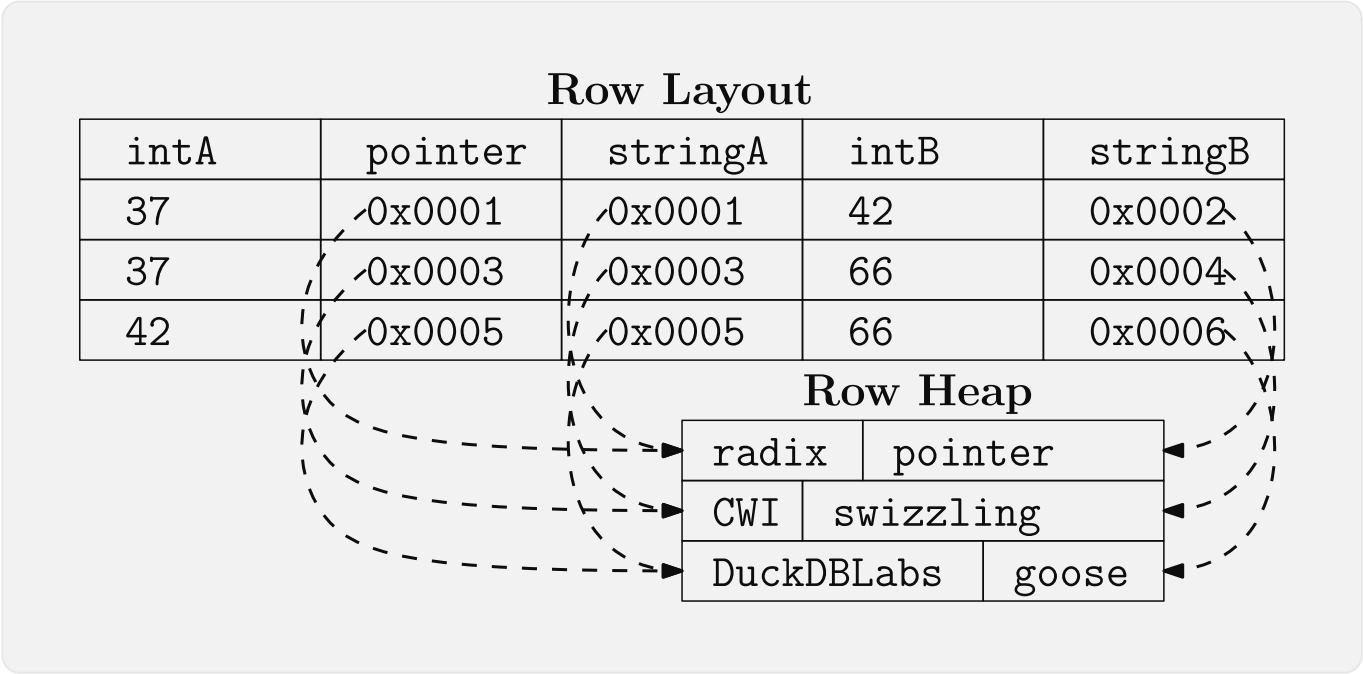
External Sorting

The buffer manager can unload blocks from memory to disk. This is not something we actively do in our sorting implementation, but rather something that the buffer manager decides to do if memory would fill up otherwise. It uses a least-recently-used queue to decide which blocks to write. More on how to properly use this queue in the appendix.

When we need a block, we "pin" it, which reads it from disk if it is not loaded already. Accessing disk is much slower than accessing memory, therefore it is crucial that we minimize the number of reads and writes.

Unloading data to disk is easy for fixed-size columns like integers, but more difficult for variable-sized columns like strings. Our row layout uses fixed-size rows, which cannot fit strings with arbitrary sizes. Therefore, strings are represented by a pointer, which points into a separate block of memory where the actual string data lives, a so-called "string heap".

We have changed our heap to also store strings row-by-row in buffer-managed blocks:



Each row has an additional 8-byte field `pointer` which points to the start of this row in the heap. This is useless in the in-memory representation, but we will see why it is useful for the on-disk representation in just a second.

If the data fits in memory, the heap blocks stay pinned, and only the fixed-size rows are re-ordered while sorting. If the data does not fit in memory, the blocks need to be offloaded to disk, and the heap will also be re-ordered while sorting. When a heap block is offloaded to disk, the pointers pointing into it are invalidated. When we load the block back into memory, the pointers will have changed.

This is where our row-wise layout comes into play. The 8-byte `pointer` field is overwritten with an 8-byte `offset` field, denoting where in the heap block strings of this row can be found. This technique is called "pointer swizzling". When we swizzle the pointers, the row layout and heap block look like this:

Row Layout

intA	offset	stringA	intB	stringB
37	0	0	42	5
37	12	0	66	3
42	24	0	66	10

Row Heap

radix	pointer
CWI	swizzling
DuckDBLabs	goose

The pointers to the subsequent string values are also overwritten with an 8-byte relative offset, denoting how far this string is offset from the start of the row in the heap (hence every `stringA` has an offset of `0`: It is the first string in the row). Using relative offsets within rows rather than absolute offsets is very useful during sorting, as these relative offsets stay constant, and do not need to be updated when a row is copied.

When the blocks need to be scanned to read the sorted result, we "unswizzle" the pointers, making them point to the string again.

With this dual-purpose row-wise representation, we can easily copy around both the fixed-size rows and the variable-sized rows in the heap. Besides having the buffer manager load/unload blocks, the only difference between in-memory and external sorting is that we swizzle/unswizzle pointers to the heap blocks, and copy data from the heap blocks during merge sort.

All this reduces overhead when blocks need to be moved in and out of memory, which will lead to graceful performance degradation as we approach the limit of available memory.

Comparison with Other Systems

Now that we have covered most of the techniques that are used in our sorting implementation, we want to know how we compare to other systems. DuckDB is often used for interactive data analysis, and is therefore often compared to tools like dplyr.

In this setting, people are usually running on laptops or PCs, therefore we will run these experiments on a 2020 MacBook Pro. This laptop has an Apple M1 CPU, which is ARM-based. The M1 processor has 8 cores: 4 high-performance (Firestorm) cores, and 4 energy-efficient (Icestorm) cores. The Firestorm cores have very, very fast single-thread performance, so this should level the playing field between single- and multi-threaded sorting implementations somewhat. The MacBook has 16 GB of memory, and one of the fastest SSDs found in a laptop.

We will be comparing against the following systems:

1. ClickHouse, version 21.7.5
2. HyPer, version 2021.2.1.12564
3. Pandas, version 1.3.2
4. SQLite, version 3.36.0

ClickHouse and HyPer are included in our comparison because they are analytical SQL engines with an emphasis on performance. Pandas and SQLite are included in our comparison because they can be used to perform relational operations within Python, like DuckDB. Pandas operates fully in memory, whereas SQLite is a more traditional disk-based system. This list of systems should give us a good mix of single-/multi-threaded, and in-memory/external sorting.

ClickHouse was built for M1 using this guide. We have set the memory limit to 12 GB, and `max_bytes_before_external_sort` to 10 GB, following this suggestion.

HyPer is Tableau's data engine, created by the database group at the University of Munich. It does not run natively (yet) on ARM-based processors like the M1. We will use Rosetta 2, macOS's x86 emulator to run it. Emulation causes some overhead, so we have included an experiment on an x86 machine in the appendix.

Benchmarking sorting in database systems is not straightforward. Ideally, we would like to measure only the time it takes to sort the data, not the time it takes to read the input data and show the output. Not every system has a profiler to measure the time of the sorting operator exactly, so this is not an option.

To approach a fair comparison, we will measure the end-to-end time of queries that sort the data and write the result to a temporary table, i.e.:

```
CREATE TEMPORARY TABLE output AS
SELECT ...
FROM ...
ORDER BY ...;
```

There is no perfect solution to this problem, but this should give us a good comparison because the end-to-end time of this query should be dominated by sorting. For Pandas we will use `sort_values` with `inplace=False` to mimic this query.

In ClickHouse, temporary tables can only exist in memory, which is problematic for our out-of-core experiments. Therefore we will use a regular `TABLE`, but then we also need to choose a table engine. Most of the table engines apply compression or create an index, which we do not want to measure. Therefore we have chosen the simplest on-disk engine, which is File, with format Native.

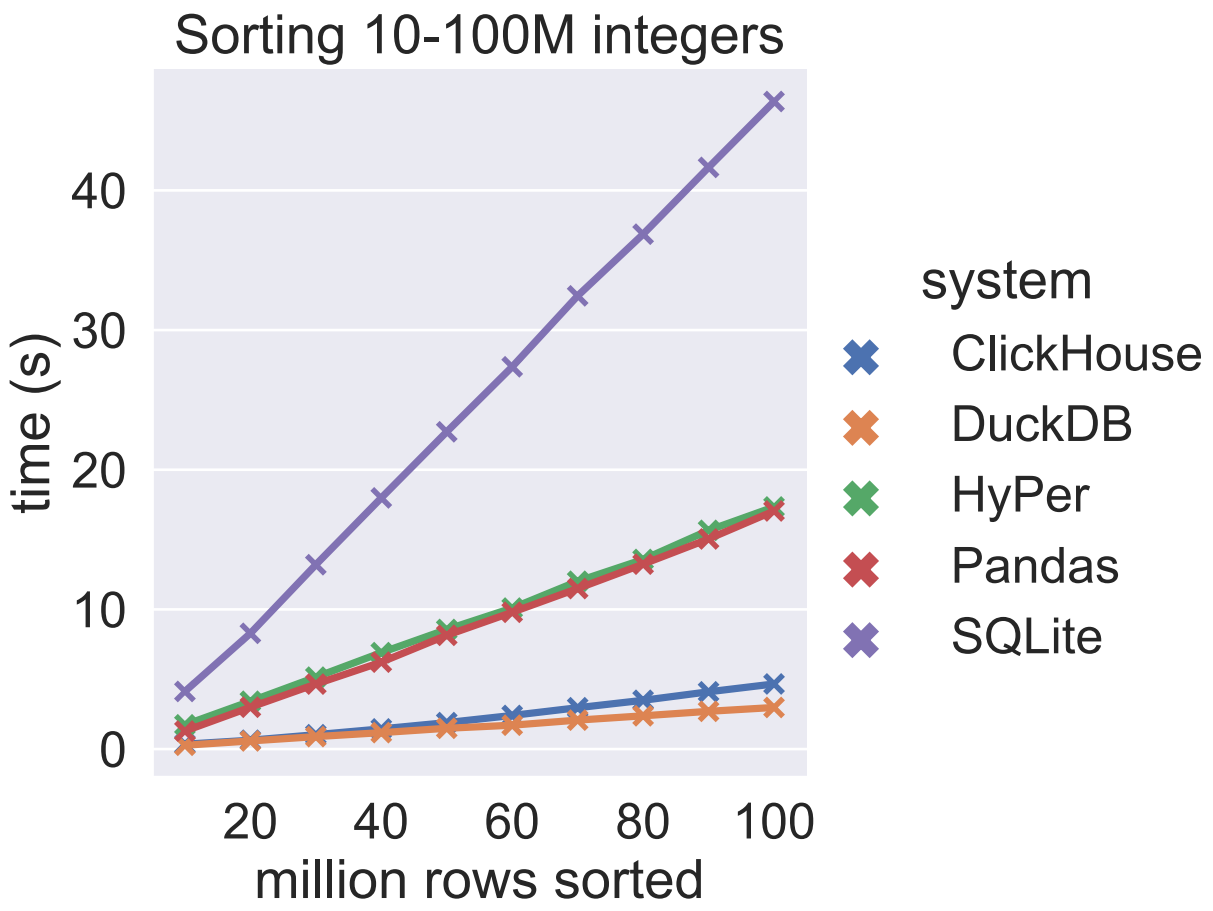
The table engine we chose for the input tables for ClickHouse is MergeTree with `ORDER BY tuple()`. We chose this because we encountered strange behavior with `File(Native)` input tables, where there was no difference in runtime between the queries `SELECT * FROM ... ORDER BY` and `SELECT col1 FROM ... ORDER BY`. Presumably, because all columns in the table were sorted regardless of how many there were selected.

To measure stable end-to-end query time, we run each query 5 times and report the median run time. There are some differences in reading/writing tables between the systems. For instance, Pandas cannot read/write from/to disk, so both the input and output data frame will be in memory. DuckDB will not write the output table to disk unless there is not enough room to keep it in memory, and therefore also may have an advantage. However, sorting dominates the total runtime, so these differences are not that impactful.

Random Integers

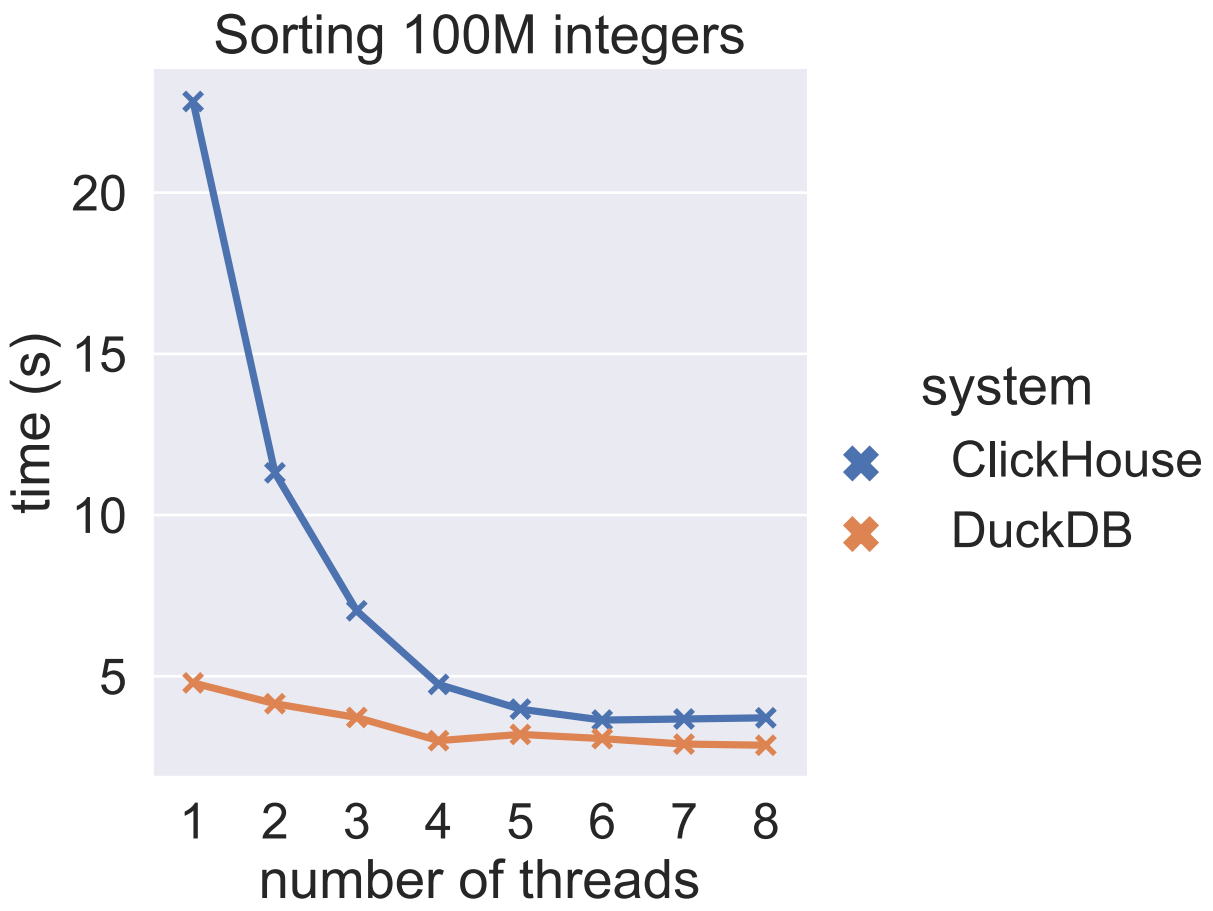
We will start with a simple example. We have generated the first 100 million integers and shuffled them, and we want to know how well the systems can sort them. This experiment is more of a micro-benchmark than anything else and is of little real-world significance.

For our first experiment, we will look at how the systems scale with the number of rows. From the initial table with integers, we have made 9 more tables, with 10M, 20M, ..., 90M integers each.



Being a traditional disk-based database system, SQLite always opts for an external sorting strategy. It writes intermediate sorted blocks to disk even if they fit in main-memory, therefore it is much slower. The performance of the other systems is in the same ballpark, with DuckDB and ClickHouse going toe-to-toe with ~3 and ~4 seconds for 100M integers. Because SQLite is so much slower, we will not include it in our next set of experiments (TPC-DS).

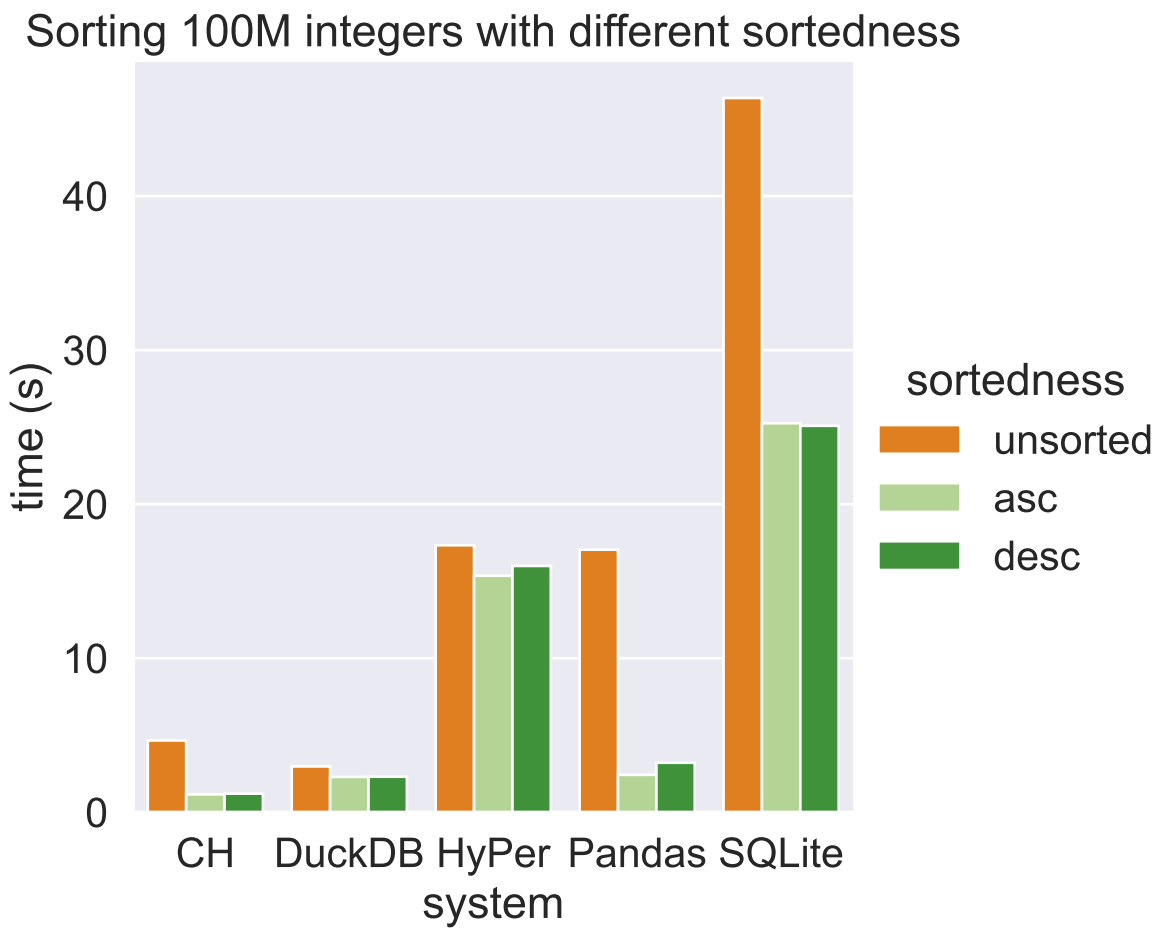
DuckDB and ClickHouse both make very good use out of all available threads, with a single-threaded sort in parallel, followed by a parallel merge sort. We are not sure what strategy HyPer uses. For our next experiment, we will zoom in on multi-threading, and see how well ClickHouse and DuckDB scale with the number of threads (we were not able to set the number of threads for HyPer).



This plot demonstrates that Radix sort is very fast. DuckDB sorts 100M integers in just under 5 seconds using a single thread, which is much faster than ClickHouse. Adding threads does not improve performance as much for DuckDB, because Radix Sort is so much faster than Merge Sort. Both systems end up at about the same performance at 4 threads.

Beyond 4 threads we do not see performance improve much more, due to the CPU architecture. For all of the of other the experiments, we have set both DuckDB and ClickHouse to use 4 threads.

For our last experiment with random integers, we will see how the sortedness of the input may impact performance. This is especially important to do in systems that use Quicksort because Quicksort performs much worse on inversely sorted data than on random data.



Not surprisingly, all systems perform better on sorted data, sometimes by a large margin. ClickHouse, Pandas, and SQLite likely have some optimization here: e.g., keeping track of sortedness in the catalog, or checking sortedness while scanning the input. DuckDB and HyPer have only a very small difference in performance when the input data is sorted, and do not have such an optimization. For DuckDB the slightly improved performance can be explained due to a better memory access pattern during sorting: When the data is already sorted the access pattern is mostly sequential.

Another interesting result is that DuckDB sorts data faster than some of the other systems can read already sorted data.

TPC-DS

For the next comparison, we have improvised a relational sorting benchmark on two tables from the standard TPC Decision Support benchmark (TPC-DS). TPC-DS is challenging for sorting implementations because it has wide tables (with many columns, unlike the tables in TPC-H), and a mix of fixed- and variable-sized types. The number of rows increases with the scale factor. The tables used here are `catalog_sales` and `customer`.

`catalog_sales` has 34 columns, all fixed-size types (integer and double), and grows to have many rows as the scale factor increases. `customer` has 18 columns (10 integers, and 8 strings), and a decent amount of rows as the scale factor increases. The row counts of both tables at each scale factor are shown in the table below.

SF	customer	catalog_sales
1	100,000	1,441,548
10	500,000	14,401,261
100	2,000,000	143,997,065

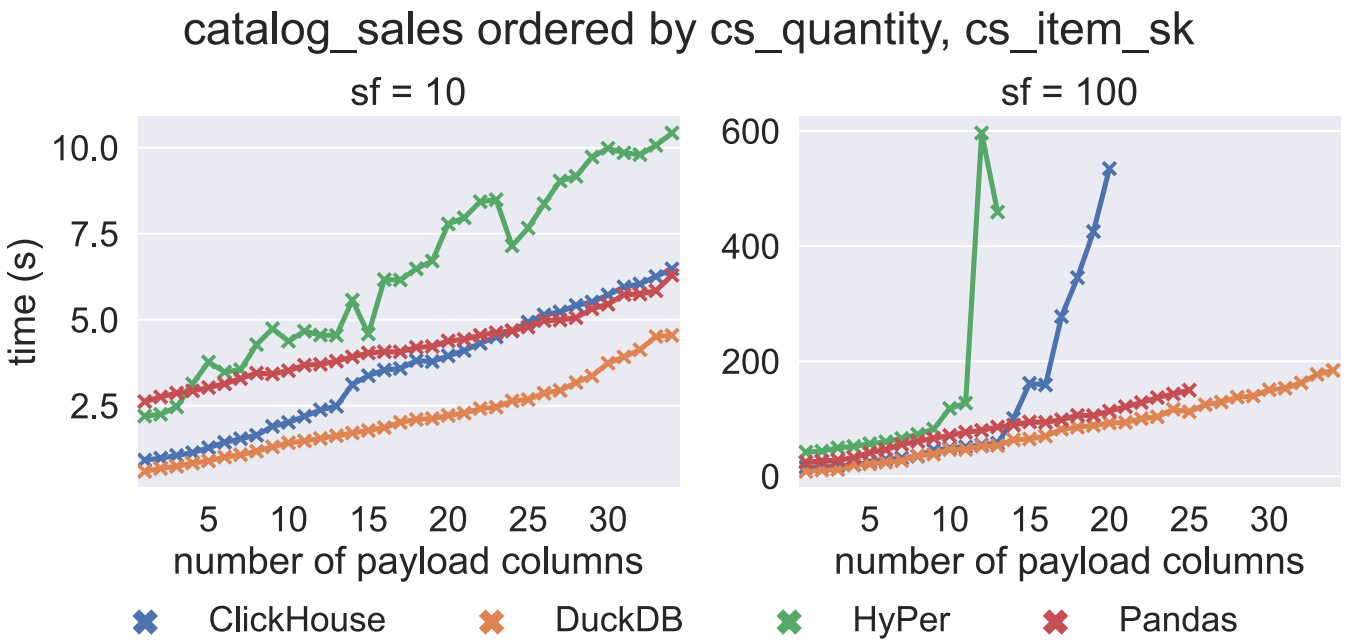
SF	customer	catalog_sales
300	5,000,000	260,014,080

We will use `customer` at SF100 and SF300, which fits in memory at every scale factor. We will use `catalog_sales` table at SF10 and SF100, which does not fit in memory anymore at SF100.

The data was generated using DuckDB's TPC-DS extension, then exported to CSV in a random order to undo any ordering patterns that could have been in the generated data.

Catalog Sales (Numeric Types)

Our first experiment on the `catalog_sales` table is selecting 1 column, then 2 columns, ..., up to all 34, always ordering by `cs_quantity` and `cs_item_sk`. This experiment will tell us how well the different systems can re-order payload columns.



We see similar trends at SF10 and SF100, but for SF100, at around 12 payload columns or so, the data does not fit in memory anymore, and ClickHouse and HyPer show a big drop in performance. ClickHouse switches to an external sorting strategy, which is much slower than its in-memory strategy. Therefore, adding a few payload columns results in a runtime that is orders of magnitude higher. At 20 payload columns ClickHouse runs into the following error:

```
DB::Exception: Memory limit (for query) exceeded: would use 11.18 GiB (attempt to allocate chunk of 4204712 bytes), maximum: 11.18 GiB: (while reading column cs_list_price): (while reading from part ./store/523/5230c288-7ed5-45fa-9230-c2887ed595fa/all_73_108_2/ from mark 4778 with max_rows_to_read = 8192): While executing MergeTreeThread.
```

HyPer also drops in performance before erroring out with the following message:

```
ERROR:  Cannot allocate 333982248 bytes of memory: The `global memory limit` limit of 12884901888 bytes was exceeded.
```

As far as we are aware, HyPer uses `mmap`, which creates a mapping between memory and a file. This allows the operating system to move data between memory and disk. While useful, it is no substitute for a proper external sort, as it creates random access to disk, which is very slow.

Pandas performs surprisingly well on SF100, despite the data not fitting in memory. Pandas can only do this because macOS dynamically increases swap size. Most operating systems do not do this and would fail to load the data at all. Using swap usually slows down processing significantly, but the SSD is so fast that there is no visible performance drop!

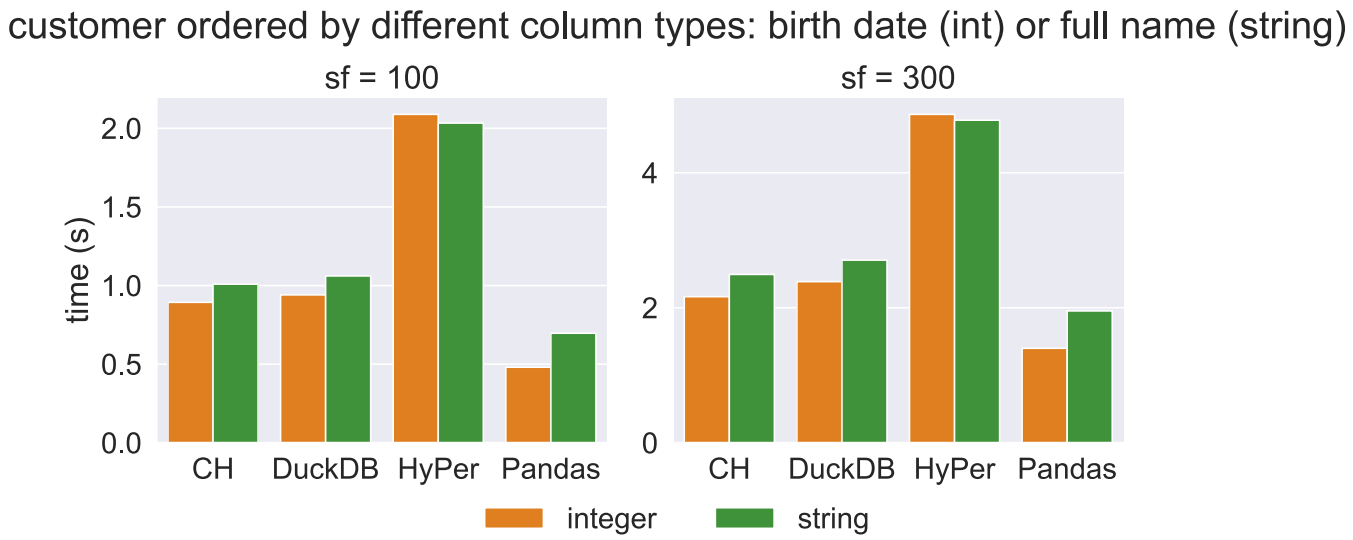
While Pandas loads the data, swap size grows to an impressive ~40 GB: Both the file and the data frame are fully in memory/swap at the same time, rather than streamed into memory. This goes down to ~20 GB of memory/swap when the file is done being read. Pandas is able to get quite far into the experiment until it crashes with the following error:

```
UserWarning: resource_tracker: There appear to be 1 leaked semaphore objects to clean up at shutdown
```

DuckDB performs well both in-memory and external, and there is no clear visible point at which data no longer fits in memory: Runtime is fast and reliable.

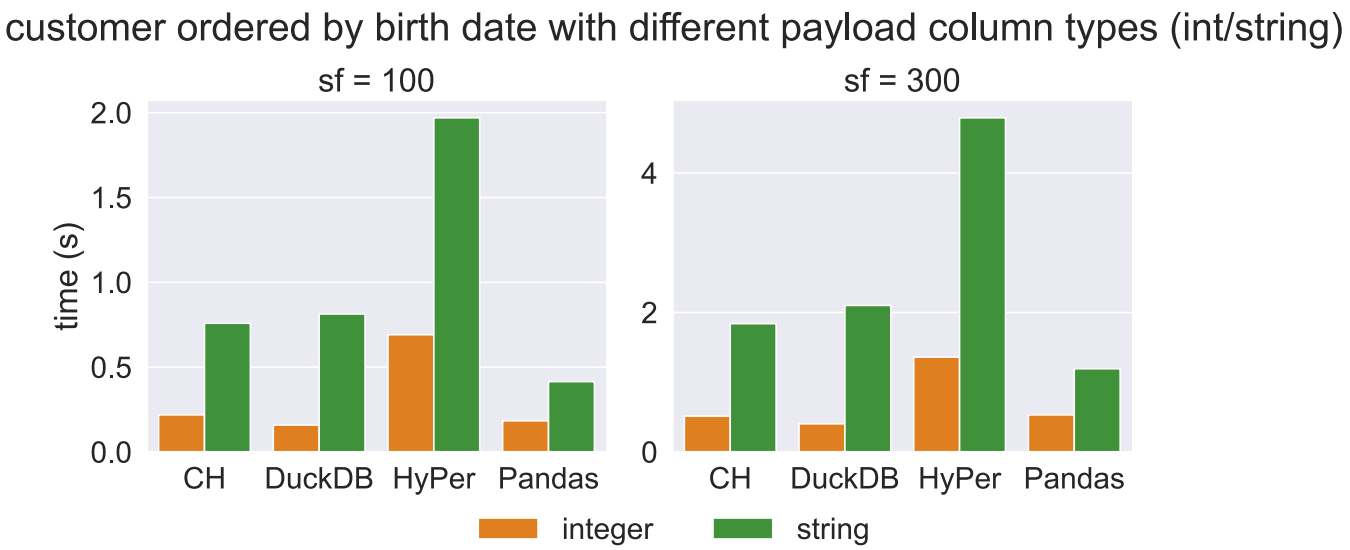
Customer (Strings & Integers)

Now that we have seen how the systems handle large amounts of fixed-size types, it is time to see some variable-size types! For our first experiment on the `customer` table, we will select all columns, and order them by either 3 integer columns (`c_birth_year`, `c_birth_month`, `c_birth_day`), or by 2 string columns (`c_first_name`, `c_last_name`). Comparing strings is much, much more difficult than comparing integers, because strings can have variable sizes, and need to be compare byte-by-byte, whereas integers always have the same comparison.



As expected, ordering by strings is more expensive than ordering by integers, except for HyPer, which is impressive. Pandas has only a slightly bigger difference between ordering by integers and ordering by strings than ClickHouse and DuckDB. This difference is explained by an expensive comparator between strings. Pandas uses NumPy's sort, which is efficiently implemented in **C**. However, when this sorts strings, it has to use virtual function calls to compare a Python string object, which is slower than a simple "<" between integers in **C**. Nevertheless, Pandas performs well on the `customer` table.

In our next experiment, we will see how the payload type affects performance. `customer` has 10 integer columns and 8 string columns. We will either select all integer columns or all string columns and order by (`c_birth_year`, `c_birth_month`, `c_birth_day`) every time.



As expected, re-ordering strings takes much more time than re-ordering integers. Pandas has an advantage here because it already has the strings in memory, and most likely only needs to re-order pointers to these strings. The database systems need to copy strings twice: Once when reading the input table, and again when creating the output table. Profiling in DuckDB reveals that the actual sorting takes less than a second at SF300, and most time is spent on (de)serializing strings.

Conclusion

DuckDB's new parallel sorting implementation can efficiently sort more data than fits in memory, making use of the speed of modern SSDs. Where other systems crash because they run out of memory, or switch to an external sorting strategy that is much slower, DuckDB's performance gracefully degrades as it goes over the memory limit.

The code that was used to run the experiments can be found on GitHub. If we made any mistakes, please let us know!

DuckDB is a free and open-source database management system (MIT licensed). It aims to be the SQLite for Analytics, and provides a fast and efficient database system with zero external dependencies. It is available not just for Python, but also for C/C++, R, Java, and more.

Discuss this post on Hacker News

Read our paper on sorting at ICDE '23

Listen to Laurens' appearance on the Disseminate podcast:

- Spotify
- Google
- Apple

Appendix A: Predication

Another technique we have used to speed up merge sort is *predication*. With this technique, we turn code with *if/else* branches into code without branches. Modern CPUs try to predict whether the *if*, or the *else* branch will be predicted. If this is hard to predict, it can slow down the code. Take a look at the example of pseudo-code with branches below.

```
// continue until merged
while (l_ptr && r_ptr) {
  // check which side is smaller
  if (memcmp(l_ptr, r_ptr, entry) < 0) {
    // copy from left side and advance
    memcpy(result_ptr, l_ptr, entry);
    l_ptr += entry;
```

```
  } else {
    // copy from right side and advance
    memcpy(result_ptr, r_ptr, entry);
    r_ptr += entry;
  }
  // advance result
  result_ptr += entry;
}
```

We are merging the data from the left and right blocks into a result block, one entry at a time, by advancing pointers. This code can be made *branchless* by using the comparison boolean as a 0 or 1, shown in the pseudo-code below.

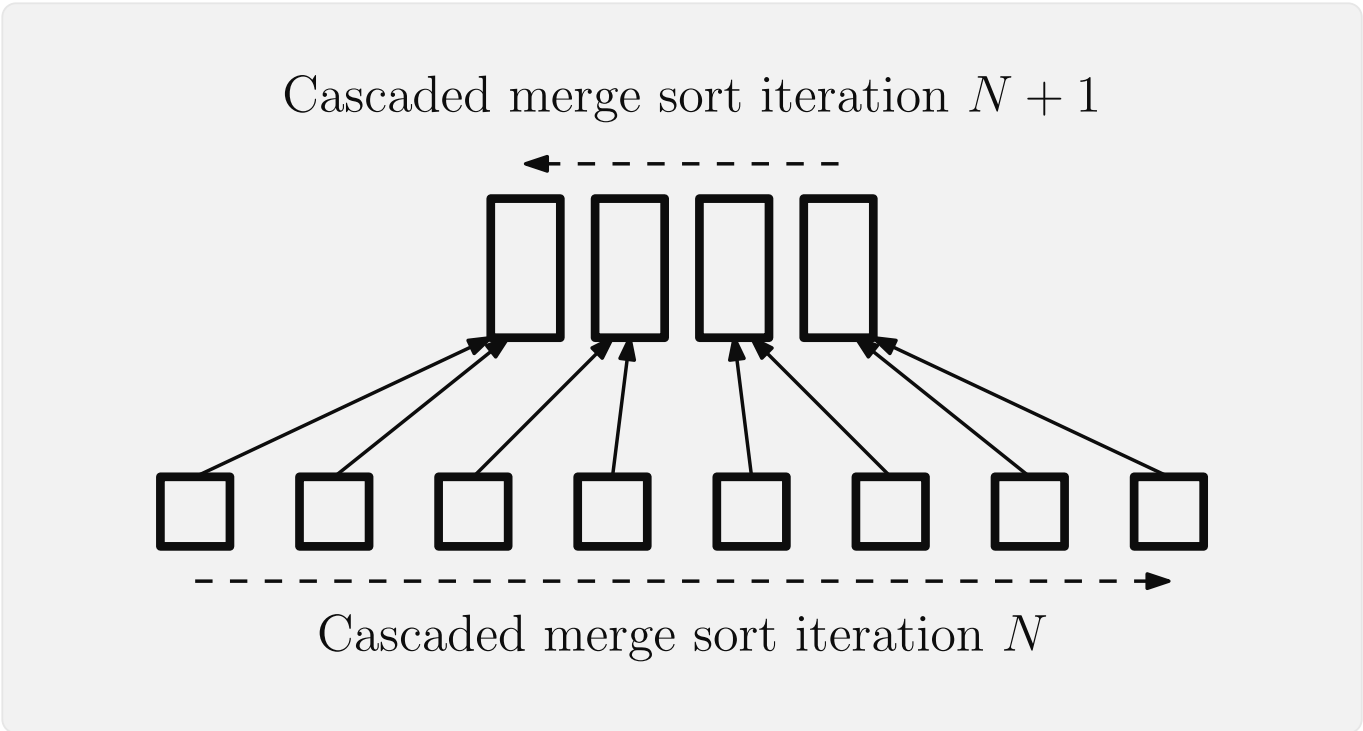
```
// continue until merged
while (l_ptr && r_ptr) {
  // store comparison result in a bool
  bool left_less = memcmp(l_ptr, r_ptr, entry) < 0;
  bool right_less = 1 - left_less;
  // copy from either side
  memcpy(result_ptr, l_ptr, left_less * entry);
  memcpy(result_ptr, r_ptr, right_less * entry);
  // advance either one
  l_ptr += left_less * entry;
  l_ptr += right_less * entry;
  // advance result
  result_ptr += entry;
}
```

When `left_less` is true, it is equal to 1. This means `right_less` is false, and therefore equal to 0. We use this to copy `entry` bytes from the left side, and 0 bytes from the right side, and incrementing the left and right pointers accordingly.

With predicated code, the CPU does not have to predict which instructions to execute, which means there will be fewer instruction cache misses!

Appendix B: Zig-Zagging

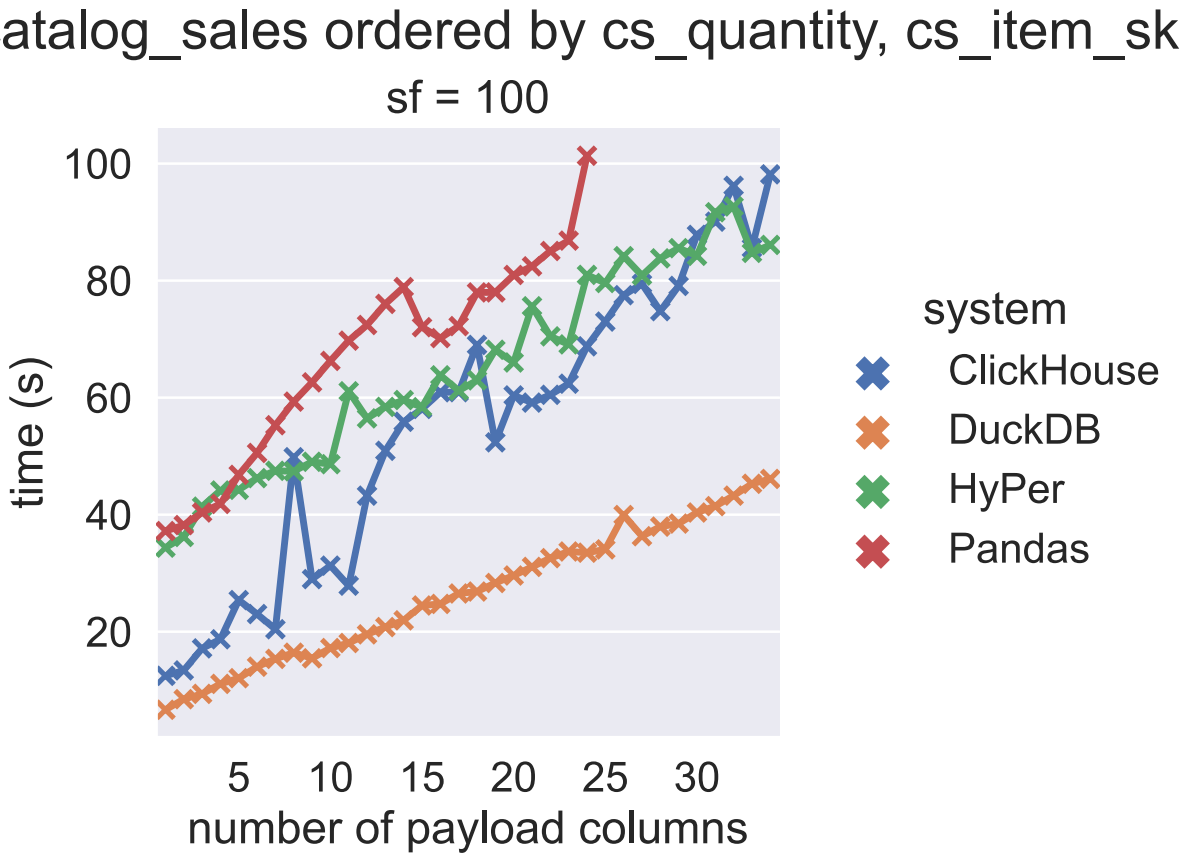
A simple trick to reduce I/O is zig-zagging through the pairs of blocks to merge in the cascaded merge sort. This is illustrated in the image below (dashes arrows indicate in which order the blocks are merged).



By zig-zagging through the blocks, we start an iteration by merging the last blocks that were merged in the previous iteration. Those blocks are likely still in memory, saving us some precious read/write operations.

Appendix C: x86 Experiment

We also ran the `catalog_sales` SF100 experiment on a machine with x86 CPU architecture, to get a more fair comparison with HyPer (without Rosetta 2 emulation). The machine has an Intel(R) Xeon(R) W-2145 CPU @ 3.70 GHz, which has 8 cores (up to 16 virtual threads), and 128 GB of RAM, so this time the data fits fully in memory. We have set the number of threads that DuckDB and ClickHouse use to 8 because we saw no visibly improved performance past 8.



Pandas performs comparatively worse than on the MacBook, because it has a single-threaded implementation, and this CPU has a lower single-thread performance. Again, Pandas crashes with an error (this machine does not dynamically increase swap):

```
numpy.core._exceptions.MemoryError: Unable to allocate 6.32 GiB for an array with shape (6, 141430723) and data type float64
```

DuckDB, HyPer, and ClickHouse all make good use out of more available threads, being significantly faster than on the MacBook.

An interesting pattern in this plot is that DuckDB and HyPer scale very similarly with additional payload columns. Although DuckDB is faster at sorting, re-ordering the payload seems to cost about the same for both systems. Therefore it is likely that HyPer also uses a row layout.

ClickHouse scales worse with additional payload columns. ClickHouse does not use a row layout, and therefore has to pay the cost of random access as each column is re-ordered after sorting.

Windowing in DuckDB

Publication date: 2021-10-13

Author: Richard Wesley

TL;DR: DuckDB, a free and open-source analytical data management system, has a state-of-the-art windowing engine that can compute complex moving aggregates like inter-quartile ranges as well as simpler moving averages.

Window functions (those using the OVER clause) are important tools for analyzing data series, but they can be slow if not implemented carefully. In this post, we will take a look at how DuckDB implements windowing. We will also see how DuckDB can leverage its aggregate function architecture to compute useful moving aggregates such as moving inter-quartile ranges (IQRs).

Beyond Sets

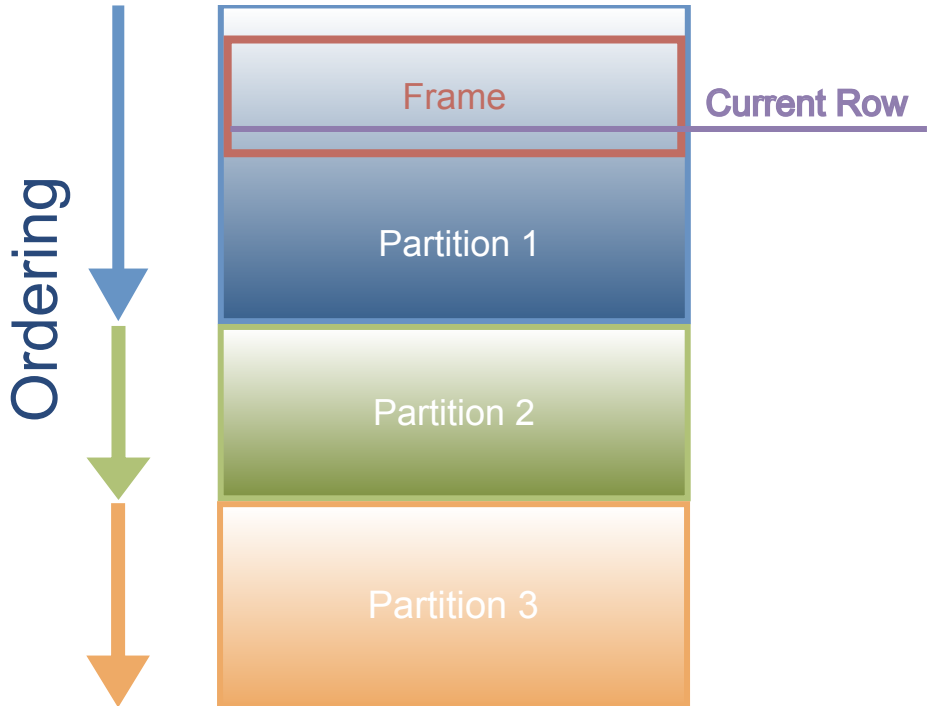
The original relational model as developed by Codd in the 1970s treated relations as *unordered sets* of tuples. While this was nice for theoretical computer science work, it ignored the way humans think using physical analogies (the "embodied brain" model from neuroscience). In particular, humans naturally order data to help them understand it and engage with it. To help with this, SQL uses the `SELECT` clause for horizontal layout and the `ORDER BY` clause for vertical layout.

Still, the orderings that humans put on data are often more than neurological crutches. For example, time places a natural ordering on measurements, and wide swings in those measurements can themselves be important data, or they may indicate that the data needs to be cleaned by smoothing. Trends may be present or relative changes may be more important for analysis than raw values. To help answer such questions, SQL introduced *analytic* (or *window*) functions in 2003.

Window Functions

Windowing works by breaking a relation up into independent *partitions*, *ordering* those partitions, and then defining various functions that can be computed for each row using the nearby values. These functions include all the aggregate functions (such as `sum` and `avg`) as well as some window-specific functions (such as `rank()` and `nth_value(<expression>, <N>)`).

Some window functions depend only on the partition boundary and the ordering, but a few (including all the aggregates) also use a *frame*. Frames are specified as a number of rows on either side (*preceding* or *following*) of the *current row*. The distance can either be specified as a number of *rows* or a *range* of values using the partition's ordering value and a distance.



Framing is the most confusing part of the windowing environment, so let's look at a very simple example and ignore the partitioning and ordering for a moment.

```
SELECT points,
    sum(points) OVER (
        ROWS BETWEEN 1 PRECEDING
                 AND 1 FOLLOWING) AS we
FROM results;
```

This query computes the sum of each point and the points on either side of it:

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

Notice that at the edge of the partition, there are only two values added together.

Power Generation Example

Now let's look at a concrete example of a window function query. Suppose we have some power plant generation data:

Plant	Date	MWh
Boston	2019-01-02	564337
Boston	2019-01-03	507405
Boston	2019-01-04	528523
Boston	2019-01-05	469538
Boston	2019-01-06	474163
Boston	2019-01-07	507213
Boston	2019-01-08	613040
Boston	2019-01-09	582588
Boston	2019-01-10	499506
Boston	2019-01-11	482014
Boston	2019-01-12	486134
Boston	2019-01-13	531518
Worcester	2019-01-02	118860
Worcester	2019-01-03	101977
Worcester	2019-01-04	106054
Worcester	2019-01-05	92182
Worcester	2019-01-06	94492
Worcester	2019-01-07	99932
Worcester	2019-01-08	118854
Worcester	2019-01-09	113506
Worcester	2019-01-10	96644
Worcester	2019-01-11	93806
Worcester	2019-01-12	98963
Worcester	2019-01-13	107170

The data is noisy, so we want to compute a 7 day moving average for each plant. To do this, we can use this window query:

```
SELECT "Plant", "Date",
    avg("MWh") OVER (
        PARTITION BY "Plant"
        ORDER BY "Date" ASC
        RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
                  AND INTERVAL 3 DAYS FOLLOWING)
        AS "MWh 7-day Moving Average"
FROM "Generation History"
ORDER BY 1, 2;
```

This query computes the seven day moving average of the power generated by each power plant on each day. The OVER clause is the way that SQL specifies that a function is to be computed in a window. It partitions the data by `Plant` (to keep the different power plants' data separate), orders each plant's partition by `Date` (to put the energy measurements next to each other), and uses a RANGE frame of three days on either side of each day for the `avg` (to handle any missing days). Here is the result:

Plant	Date	MWh 7-dayMoving Average
Boston	2019-01-02	517450.75

Plant	Date	MWh 7-dayMoving Average
Boston	2019-01-03	508793.20
Boston	2019-01-04	508529.83
Boston	2019-01-05	523459.85
Boston	2019-01-06	526067.14
Boston	2019-01-07	524938.71
Boston	2019-01-08	518294.57
Boston	2019-01-09	520665.42
Boston	2019-01-10	528859.00
Boston	2019-01-11	532466.66
Boston	2019-01-12	516352.00
Boston	2019-01-13	499793.00
Worcester	2019-01-02	104768.25
Worcester	2019-01-03	102713.00
Worcester	2019-01-04	102249.50
Worcester	2019-01-05	104621.57
Worcester	2019-01-06	103856.71
Worcester	2019-01-07	103094.85
Worcester	2019-01-08	101345.14
Worcester	2019-01-09	102313.85
Worcester	2019-01-10	104125.00
Worcester	2019-01-11	104823.83
Worcester	2019-01-12	102017.80
Worcester	2019-01-13	99145.75

You can request multiple different `OVER` clauses in the same `SELECT`, and each will be computed separately. Often, however, you want to use the same window for multiple functions, and you can do this by using a `WINDOW` clause to define a *named* window:

```
SELECT "Plant", "Date",
    avg("MWh") OVER seven AS "MWh 7-day Moving Average"
FROM "Generation History"
WINDOW seven AS (
    PARTITION BY "Plant"
    ORDER BY "Date" ASC
    RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
              AND INTERVAL 3 DAYS FOLLOWING)
ORDER BY 1, 2;
```

This would be useful, for example, if one also wanted the 7-day moving `min` and `max` to show the bounds of the data.

Under the Feathers

That is a long list of complicated functionality! Making it all work relatively quickly has many pieces, so lets have a look at how they all get implemented in DuckDB.

Pipeline Breaking

The first thing to notice is that windowing is a "pipeline breaker". That is, the `Window` operator has to read all of its inputs before it can start computing a function. This means that if there is some other way to compute something, it may well be faster to use a different technique.

One common analytic task is to find the last value in some group. For example, suppose we want the last recorded power output for each plant. It is tempting to use the `rank()` window function with a reverse sort for this task:

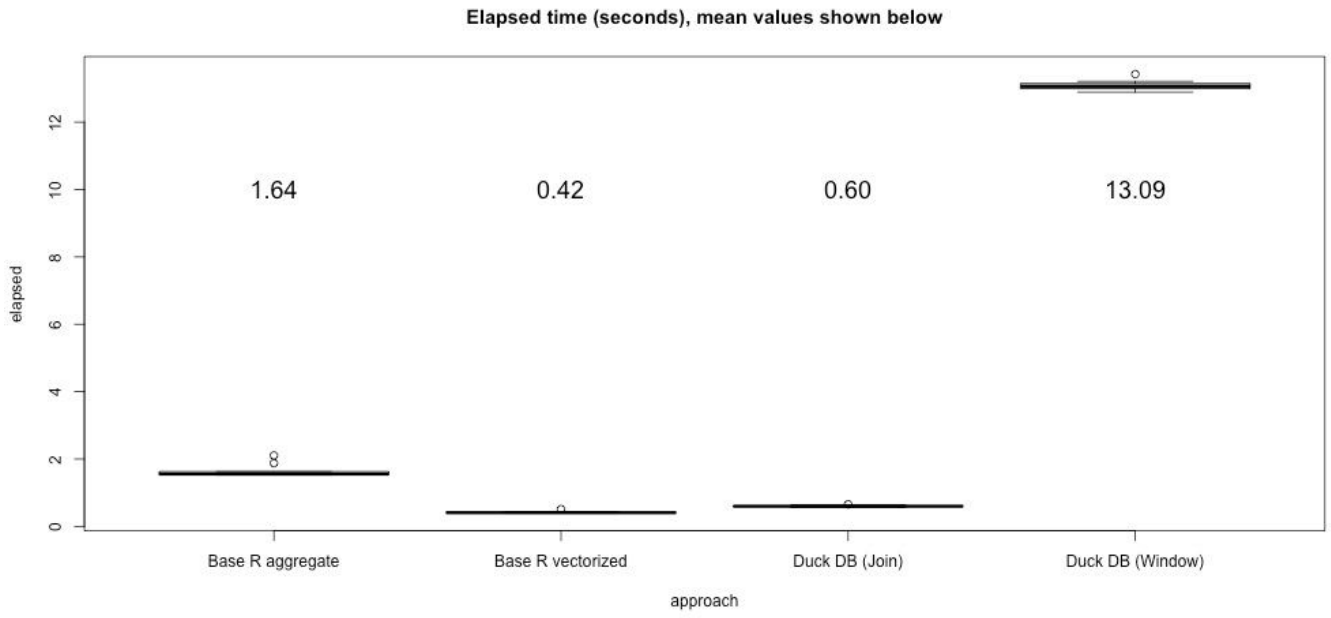
```
SELECT "Plant", "MWh"
FROM (
    SELECT "Plant", "MWh",
        rank() OVER (
            PARTITION BY "Plant"
            ORDER BY "Date" DESC) AS r
    FROM table) t
WHERE r = 1;
```

but this requires materialising the entire table, partitioning it, sorting the partitions, and then pulling out a single row from those partitions. A much faster way to do this is to use a self join to filter the table to contain only the last (`max`) value of the `DATE` field:

```
SELECT table."Plant", "MWh"
FROM table,
    (SELECT "Plant", max("Date") AS "Date"
     FROM table GROUP BY 1) lasts
WHERE table."Plant" = lasts."Plant"
  AND table."Date" = lasts."Date";
```

This join query requires two scans of the table, but the only materialised data is the filtering table (which is probably much smaller than the original table), and there is no sorting at all.

This type of query showed up in a user's blog and we found that the join query was over 20 times faster on their data set:



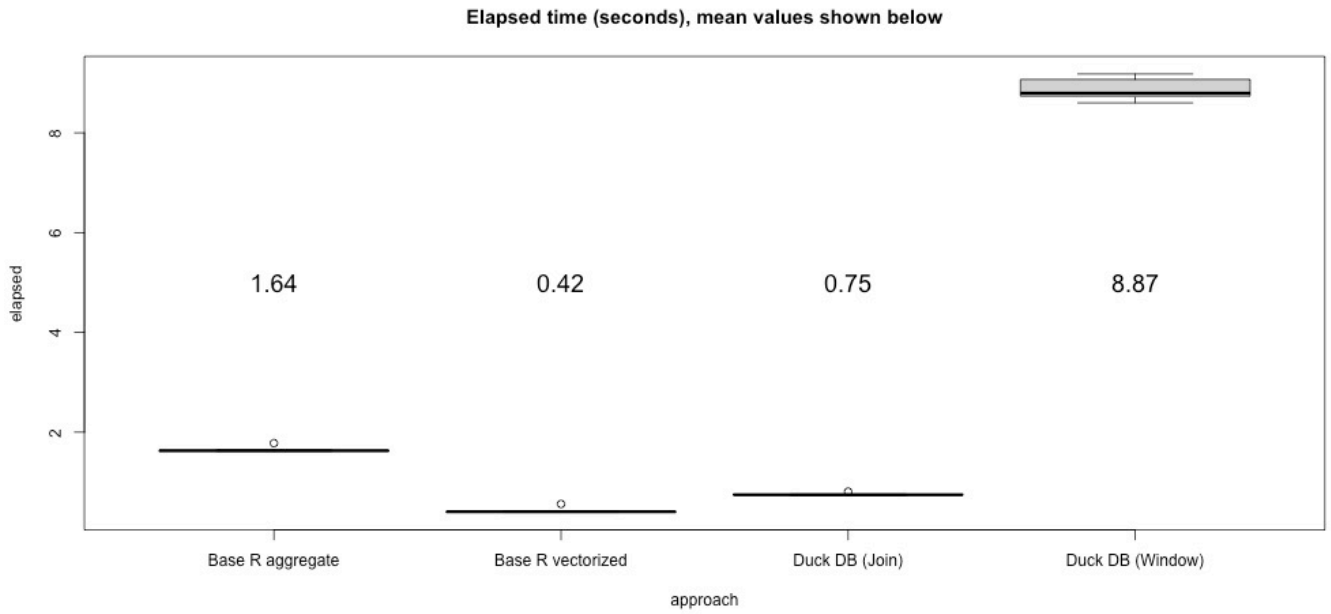
Of course most analytic tasks that use windowing *do* require using the `Window` operator, and DuckDB uses a collection of techniques to make the performance as fast as possible.

Partitioning and Sorting

At one time, windowing was implemented by sorting on both the partition and the ordering fields and then finding the partition boundaries. This is resource intensive, both because the entire relation must be sorted, and because sorting is `O(N log N)` in the size of the relation. Fortunately, there are faster ways to implement this step.

To reduce resource consumption, DuckDB uses the partitioning scheme from Leis et al.'s *Efficient Processing of Window Functions in Analytical SQL Queries* and breaks the partitions up into 1024 chunks using `O(N)` hashing. The chunks still need to be sorted on all the fields because there may be hash collisions, but each partition can now be 1024 times smaller, which reduces the runtime significantly. Moreover, the partitions can easily be extracted and processed in parallel.

Sorting in DuckDB recently got a big performance boost, along with the ability to work on partitions that were larger than memory. This functionality has been also added to the `Window` operator, resulting in a 33% improvement in the last-in-group example:



As a final optimization, even though you can request multiple window functions, DuckDB will collect functions that use the same partitioning and ordering, and share the data layout between those functions.

Aggregation

Most of the general-purpose window functions are straightforward to compute, but windowed aggregate functions can be expensive because they need to look at multiple values for each row. They often need to look at the same value multiple times, or repeatedly look at a large number of values, so over the years several approaches have been taken to improve performance.

Naïve Windowed Aggregation Before explaining how DuckDB implements windowed aggregation, we need to take a short detour through how ordinary aggregates are implemented. Aggregate "functions" are implemented using three required operations and one optional operation:

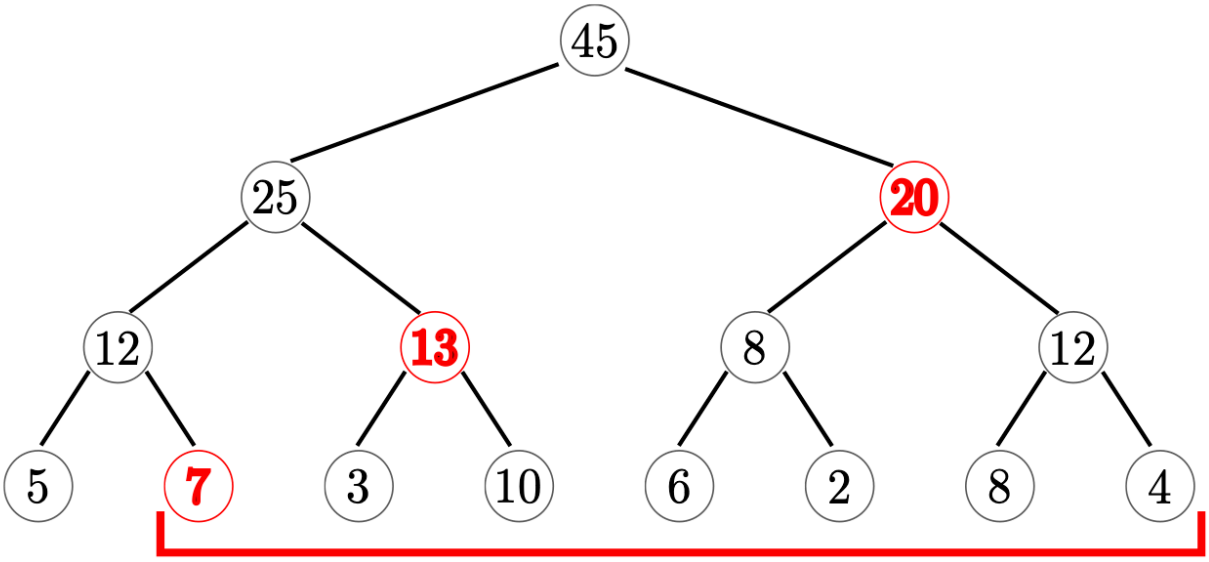
- *Initialize* – Creates a state that will be updated. For `sum`, this is the running total, starting at `NULL` (because a `sum` of zero items is `NULL`, not zero.)
- *Update* – Updates the state with a new value. For `sum`, this adds the value to the state.
- *Finalize* – Produces the final aggregate value from the state. For `sum`, this just copies the running total.
- *Combine* – Combines two states into a single state. Combine is optional, but when present it allows the aggregate to be computed in parallel. For `sum`, this produces a new state with the sum of the two input values.

The simplest way to compute an individual windowed aggregate value is to *initialize* a state, *update* the state with all the values in the window frame, and then use *finalize* to produce the value of the windowed aggregate. This naïve algorithm will always work, but it is quite inefficient. For example, a running total will re-add all the values from the start of the partition for each running total, and this has a run time of `O(N^2)`.

To improve on this, some databases add additional "moving state" operations that can add or remove individual values incrementally. This reduces computation in some common cases, but it can only be used for certain aggregates. For example, it doesn't work for `min`) because

you don't know if there are multiple duplicate minima. Moreover, if the frame boundaries move around a lot, it can still degenerate to an `O(N^2)` run time.

Segment Tree Aggregation Instead of adding more functions, DuckDB uses the *segment tree* approach from Leis et al. above. This works by building a tree on top of the entire partition with the aggregated values at the bottom. Values are combined into states at nodes above them in the tree until there is a single root:



To compute a value, the algorithm generates states for the ragged ends of the frame, *combines* states in the tree above the values in the frame, and *finalizes* the result from the last remaining state. So in the example above (Figure 5 from Leis et al.) only three values need to be added instead of 7. This technique can be used for all *combinable* aggregates.

General Windowed Aggregation The biggest drawback of segment trees is the need to manage a potentially large number of intermediate states. For the simple states used for standard distributive aggregates like `sum`, this is not a problem because the states are small, the tree keeps the number of states logarithmically low, and the state used to compute each value is also cheap.

For some aggregates, however, the state is not small. Typically these are so-called *holistic* aggregates, where the value depends on all the values of the frame. Examples of such aggregates are `mode` and `quantile`, where each state may have to contain a copy of *all* the values seen so far. While segment trees *can* be used to implement moving versions of any combinable aggregate, this can be quite expensive for large, complex states – and this was not the original goal of the algorithm.

To solve this problem, we use the approach from Wesley and Xu's *Incremental Computation of Common Windowed Holistic Aggregates*, which generalises segment trees to aggregate-specific data structures. The aggregate can define a fifth optional *window* operation, which will be passed the bottom of the tree and the bounds of the current and previous frame. The aggregate can then create an appropriate data structure for its implementation.

For example, the `mode` function maintains a hash table of counts that it can update efficiently, and the `quantile` function maintains a partially sorted list of frame indexes. Moreover, the `quantile` functions can take an array of quantile values, which further increases performance by sharing the partially ordered results among the different quantile values.

Because these aggregates can be used in a windowing context, the moving average example above can be easily modified to produce a moving inter-quartile range:

```
SELECT "Plant", "Date",
    quantile_cont("MWh", [0.25, 0.5, 0.75]) OVER seven
        AS "MWh 7-day Moving IQR"
FROM "Generation History"
```

```
WINDOW seven AS (
    PARTITION BY "Plant"
    ORDER BY "Date" ASC
    RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
              AND INTERVAL 3 DAYS FOLLOWING)
ORDER BY 1, 2;
```

Moving quantiles like this are more robust to anomalies, which makes them a valuable tool for data series analysis, but they are not generally implemented in most database systems. There are some approaches that can be used in some query engines, but the lack of a general moving aggregation architecture means that these solutions can be unnatural or complex. DuckDB's implementation uses the standard window notation, which means you don't have to learn new syntax or pull the data out into another tool.

Ordered Set Aggregates Window functions are often closely associated with some special "ordered set aggregates" defined by the SQL standard. Some databases implement these functions using the `Window` operator, but this is rather inefficient because sorting the data (an `O(N log N)` operation) is not required – it suffices to use Hoare's `O(N)` `FIND` algorithm as used in the STL's `std::nth_element`. DuckDB translates these ordered set aggregates to use the faster `quantile_cont`, `quantile_disc`, and `mode` regular aggregate functions, thereby avoiding using windowing entirely.

Extensions This architecture also means that any new aggregates we add can benefit from the existing windowing infrastructure. DuckDB is an open source project, and we welcome submissions of useful aggregate functions – or you can create your own domain-specific ones in your own fork. At some point we hope to have a UDF architecture that will allow plug-in aggregates, and the simplicity and power of the interface will let these plugins leverage the notational simplicity and run time performance that the internal functions enjoy.

Conclusion

DuckDB's windowing implementation uses a variety of techniques to speed up what can be the slowest part of an analytic query. It is well integrated with the sorting subsystem and the aggregate function architecture, which makes expressing advanced moving aggregates both natural and efficient.

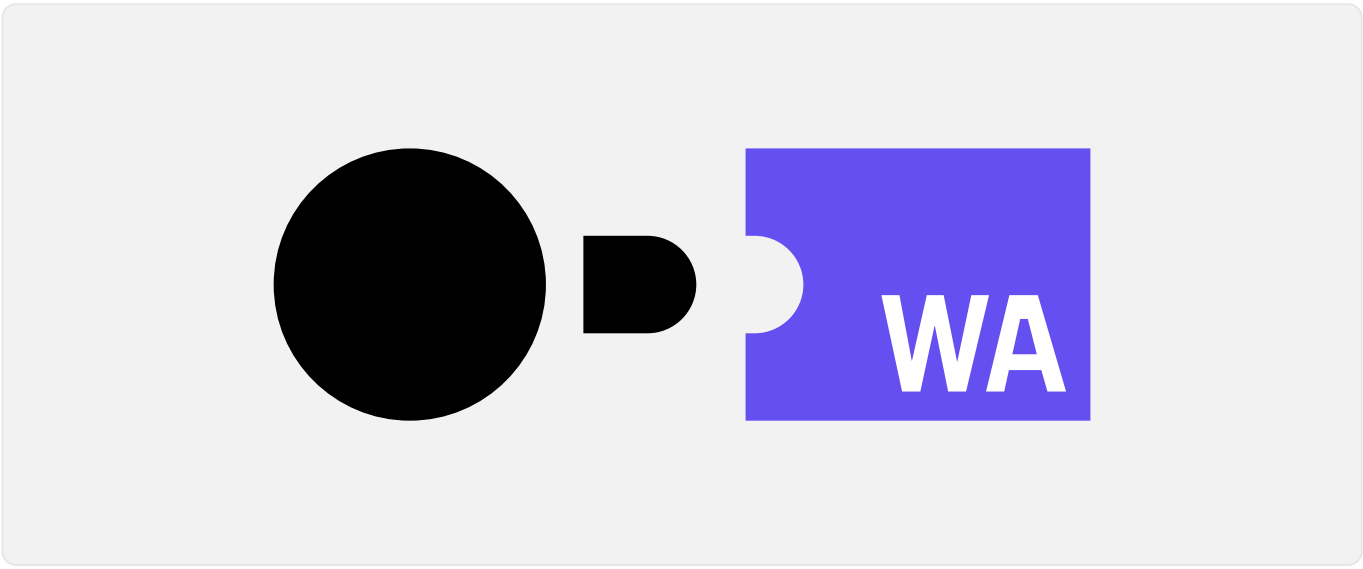
DuckDB is a free and open-source database management system (MIT licensed). It aims to be the SQLite for Analytics, and provides a fast and efficient database system with zero external dependencies. It is available not just for Python, but also for C/C++, R, Java, and more.

DuckDB-Wasm: Efficient Analytical SQL in the Browser

Publication date: 2021-10-29

Authors: André Kohn, Dominik Moritz

TL;DR: DuckDB-Wasm is an in-process analytical SQL database for the browser. It is powered by WebAssembly, speaks Arrow fluently, reads Parquet, CSV and JSON files backed by Filesystem APIs or HTTP requests and has been tested with Chrome, Firefox, Safari and Node.js. You can try it at shell.duckdb.org or on Observable.



DuckDB-Wasm is fast! If you're here for performance numbers, head over to our benchmarks at shell.duckdb.org/versus.

Efficient Analytics in the Browser

The web browser has evolved to a universal computation platform that even runs in your car. Its rise has been accompanied by increasing requirements for the browser programming language JavaScript. JavaScript was, first and foremost, designed to be very flexible which comes at the cost of a reduced processing efficiency compared to native languages like C++. This becomes particularly apparent when considering the execution times of more complex data analysis tasks that often fall behind the native execution by orders of magnitude. In the past, such analysis tasks have therefore been pushed to servers that tie any client-side processing to additional round-trips over the internet and introduce their own set of scalability problems.

The processing capabilities of browsers were boosted tremendously 4 years ago with the introduction of WebAssembly:

> WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.
>
> The Wasm stack machine is designed to be encoded in a size- and load-time efficient binary format. WebAssembly aims to execute at native speed by taking advantage of common hardware capabilities available on a wide range of platforms.
>
> (ref: https://webassembly.org/)

Four years later, the WebAssembly revolution is in full progress with first implementations being shipped in four major browsers. It has already brought us game engines, entire IDEs and even a browser version of Photoshop. Today, we join the ranks with a first release of the npm library @duckdb/duckdb-wasm.

As an in-process analytical database, DuckDB has the rare opportunity to significantly speed up OLAP workloads in the browser. We believe that there is a need for a comprehensive and self-contained data analysis library. DuckDB-wasm automatically offloads your queries to dedicated worker threads and reads Parquet, CSV and JSON files from either your local filesystem or HTTP servers driven by plain SQL input. In this blog post, we want to introduce the library and present challenges on our journey towards a browser-native OLAP database.

DuckDB-Wasm is not yet stable. You will find rough edges and bugs in this release. Please share your thoughts with us on GitHub.

How to Get Data In?

Let's dive into examples. DuckDB-Wasm provides a variety of ways to load your data. First, raw SQL value clauses like `INSERT INTO sometable VALUES (1, 'foo'), (2, 'bar')` are easy to formulate and only depend on plain SQL text. Alternatively, SQL statements like `CREATE TABLE foo AS SELECT * FROM 'somefile.parquet'` consult our integrated web filesystem to resolve `somefile.parquet` locally, remotely or from a buffer. The methods `insertCSVFromPath` and `insertJSONFromPath` further provide convenient ways to import CSV and JSON files using additional typed settings like column types. And finally, the method `insertArrowFromIPCStream` (optionally through `insertArrowTable`, `insertArrowBatches` or `insertArrowVectors`) copies raw IPC stream bytes directly into a WebAssembly stream decoder.

The following example presents different options how data can be imported into DuckDB-Wasm:

```
// Data can be inserted from an existing arrow.Table
await c.insertArrowTable(existingTable, { name: "arrow_table" });
// ..., from Arrow vectors
await c.insertArrowVectors({
    col1: arrow.Int32Vector.from([1, 2]),
    col2: arrow.Utf8Vector.from(["foo", "bar"]),
}, {
    name: "arrow_vectors"
});
// ..., from a raw Arrow IPC stream
const c = await db.connect();
const streamResponse = await fetch(` someapi`);
const streamReader = streamResponse.body.getReader();
const streamInserts = [];
while (true) {
    const { value, done } = await streamReader.read();
    if (done) break;
    streamInserts.push(c.insertArrowFromIPCStream(value, { name: "streamed" }));
}
await Promise.all(streamInserts);

// ..., from CSV files
// (interchangeable: registerFile{Text,Buffer,URL,Handle})
await db.registerFileText(` data.csv`, "1|foo\n2|bar\n");
// ... with typed insert options
await db.importCSVFromPath('data.csv', {
    schema: 'main',
    name: 'foo',
    detect: false,
    header: false,
    delimiter: '|',
    columns: {
        col1: new arrow.Int32(),
        col2: new arrow.Utf8(),
    }
});

// ..., from JSON documents in row-major format
await db.registerFileText("rows.json", `[
    { "col1": 1, "col2": "foo" },
    { "col1": 2, "col2": "bar" },
]`);
// ... or column-major format
```

```
await db.registerFileText("columns.json", `{
    "col1": [1, 2],
    "col2": ["foo", "bar"]
}`);
// ... with typed insert options
await db.importJSONFromPath('rows.json', { name: 'rows' });
await db.importJSONFromPath('columns.json', { name: 'columns' });

// ..., from Parquet files
const pickedFile: File = letUserPickFile();
await db.registerFileHandle("local.parquet", pickedFile);
await db.registerFileURL("remote.parquet", "https://origin/remote.parquet");

// ..., by specifying URLs in the SQL text
await c.query(`
    CREATE TABLE direct AS
        SELECT * FROM 'https://origin/remote.parquet'
`);
// ..., or by executing raw insert statements
await c.query(` INSERT INTO existing_table
    VALUES (1, "foo"), (2, "bar")`);
```

How to Get Data Out?

Now that we have the data loaded, DuckDB-Wasm can run queries on two different ways that differ in the result materialization. First, the method `query` runs a query to completion and returns the results as single `arrow.Table`. Second, the method `send` fetches query results lazily through an `arrow.RecordBatchStreamReader`. Both methods are generic and allow for typed results in Typescript:

```
// Either materialize the query result
await conn.query<{ v: arrow.Int32 }>(`
    SELECT * FROM generate_series(1, 100) t(v)
`);
// ..., or fetch the result chunks lazily
for await (const batch of await conn.send<{ v: arrow.Int32 }>(`
    SELECT * FROM generate_series(1, 100) t(v)
`)) {
    // ...
}
```

Alternatively, you can prepare statements for parameterized queries using:

```
// Prepare query
const stmt = await conn.prepare<{ v: arrow.Int32 }>(
    `SELECT (v + ?) AS v FROM generate_series(0, 10000) t(v);`
);
// ... and run the query with materialized results
await stmt.query(234);
// ... or result chunks
for await (const batch of await stmt.send(234)) {
    // ...
}
```

Looks like Arrow to Me

DuckDB-Wasm uses Arrow as data protocol for the data import and all query results. Arrow is a database-friendly columnar format that is organized in chunks of column vectors, called record batches and that support zero-copy reads with only a small overhead. The npm library `apache-arrow` implements the Arrow format in the browser and is already used by other data processing frameworks, like Arquero. Arrow therefore not only spares us the implementation of the SQL type logic in JavaScript, it also makes us compatible to existing tools.

Why not use plain Javascript objects?

WebAssembly is isolated and memory-safe. This isolation is part of it's DNA and drives fundamental design decisions in DuckDB-Wasm. For example, WebAssembly introduces a barrier towards the traditional JavaScript heap. Crossing this barrier is difficult as JavaScript has to deal with native function calls, memory ownership and serialization performance. Languages like C++ make this worse as they rely on smart pointers that are not available through the FFI. They leave us with the choice to either pass memory ownership to static singletons within the WebAssembly instance or maintain the memory through C-style APIs in JavaScript, a language that is too dynamic for sound implementations of the RAII idiom. The memory-isolation forces us to serialize data before we can pass it to the WebAssembly instance. Browsers can serialize JavaScript objects natively to and from JSON using the functions `JSON.stringify` and `JSON.parse` but this is slower compared to, for example, copying raw native arrays.

Web Filesystem

DuckDB-Wasm integrates a dedicated filesystem for WebAssembly. DuckDB itself is built on top of a virtual filesystem that decouples higher level tasks, such as reading a Parquet file, from low-level filesystem APIs that are specific to the operating system. We leverage this abstraction in DuckDB-Wasm to tailor filesystem implementations to the different WebAssembly environments.

The following figure shows our current web filesystem in action. The sequence diagram presents a user running a SQL query that scans a single Parquet file. The query is first offloaded to a dedicated web worker through a JavaScript API. There, it is passed to the WebAssembly module that processes the query until the execution hits the `parquet_scan` table function. This table function then reads the file using a buffered filesystem which, in turn, issues paged reads on the web filesystem. This web filesystem then uses an environment-specific runtime to read the file from several possible locations.

Depending on the context, the Parquet file may either reside on the local device, on a remote server or in a buffer that was registered by the user upfront. We deliberately treat all three cases equally to unify the retrieval and processing of external data. This does not only simplify the analysis, it also enables more advanced features like partially consuming structured file formats. Parquet files, for example, consist of multiple row groups that store data in a column-major fashion. As a result, we may not need to download the entire file for a query but only required bytes.

A query like `SELECT count(*) FROM parquet_scan(...)`, for example, can be evaluated on the file metadata alone and will finish in milliseconds even on remote files that are several terabytes large. Another more general example are paging scans with `LIMIT` and `OFFSET` qualifiers such as `SELECT * FROM parquet_scan(...) LIMIT 20 OFFSET 40`, or queries with selective filter predicates where entire row groups can be skipped based on metadata statistics. These partial file reads are no groundbreaking novelty and could be implemented in JavaScript today, but with DuckDB-Wasm, these optimizations are now driven by the semantics of SQL queries instead of fine-tuned application logic.

Note: The common denominator among the available File APIs is unfortunately not large. This limits the features that we can provide in the browser. For example, local persistency of DuckDB databases would be a feature with significant impact but requires a way to either read and write synchronously into user-provided files or IndexedDB. We might be able to bypass these limitations in the future but this is subject of ongoing research.

Advanced Features

WebAssembly 1.0 has landed in all major browsers. The WebAssembly Community Group fixed the design of this first version back in November 2017, which is now referred to as WebAssembly MVP. Since then, the development has been ongoing with eight additional features that have been added to the standard and at least five proposals that are currently in progress.

The rapid pace of this development presents challenges and opportunities for library authors. On the one hand, the different features find their way into the browsers at different speeds which leads to a fractured space of post-MVP functionality. On the other hand, features can bring flat performance improvements and are therefore indispensable when aiming for a maximum performance.

The most promising feature for DuckDB-Wasm is exception handling which is already enabled by default in Chrome 95. DuckDB and DuckDB-Wasm are written in C++ and use exceptions for faulty situations. DuckDB does not use exceptions for general control flow but to automatically propagate errors upwards to the top-level plan driver. In native environments, these exceptions are implemented as “zero-cost exceptions” as they induce no overhead until they are thrown. With the WebAssembly MVP, however, that is no longer possible as the compiler toolchain Emscripten has to emulate exceptions through JavaScript. Without WebAssembly exceptions, DuckDB-Wasm calls throwing functions through a JavaScript hook that can catch exceptions emulated through JavaScript `aborts`. An example for these hook calls is

shown in the following figure. Both stack traces originate from a single paged read of a Parquet file in DuckDB-Wasm. The left side shows a stack trace with the WebAssembly MVP and requires multiple calls through the functions `wasm-to-js-i*`. The right stack trace uses WebAssembly exceptions without any hook calls.

This fractured feature space is a temporary challenge that will be resolved once high-impact features like exception handling, SIMD and bulk-memory operations are available everywhere. In the meantime, we will ship multiple WebAssembly modules that are compiled for different feature sets and adaptively pick the best bundle for you using dynamic browser checks.

The following example shows how the asynchronous version of DuckDB-Wasm can be instantiated using either manual or JsDelivr bundles:

```
// Import the ESM bundle (supports tree-shaking)
import * as duckdb from '@duckdb/duckdb-wasm/dist/duckdb-esm.js';

// Either bundle them manually, for example as Webpack assets
import duckdb_wasm from '@duckdb/duckdb-wasm/dist/duckdb.wasm';
import duckdb_wasm_next from '@duckdb/duckdb-wasm/dist/duckdb-next.wasm';
import duckdb_wasm_next_coi from '@duckdb/duckdb-wasm/dist/duckdb-next-coi.wasm';
const WEBPACK_BUNDLES: duckdb.DuckDBBundles = {
    asyncDefault: {
        mainModule: duckdb_wasm,
        mainWorker: new URL('@duckdb/duckdb-wasm/dist/duckdb-browser-async.worker.js',
import.meta.url).toString(),
    },
    asyncNext: {
        mainModule: duckdb_wasm_next,
        mainWorker: new URL('@duckdb/duckdb-wasm/dist/duckdb-browser-async-next.worker.js',
import.meta.url).toString(),
    },
    asyncNextCOI: {
        mainModule: duckdb_wasm_next_coi,
        mainWorker: new URL(
            '@duckdb/duckdb-wasm/dist/duckdb-browser-async-next-coi.worker.js',
            import.meta.url,
        ).toString(),
        pthreadWorker: new URL(
            '@duckdb/duckdb-wasm/dist/duckdb-browser-async-next-coi.pthread.worker.js',
            import.meta.url,
        ).toString(),
    },
};
// ..., or load the bundles from jsdelivr
const JSDELIVR_BUNDLES = duckdb.getJsDelivrBundles();

// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(JSDELIVR_BUNDLES);
// Instantiate the asynchronous version of DuckDB-Wasm
const worker = new Worker(bundle.mainWorker!);
const logger = new duckdb.ConsoleLogger();
const db = new duckdb.AsyncDuckDB(logger, worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
```

You can also test the features and selected bundle in your browser using the web shell command `.features`.

Multithreading

In 2018, the Spectre and Meltdown vulnerabilities sent crippling shockwaves through the internet. Today, we are facing the repercussions of these events, in particular in software that runs arbitrary user code – such as web browsers. Shortly after the publications, all major browser vendors restricted the use of `SharedArrayBuffers` to prevent dangerous timing attacks. `SharedArrayBuffers` are raw buffers that can be shared among web workers for global state and an alternative to the browser-specific message passing. These restrictions had detrimental effects on WebAssembly modules since `SharedArrayBuffers` are necessary for the implementation of POSIX threads in WebAssembly.

Without `SharedArrayBuffers`, WebAssembly modules can run in a dedicated web worker to unblock the main event loop but won't be able to spawn additional workers for parallel computations within the same instance. By default, we therefore cannot unleash the parallel query execution of DuckDB in the web. However, browser vendors have recently started to reenable `SharedArrayBuffers` for websites that are cross-origin-isolated. A website is cross-origin-isolated if it ships the main document with the following HTTP headers:

```
Cross-Origin-Embedder-Policy: require-corp
Cross-Origin-Opener-Policy: same-origin
```

These headers will instruct browsers to A) isolate the top-level document from other top-level documents outside its own origin and B) prevent the document from making arbitrary cross-origin requests unless the requested resource explicitly opts in. Both restrictions have far reaching implications for a website since many third-party data sources won't yet provide the headers today and the top-level isolation currently hinders the communication with, for example, OAuth pop up's (there are plans to lift that).

We therefore assume that DuckDB-Wasm will find the majority of users on non-isolated websites. We are, however, experimenting with dedicated bundles for isolated sites using the suffix `-next-coi`) and will closely monitor the future need of our users.

Web Shell

We further host a web shell powered by DuckDB-Wasm alongside the library release at shell.duckdb.org. Use the following shell commands to query remote TPC-H files at scale factor 0.01. When querying your own, make sure to properly set CORS headers since your browser will otherwise block these requests. You can alternatively use the `.files` command to register files from the local filesystem.

```
.timer on

SELECT count(*)
FROM 'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/lineitem.parquet';

SELECT count(*)
FROM 'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/customer.parquet';

SELECT avg(c_acctbal)
FROM 'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/customer.parquet';

SELECT *
FROM 'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/orders.parquet'
LIMIT 10;

SELECT n_name, avg(c_acctbal)
FROM
    'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/customer.parquet',
    'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/nation.parquet'
WHERE c_nationkey = n_nationkey
GROUP BY n_name;

SELECT *
FROM
    'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/region.parquet',
    'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/nation.parquet'
WHERE r_regionkey = n_regionkey;
```

Evaluation

The following table teases the execution times of some TPC-H queries at scale factor 0.5 using the libraries DuckDB-Wasm, sql.js, Arquero and Lovefield. You can find a more in-depth discussion with all TPC-H queries, additional scale factors and microbenchmarks on the "DuckDB-Wasm versus X" page.

Query	DuckDB-Wasm	sql.js	Arquero	Lovefield
1	**0.855 s**	8.441 s	24.031 s	12.666 s
3	**0.179 s**	1.758 s	16.848 s	3.587 s
4	**0.151 s**	0.384 s	6.519 s	3.779 s
5	**0.197 s**	1.965 s	18.286 s	13.117 s
6	**0.086 s**	1.294 s	1.379 s	5.253 s
7	**0.319 s**	2.677 s	6.013 s	74.926 s
8	**0.236 s**	4.126 s	2.589 s	18.983 s
10	**0.351 s**	1.238 s	23.096 s	18.229 s
12	**0.276 s**	1.080 s	11.932 s	10.372 s
13	**0.194 s**	5.887 s	16.387 s	9.795 s
14	**0.086 s**	1.194 s	6.332 s	6.449 s
16	**0.137 s**	0.453 s	0.294 s	5.590 s
19	**0.377 s**	1.272 s	65.403 s	9.977 s

Future Research

We believe that WebAssembly unveils hitherto dormant potential for shared query processing between clients and servers. Pushing computation closer to the client can eliminate costly round-trips to the server and thus increase interactivity and scalability of in-browser analytics. We further believe that the release of DuckDB-Wasm could be the first step towards a more universal data plane spanning across multiple layers including traditional database servers, clients, CDN workers and computational storage. As an in-process analytical database, DuckDB might be the ideal driver for distributed query plans that increase the scalability and interactivity of SQL databases at low costs.

Fast Moving Holistic Aggregates

Publication date: 2021-11-12

Author: Richard Wesley

TL;DR: DuckDB, a free and open-source analytical data management system, has a windowing API that can compute complex moving aggregates like interquartile ranges and median absolute deviation much faster than the conventional approaches.

In a previous post, we described the DuckDB windowing architecture and mentioned the support for some advanced moving aggregates. In this post, we will compare the performance various possible moving implementations of these functions and explain how DuckDB's performant implementations work.

What Is an Aggregate Function?

When people think of aggregate functions, they typically have something simple in mind such as `SUM` or `AVG`. But more generally, what an aggregate function does is *summarise* a set of values into a single value. Such summaries can be arbitrarily complex, and involve any data type. For example, DuckDB provides aggregates for concatenating strings (`STRING_AGG`) and constructing lists (`LIST`). In SQL, aggregated sets come from either a `GROUP BY` clause or an `OVER` windowing specification.

Holistic Aggregates

All of the basic SQL aggregate functions like `SUM` and `MAX` can be computed by reading values one at a time and throwing them away. But there are some functions that potentially need to keep track of all the values before they can produce a result. These are called *holistic* aggregates, and they require more care when implementing.

For some aggregates (like `STRING_AGG`) the order of the values can change the result. This is not a problem for windowing because `OVER` clauses can specify an ordering, but in a `GROUP BY` clause, the values are unordered. To handle this, order-sensitive aggregates can include a `WITHIN GROUP(ORDER BY <expr>)` clause to specify the order of the values. Because the values must all be collected and sorted, aggregates that use the `WITHIN GROUP` clause are holistic.

Statistical Holistic Aggregates

Because sorting the arguments to a windowed aggregate can be specified with the `OVER` clause, you might wonder if there are any other kinds of holistic aggregates that do not use sorting, or which use an ordering different from the one in the `OVER` clause. It turns out that there are a number of important statistical functions that turn into holistic aggregates in SQL. In particular, here are the statistical holistic aggregates that DuckDB currently supports:

Function	Description
`mode(x)`	The most common value in a set
`median(x)`	The middle value of a set
`quantile_disc(x, <frac>)`	The exact value corresponding to a fractional position.
`quantile_cont(x, <frac>)`	The interpolated value corresponding to a fractional position.
`quantile_disc(x, [<frac>...])`	A list of the exact values corresponding to a list of fractional positions.

Function	Description
`quantile_cont(x, [<frac>...])`	A list of the interpolated value corresponding to a list of fractional positions.
`mad(x)`	The median of the absolute values of the differences of each value from the median.

Where things get really interesting is when we try to compute moving versions of these aggregates. For example, computing a moving AVG is fairly straightforward: You can subtract values that have left the frame and add in the new ones, or use the segment tree approach from the previous post on windowing.

Python Example

Computing a moving median is not as easy. Let's look at a simple example of how we might implement moving `median` in Python for the following string data, using a frame that includes one element from each side:

Row	Data						Median
0	a	b	c	d	c	b	a
1	a	b	c	d	c	b	b
2	a	b	c	d	c	b	c
3	a	b	c	d	c	b	c
4	a	b	c	d	c	b	c
5	a	b	c	d	c	b	b

For this example we are using strings so we don't have to worry about interpolating values.

```
data = ('a', 'b', 'c', 'd', 'c', 'b',)
w = len(data)
for row in range(w):
    l = max(row - 1, 0)       # First index of the frame
    r = min(row + 1, w-1)     # Last index of the frame
    frame = list(data[l:r+1]) # Copy the frame values
    frame.sort()              # Sort the frame values
    n = (r - l) // 2          # Middle index of the frame
    median = frame[n]         # The median is the middle value
    print(row, data[row], median)
```

Each frame has a different set of values to aggregate and we can't change the order in the table, so we have to copy them each time before we sort. Sorting is slow, and there is a lot of repetition.

All of these holistic aggregates have similar problems if we just reuse the simple implementations for moving versions. Fortunately, there are much faster approaches for all of them.

Moving Holistic Aggregation

In the previous post on windowing, we explained the component operations used to implement a generic aggregate function (initialize, update, finalize, combine and window). In the rest of this post, we will dig into how they can be implemented for these complex aggregates.

Quantile

The `quantile` aggregate variants all extract the value(s) at a given fraction (or fractions) of the way through the ordered list of values in the set. The simplest variant is the `median` function, which we met in the introduction, which uses a fraction of `0.5`. There are other variants depending on whether the values are quantitative (i.e., they have a distance and the values can be interpolated) or merely ordinal (i.e., they can be ordered, but ties have to be broken.) Still other variants depend on whether the fraction is a single value or a list of values, but they can all be implemented in similar ways.

A common way to implement `quantile` that we saw in the Python example is to collect all the values into the state, sort them, and then read out the values at the requested positions. (This is probably why the SQL standard refers to it as an "ordered-set aggregate".) States can be combined by concatenation, which lets us group in parallel and build segment trees for windowing.

This approach is very time-consuming because sorting is `O(N log N)`, but happily for `quantile` we can use a related algorithm called `QuickSelect`, which can find a positional value in only `O(N)` time by *partially sorting* the array. You may have run into this algorithm if you have ever used the `std::nth_element` algorithm in the C++ standard library. This works well for grouped quantiles, but for moving quantiles the segment tree approach ends up being about 5% slower than just starting from scratch for each value.

To really improve the performance of moving quantiles, we note that the partial order probably does not change much between frames. If we maintain a list of indirect indices into the window and call `nth_element` on the indices, we can reorder the partially ordered indices instead of the values themselves. In the common case where the frame has the same size, we can even check to see whether the new value disrupts the partial ordering at all, and skip the reordering! With this approach, we can obtain a significant performance boost of 1.5-10 times.

In this example, we have a 3-element frame (green) that moves one space to the right for each value:

Row	Data						Median
0	0	1	2	0	1	2	0.5
1	0	1	2	0	1	2	1.0
2	0	1	2	0	1	2	1.0
3	0	1	2	0	1	2	1.0
4	0	1	2	0	1	2	1.0
5	0	1	2	0	1	2	1.5

The median values in orange must be computed from scratch. Notice that in the example, this only happens at the start of the window. The median values in white are computed using the existing partial ordering. In the example, this happens when the frame changes size. Finally, the median values in blue do not require reordering because the new value is the same as the old value. With this algorithm, we can create a faster implementation of single-fraction `quantile` without sorting.

InterQuartile Ranges (IQR)

We can extend this implementation to *lists* of fractions by leveraging the fact that each call to `nth_element` partially orders the values, which further improves performance. The "reuse" trick can be generalised to distinguish between fractions that are undisturbed and ones that need to be recomputed.

A common application of multiple fractions is computing interquartile ranges by using the fraction list `[0.25, 0.5, 0.75]`. This is the fraction list we use for the multiple fraction benchmarks. Combined with moving `MIN` and `MAX`, this moving aggregate can be used to generate the data for a moving box-and-whisker plot.

Median Absolute Deviation (MAD)

Maintaining the partial ordering can also be used to boost the performance of the median absolute deviation (or `mad`) aggregate. Unfortunately, the second partial ordering can't use the single value trick because the "function" being used to partially order the values will have changed if the data median changes. Still, the values are still probably not far off, which again improves the performance of `nth_element`.

Mode

The `mode` aggregate returns the most common value in a set. One common way to implement it is to accumulate all the values in the state, sort them and then scan for the longest run. These states can be combined by merging, which lets us compute the mode in parallel and build segment trees for windowing.

Once again, this approach is very time-consuming because sorting is `O(N log N)`. It may also use more memory than necessary because it keeps *all* the values instead of keeping only the unique values. If there are heavy-hitters in the list, (which is typically what `mode` is being used to find) this can be significant.

Another way to implement `mode` is to use a hash map for the state that maps values to counts. Hash tables are typically `O(N)` for accumulation, which is an improvement on sorting, and they only need to store unique values. If the state also tracks the largest value and count seen so far, we can just return that value when we finalize the aggregate. States can be combined by merging, which lets us group in parallel and build segment trees for windowing.

Unfortunately, as the benchmarks below demonstrate, this segment tree approach for windowing is quite slow! The overhead of merging the hash tables for the segment trees turns out to be about 5% slower than just building a new hash table for each row in the window. But for a moving `mode` computation, we can instead make a single hash table and update it every time the frame moves, removing the old values, adding the new values, and updating the value/count pair. At times the current mode value may have its count decremented, but when that happens we can rescan the table to find the new mode.

In this example, the 4-element frame (green) moves one space to the right for each value:

Row	Data							
0	c	d	c	b	g	b	e	d
1	c	d	c	b	g	b	e	d
2	c	d	c	b	g	b	e	d
3	c	d	c	b	g	b	e	d
4	c	d	c	b	g	b	e	d

Mode	Count
c	2
(g)	1
b	2
b	2
(d)	2

When the mode is unchanged (blue) it can be used directly. When the mode becomes ambiguous (orange), we must recan the table. This approach is much faster, and in the benchmarks it comes in between 15 and 55 times faster than the other two.

Microbenchmarks

To benchmark the various implementations, we run moving window queries against a 10M table of integers:

```
CREATE TABLE rank100 AS
    SELECT b % 100 AS a, b FROM range(10000000) tbl(b);
```

The results are then re-aggregated down to one row to remove the impact of streaming the results. The frames are 100 elements wide, and the test is repeated with a fixed trailing frame:

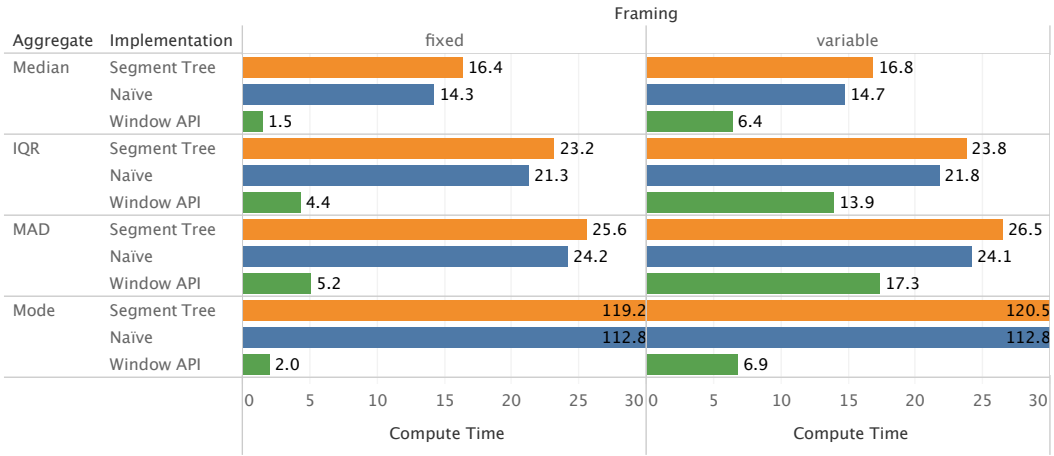
```
SELECT quantile_cont(a, [0.25, 0.5, 0.75]) OVER (
    ORDER BY b ASC
    ROWS BETWEEN 100 PRECEDING AND CURRENT ROW) AS iqr
FROM rank100;
```

and a variable frame that moves pseudo-randomly around the current value:

```
SELECT quantile_cont(a, [0.25, 0.5, 0.75]) OVER (
    ORDER BY b ASC
    ROWS BETWEEN mod(b * 47, 521) PRECEDING AND 100 - mod(b * 47, 521) FOLLOWING) AS iqr
FROM rank100;
```

The two examples here are the interquartile range queries; the other queries use the single argument aggregates `median`, `mad` and `mode`.

As a final step, we ran the same query with `count(*)`, which has the same overhead as the other benchmarks, but is trivial to compute (it just returns the frame size). That overhead was subtracted from the run times to give the algorithm timings:



As can be seen, there is a substantial benefit from implementing the window operation for all of these aggregates, often on the order of a factor of ten.

An unexpected finding was that the segment tree approach for these complex states is always slower (by about 5%) than simply creating the state for each output row. This suggests that when writing combinable complex aggregates, it is well worth benchmarking the aggregate and then considering providing a window operation instead of deferring to the segment tree machinery.

Conclusion

DuckDB's aggregate API enables aggregate functions to define a windowing operation that can significantly improve the performance of moving window computations for complex aggregates. This functionality has been used to significantly speed up windowing for several statistical aggregates, such as mode, interquartile ranges and median absolute deviation.

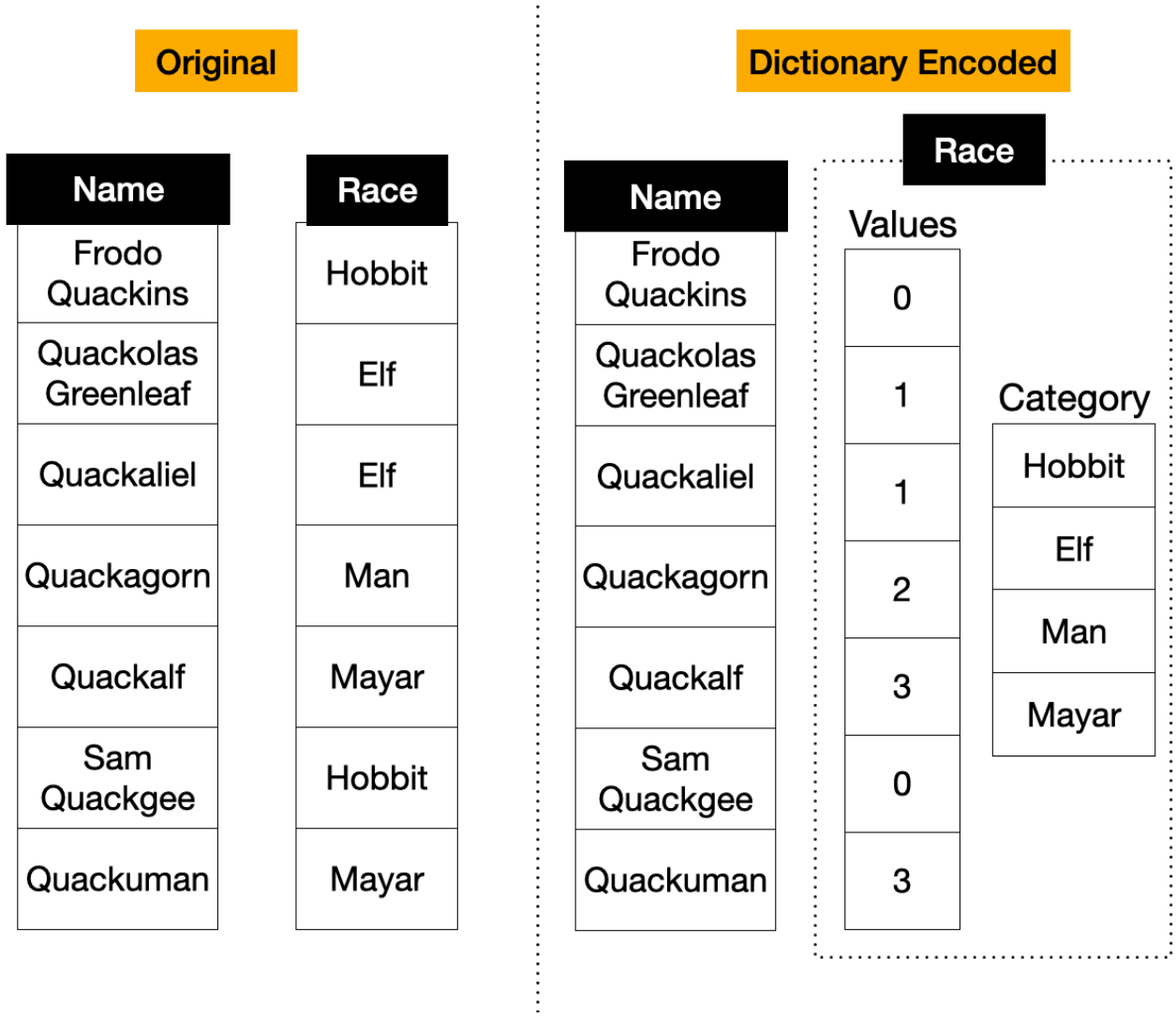
DuckDB is a free and open-source database management system (MIT licensed). It aims to be the SQLite for Analytics, and provides a fast and efficient database system with zero external dependencies. It is available not just for Python, but also for C/C++, R, Java, and more.

DuckDB – Lord of the Enums: The Fellowship of the Categorical and Factors

Publication date: 2021-11-26

Author: Pedro Holanda

String types are one of the most commonly used types. However, often string columns have a limited number of distinct values. For example, a country column will never have more than a few hundred unique entries. Storing a data type as a plain string causes a waste of storage and compromises query performance. A better solution is to dictionary encode these columns. In dictionary encoding, the data is split into two parts: the category and the values. The category stores the actual strings, and the values stores a reference to the strings. This encoding is depicted below.



In the old times, users would manually perform dictionary encoding by creating lookup tables and translating their ids back with join operations. Environments like Pandas and R support these types more elegantly. Pandas Categorical and R Factors are types that allow for columns of strings with many duplicate entries to be efficiently stored through dictionary encoding.

Dictionary encoding not only allows immense storage savings but also allows systems to operate on numbers instead of on strings, drastically boosting query performance. By lowering RAM usage, ENUMs also allow DuckDB to scale to significantly larger datasets.

To allow DuckDB to fully integrate with these encoded structures, we implemented Enum Types. This blog post will show code snippets of how to use ENUM types from both SQL API and Python/R clients, and will demonstrate the performance benefits of the enum types over using regular strings. To the best of our knowledge, DuckDB is the first RDBMS that natively integrates with Pandas categorical columns and R factors.

SQL

Our Enum SQL syntax is heavily inspired by Postgres. Below, we depict how to create and use the ENUM type.

```
CREATE TYPE lotr_race AS ENUM ('Mayar', 'Hobbit', 'Orc');

CREATE TABLE character (
    name text,
    race lotr_race
);
```

```
INSERT INTO character VALUES ('Frodo Quackins','Hobbit'), ('Quackalf ', 'Mayar');

-- We can perform a normal string comparison
-- Note that 'Hobbit' will be cast to a lotr_race
-- hence this comparison is actually a fast integer comparison
SELECT name FROM character WHERE race = 'Hobbit';
----
Frodo Quackins
```

ENUM columns behave exactly the same as normal VARCHAR columns. They can be used in string functions (such as LIKE or substring), they can be compared, ordered, etc. The only exception is that ENUM columns can only hold the values that are specified in the enum definition. Inserting a value that is not part of the enum definition will result in an error.

DuckDB ENUMs are currently static (i.e., values can not be added or removed after the ENUM definition). However, ENUM updates are on the roadmap for the next version.

See the documentation for more information.

Python

Setup

First we need to install DuckDB and Pandas. The installation process of both libraries in Python is straightforward:

```
# Python Install
pip install duckdb
pip install pandas
```

Usage

Pandas columns from the categorical type are directly converted to DuckDB's ENUM types:

```
import pandas as pd
import duckdb

# Our unencoded data.
data = ['Hobbit', 'Elf', 'Elf', 'Man', 'Mayar', 'Hobbit', 'Mayar']

# 'pd.Categorical' automatically encodes the data as a categorical column
df_in = pd.DataFrame({'races': pd.Categorical(data),})

# We can query this dataframe as we would any other
# The conversion from categorical columns to enums happens automatically
df_out = duckdb.execute("SELECT * FROM df_in").df()
```

R

Setup

We only need to install DuckDB in our R client, and we are ready to go.

```
# R Install
install.packages("duckdb")
```

Usage

Similar to our previous example with Pandas, R Factor columns are also automatically converted to DuckDB's ENUM types.

```r
library ("duckdb")

con <- dbConnect(duckdb::duckdb())
on.exit(dbDisconnect(con, shutdown = TRUE))

# Our unencoded data.
data <- c('Hobbit', 'Elf', 'Elf', 'Man', 'Mayar', 'Hobbit', 'Mayar')

# Our R dataframe holding an encoded version of our data column
# 'as.factor' automatically encodes it.
df_in <- data.frame(races=as.factor(data))


duckdb::duckdb_register(con, "characters", df_in)
df_out <- dbReadTable(con, "characters")
```

Benchmark Comparison

To demonstrate the performance of DuckDB when running operations on categorical columns of Pandas DataFrames, we present a number of benchmarks. The source code for the benchmarks is available on GitHub. In our benchmarks we always consume and produce Pandas DataFrames.

Dataset

Our dataset is composed of one dataframe with 4 columns and 10 million rows. The first two columns are named `race` and `subrace` representing races. They are both categorical, with the same categories but different values. The other two columns `race_string` and `subrace_string` are the string representations of `race` and ‘subrace“‘.

```python
def generate_df(size):
  race_categories = ['Hobbit', 'Elf', 'Man', 'Mayar']
  race = np.random.choice(race_categories, size)
  subrace = np.random.choice(race_categories, size)
  return pd.DataFrame({'race': pd.Categorical(race),
                       'subrace': pd.Categorical(subrace),
                       'race_string': race,
                       'subrace_string': subrace,})

size = pow(10,7) #10,000,000 rows
df = generate_df(size)
```

Grouped Aggregation

In our grouped aggregation benchmark, we do a count of how many characters for each race we have in the `race` or `race_string` column of our table.

```python
def duck_categorical(df):
  return con.execute("SELECT race, count(*) FROM df GROUP BY race").df()

def duck_string(df):
  return con.execute("SELECT race_string, count(*) FROM df GROUP BY race_string").df()

def pandas(df):
  return df.groupby(['race']).agg({'race': 'count'})

def pandas_string(df):
  return df.groupby(['race_string']).agg({'race_string': 'count'})
```

The table below depicts the timings of this operation. We can see the benefits of performing grouping on encoded values over strings, with DuckDB being 4× faster when grouping small unsigned values.

Name	Time (s)
DuckDB (Categorical)	0.01
DuckDB (String)	0.04
Pandas (Categorical)	0.06
Pandas (String)	0.40

Filter

In our filter benchmark, we do a count of how many Hobbit characters we have in the `race` or `race_string` column of our table.

```
def duck_categorical(df):
  return con.execute("SELECT count(*) FROM df WHERE race = 'Hobbit'").df()

def duck_string(df):
  return con.execute("SELECT count(*) FROM df WHERE race_string = 'Hobbit'").df()

def pandas(df):
  filtered_df = df[df.race == "Hobbit"]
  return filtered_df.agg({'race': 'count'})

def pandas_string(df):
  filtered_df = df[df.race_string == "Hobbit"]
  return filtered_df.agg({'race_string': 'count'})
```

For the DuckDB enum type, DuckDB converts the string `Hobbit` to a value in the `ENUM`, which returns an unsigned integer. We can then do fast numeric comparisons, instead of expensive string comparisons, which results in greatly improved performance.

Name	Time (s)
DuckDB (Categorical)	0.003
DuckDB (String)	0.023
Pandas (Categorical)	0.158
Pandas (String)	0.440

Enum – Enum Comparison

In this benchmark, we perform an equality comparison of our two breed columns. `race` and `subrace` or `race_string` and ‘subrace_string“‘

```
def duck_categorical(df):
  return con.execute("SELECT count(*) FROM df WHERE race = subrace").df()

def duck_string(df):
  return con.execute("SELECT count(*) FROM df WHERE race_string = subrace_string").df()

def pandas(df):
  filtered_df = df[df.race == df.subrace]
  return filtered_df.agg({'race': 'count'})

def pandas_string(df):
  filtered_df = df[df.race_string == df.subrace_string]
  return filtered_df.agg({'race_string': 'count'})
```

DuckDB ENUMs can be compared directly on their encoded values. This results in a time difference similar to the previous case, again because we are able to compare numeric values instead of strings.

Name	Time (s)
DuckDB (Categorical)	0.005
DuckDB (String)	0.040
Pandas (Categorical)	0.130
Pandas (String)	0.550

Storage

In this benchmark, we compare the storage savings of storing ENUM Types vs Strings.

```
race_categories = ['Hobbit', 'Elf', 'Man','Mayar']
race = np.random.choice(race_categories, size)
categorical_race = pd.DataFrame({'race': pd.Categorical(race),})
string_race = pd.DataFrame({'race': race,})
con = duckdb.connect('duck_cat.db')
con.execute("CREATE TABLE character AS SELECT * FROM categorical_race")
con = duckdb.connect('duck_str.db')
con.execute("CREATE TABLE character AS SELECT * FROM string_race")
```

The table below depicts the DuckDB file size differences when storing the same column as either an Enum or a plain string. Since the dictionary-encoding does not repeat the string values, we can see a reduction of one order of magnitude in size.

Name	Size (MB)
DuckDB (Categorical)	11
DuckDB (String)	102

What about the Sequels?

There are three main directions we will pursue in the following versions of DuckDB related to ENUMs.

1. Automatic Storage Encoding: As described in the introduction, users frequently define database columns as Strings when in reality they are ENUMs. Our idea is to automatically detect and dictionary-encode these columns, without any input of the user and in a way that is completely invisible to them.
2. ENUM Updates: As said in the introduction, our ENUMs are currently static. We will allow the insertion and removal of ENUM categories.
3. Integration with other Data Formats: We want to expand our integration with data formats that implement ENUM-like structures.

Feedback

If you encounter any problems when using our ENUMs, please open an issue in our issue tracker!

DuckDB Quacks Arrow: A Zero-Copy Data Integration between Apache Arrow and DuckDB

Publication date: 2021-12-03

Authors: Pedro Holanda, Jonathan Keane

TL;DR: The zero-copy integration between DuckDB and Apache Arrow allows for rapid analysis of larger than memory datasets in Python and R using either SQL or relational APIs.

This post is a collaboration with and cross-posted on the Arrow blog.

Part of Apache Arrow is an in-memory data format optimized for analytical libraries. Like Pandas and R Dataframes, it uses a columnar data model. But the Arrow project contains more than just the format: The Arrow C++ library, which is accessible in Python, R, and Ruby via bindings, has additional features that allow you to compute efficiently on datasets. These additional features are on top of the implementation of the in-memory format described above. The datasets may span multiple files in Parquet, CSV, or other formats, and files may even be on remote or cloud storage like HDFS or Amazon S3. The Arrow C++ query engine supports the streaming of query results, has an efficient implementation of complex data types (e.g., Lists, Structs, Maps), and can perform important scan optimizations like Projection and Filter Pushdown.

DuckDB is a new analytical data management system that is designed to run complex SQL queries within other processes. DuckDB has bindings for R and Python, among others. DuckDB can query Arrow datasets directly and stream query results back to Arrow. This integration allows users to query Arrow data using DuckDB's SQL Interface and API, while taking advantage of DuckDB's parallel vectorized execution engine, without requiring any extra data copying. Additionally, this integration takes full advantage of Arrow's predicate and filter pushdown while scanning datasets.

This integration is unique because it uses zero-copy streaming of data between DuckDB and Arrow and vice versa so that you can compose a query using both together. This results in three main benefits:

1. **Larger Than Memory Analysis:** Since both libraries support streaming query results, we are capable of executing on data without fully loading it from disk. Instead, we can execute one batch at a time. This allows us to execute queries on data that is bigger than memory.
2. **Complex Data Types:** DuckDB can efficiently process complex data types that can be stored in Arrow vectors, including arbitrarily nested structs, lists, and maps.
3. **Advanced Optimizer:** DuckDB's state-of-the-art optimizer can push down filters and projections directly into Arrow scans. As a result, only relevant columns and partitions will be read, allowing the system to e.g., take advantage of partition elimination in Parquet files. This significantly accelerates query execution.

For those that are just interested in benchmarks, you can jump ahead benchmark section below.

Quick Tour

Before diving into the details of the integration, in this section we provide a quick motivating example of how powerful and simple to use is the DuckDB-Arrow integration. With a few lines of code, you can already start querying Arrow datasets. Say you want to analyze the infamous NYC Taxi Dataset and figure out if groups tip more or less than single riders.

R

Both Arrow and DuckDB support dplyr pipelines for people more comfortable with using dplyr for their data analysis. The Arrow package includes two helper functions that allow us to pass data back and forth between Arrow and DuckDB (`to_duckdb()` and `to_arrow()`).

This is especially useful in cases where something is supported in one of Arrow or DuckDB but not the other. For example, if you find a complex dplyr pipeline where the SQL translation doesn't work with DuckDB, use `to_arrow()` before the pipeline to use the Arrow engine. Or, if you have a function (e.g., windowed aggregates) that aren't yet implemented in Arrow, use `to_duckdb()` to use the DuckDB engine. All while not paying any cost to (re)serialize the data when you pass it back and forth!

```
library(duckdb)
library(arrow)
library(dplyr)

# Open dataset using year, month folder partition
ds <- arrow::open_dataset("nyc-taxi", partitioning = c("year", "month"))

ds %>%
  # Look only at 2015 on, where the number of passenger is positive, the trip distance is
  # greater than a quarter mile, and where the fare amount is positive
  filter(year > 2014 & passenger_count > 0 & trip_distance > 0.25 & fare_amount > 0) %>%
  # Pass off to DuckDB
  to_duckdb() %>%
  group_by(passenger_count) %>%
  mutate(tip_pct = tip_amount / fare_amount) %>%
  summarise(
    fare_amount = mean(fare_amount, na.rm = TRUE),
    tip_amount = mean(tip_amount, na.rm = TRUE),
    tip_pct = mean(tip_pct, na.rm = TRUE)
  ) %>%
  arrange(passenger_count) %>%
  collect()
```

Python

The workflow in Python is as simple as it is in R. In this example we use DuckDB's Relational API.

```
import duckdb
import pyarrow as pa
import pyarrow.dataset as ds

# Open dataset using year, month folder partition
nyc = ds.dataset('nyc-taxi/', partitioning=["year", "month"])

# We transform the nyc dataset into a DuckDB relation
nyc = duckdb.arrow(nyc)

# Run same query again
nyc.filter("year > 2014 & passenger_count > 0 & trip_distance > 0.25 & fare_amount > 0")
    .aggregate("SELECT avg(fare_amount), avg(tip_amount), avg(tip_amount / fare_amount) AS tip_pct",
"passenger_count").arrow()
```

DuckDB and Arrow: The Basics

In this section, we will look at some basic examples of the code needed to read and output Arrow tables in both Python and R.

Setup

First we need to install DuckDB and Arrow. The installation process for both libraries is shown below.

Python:

```
pip install duckdb
pip install pyarrow
```

R:

```
install.packages("duckdb")
install.packages("arrow")
```

To execute the sample examples in this section, we need to download the following custom Parquet files:

- integers.parquet
- lineitemsf1.snappy.parquet

Python There are two ways in Python of querying data from Arrow.

1. Through the Relational API:

```
# Reads Parquet File to an Arrow Table
arrow_table = pq.read_table('integers.parquet')

# Transforms Arrow Table -> DuckDB Relation
rel_from_arrow = duckdb.arrow(arrow_table)

# we can run a SQL query on this and print the result
print(rel_from_arrow.query('arrow_table', 'SELECT sum(data) FROM arrow_table WHERE data >
50').fetchone())

# Transforms DuckDB Relation -> Arrow Table
arrow_table_from_duckdb = rel_from_arrow.arrow()
```

2. By using replacement scans and querying the object directly with SQL:

```
# Reads Parquet File to an Arrow Table
arrow_table = pq.read_table('integers.parquet')

# Gets Database Connection
con = duckdb.connect()

# we can run a SQL query on this and print the result
print(con.execute('SELECT sum(data) FROM arrow_table WHERE data > 50').fetchone())

# Transforms Query Result from DuckDB to Arrow Table
# We can directly read the arrow object through DuckDB's replacement scans.
con.execute("SELECT * FROM arrow_table").fetch_arrow_table()
```

It is possible to transform both DuckDB Relations and Query Results back to Arrow.

R In R, you can interact with Arrow data in DuckDB by registering the table as a view (an alternative is to use dplyr as shown above).

```
library(duckdb)
library(arrow)
library(dplyr)

# Reads Parquet File to an Arrow Table
arrow_table <- arrow::read_parquet("integers.parquet", as_data_frame = FALSE)

# Gets Database Connection
con <- dbConnect(duckdb::duckdb())

# Registers arrow table as a DuckDB view
arrow::to_duckdb(arrow_table, table_name = "arrow_table", con = con)

# we can run a SQL query on this and print the result
print(dbGetQuery(con, "SELECT sum(data) FROM arrow_table WHERE data > 50"))

# Transforms Query Result from DuckDB to Arrow Table
result <- dbSendQuery(con, "SELECT * FROM arrow_table")
```

Streaming Data from/to Arrow

In the previous section, we depicted how to interact with Arrow tables. However, Arrow also allows users to interact with the data in a streaming fashion. Either consuming it (e.g., from an Arrow Dataset) or producing it (e.g., returning a RecordBatchReader). And of course, DuckDB is able to consume Datasets and produce RecordBatchReaders. This example uses the NYC Taxi Dataset, stored in Parquet files partitioned by year and month, which we can download through the Arrow R package:

```
arrow::copy_files("s3://ursa-labs-taxi-data", "nyc-taxi")
```

Python

```
# Reads dataset partitioning it in year/month folder
nyc_dataset = ds.dataset('nyc-taxi/', partitioning=["year", "month"])

# Gets Database Connection
con = duckdb.connect()

query = con.execute("SELECT * FROM nyc_dataset")
# DuckDB's queries can now produce a Record Batch Reader
record_batch_reader = query.fetch_record_batch()
# Which means we can stream the whole query per batch.
# This retrieves the first batch
chunk = record_batch_reader.read_next_batch()
```

R

```
# Reads dataset partitioning it in year/month folder
nyc_dataset = open_dataset("nyc-taxi/", partitioning = c("year", "month"))

# Gets Database Connection
con <- dbConnect(duckdb::duckdb())

# We can use the same function as before to register our arrow dataset
duckdb::duckdb_register_arrow(con, "nyc", nyc_dataset)

res <- dbSendQuery(con, "SELECT * FROM nyc", arrow = TRUE)
# DuckDB's queries can now produce a Record Batch Reader
record_batch_reader <- duckdb::duckdb_fetch_record_batch(res)

# Which means we can stream the whole query per batch.
# This retrieves the first batch
cur_batch <- record_batch_reader$read_next_batch()
```

The preceding R code shows in low-level detail how the data is streaming. We provide the helper `to_arrow()` in the Arrow package which is a wrapper around this that makes it easy to incorporate this streaming into a dplyr pipeline.

> In Arrow 6.0.0, `to_arrow()` currently returns the full table, but will allow full streaming in our upcoming 7.0.0 release.

Benchmark Comparison

Here we demonstrate in a simple benchmark the performance difference between querying Arrow datasets with DuckDB and querying Arrow datasets with Pandas. For both the Projection and Filter pushdown comparison, we will use Arrow tables. That is due to Pandas not being capable of consuming Arrow stream objects.

For the NYC Taxi benchmarks, we used a server in the SciLens cluster and for the TPC-H benchmarks, we used a MacBook Pro with an M1 CPU. In both cases, parallelism in DuckDB was used (which is now on by default).

For the comparison with Pandas, note that DuckDB runs in parallel, while pandas only support single-threaded execution. Besides that, one should note that we are comparing automatic optimizations. DuckDB's query optimizer can automatically push down filters and projections. This automatic optimization is not supported in pandas, but it is possible for users to manually perform some of these predicate and filter pushdowns by manually specifying them in the `read_parquet()` call.

Projection Pushdown

In this example we run a simple aggregation on two columns of our lineitem table.

```
# DuckDB
lineitem = pq.read_table('lineitemsf1.snappy.parquet')
con = duckdb.connect()

# Transforms Query Result from DuckDB to Arrow Table
con.execute("""SELECT sum(l_extendedprice * l_discount) AS revenue
               FROM
               lineitem;""").fetch_arrow_table()

# Pandas
arrow_table = pq.read_table('lineitemsf1.snappy.parquet')

# Converts an Arrow table to a Dataframe
df = arrow_table.to_pandas()

# Runs aggregation
res =  pd.DataFrame({'sum': [(df.l_extendedprice * df.l_discount).sum()]})

# Creates an Arrow Table from a Dataframe
new_table = pa.Table.from_pandas(res)
```

Name	Time (s)
DuckDB	0.19
Pandas	2.13

The lineitem table is composed of 16 columns, however, to execute this query only two columns `l_extendedprice` and `l_discount` are necessary. Since DuckDB can push down the projection of these columns, it is capable of executing this query about one order of magnitude faster than Pandas.

Filter Pushdown

For our filter pushdown we repeat the same aggregation used in the previous section, but add filters on 4 more columns.

```
# DuckDB
lineitem = pq.read_table('lineitemsf1.snappy.parquet')

# Get database connection
con = duckdb.connect()

# Transforms Query Result from DuckDB to Arrow Table
con.execute("""SELECT sum(l_extendedprice * l_discount) AS revenue
        FROM
            lineitem
        WHERE
            l_shipdate >= CAST('1994-01-01' AS date)
            AND l_shipdate < CAST('1995-01-01' AS date)
            AND l_discount BETWEEN 0.05
            AND 0.07
            AND l_quantity < 24; """).fetch_arrow_table()

# Pandas
arrow_table = pq.read_table('lineitemsf1.snappy.parquet')

df = arrow_table.to_pandas()
filtered_df = lineitem[
```

```
        (lineitem.l_shipdate >= "1994-01-01") &
        (lineitem.l_shipdate < "1995-01-01") &
        (lineitem.l_discount >= 0.05) &
        (lineitem.l_discount <= 0.07) &
        (lineitem.l_quantity < 24)]

res =  pd.DataFrame({'sum': [(filtered_df.l_extendedprice * filtered_df.l_discount).sum()]})
new_table = pa.Table.from_pandas(res)
```

Name	Time (s)
DuckDB	0.04
Pandas	2.29

The difference now between DuckDB and Pandas is more drastic, being two orders of magnitude faster than Pandas. Again, since both the filter and projection are pushed down to Arrow, DuckDB reads less data than Pandas, which can't automatically perform this optimization.

Streaming

As demonstrated before, DuckDB is capable of consuming and producing Arrow data in a streaming fashion. In this section we run a simple benchmark, to showcase the benefits in speed and memory usage when comparing it to full materialization and Pandas. This example uses the full NYC taxi dataset which you can download

```
# DuckDB
# Open dataset using year, month folder partition
nyc = ds.dataset('nyc-taxi/', partitioning=["year", "month"])

# Get database connection
con = duckdb.connect()

# Run query that selects part of the data
query = con.execute("SELECT total_amount, passenger_count, year FROM nyc where total_amount > 100 and year
> 2014")

# Create Record Batch Reader from Query Result.
# "fetch_record_batch()" also accepts an extra parameter related to the desired produced chunk size.
record_batch_reader = query.fetch_record_batch()

# Retrieve all batch chunks
chunk = record_batch_reader.read_next_batch()
while len(chunk) > 0:
    chunk = record_batch_reader.read_next_batch()

# Pandas
# We must exclude one of the columns of the NYC dataset due to an unimplemented cast in Arrow.
working_columns = ["vendor_id","pickup_at","dropoff_at","passenger_count","trip_distance","pickup_
longitude",
    "pickup_latitude","store_and_fwd_flag","dropoff_longitude","dropoff_latitude","payment_type",
    "fare_amount","extra","mta_tax","tip_amount","tolls_amount","total_amount","year", "month"]

# Open dataset using year, month folder partition
nyc_dataset = ds.dataset(dir, partitioning=["year", "month"])
# Generate a scanner to skip problematic column
dataset_scanner = nyc_dataset.scanner(columns=working_columns)

# Materialize dataset to an Arrow Table
nyc_table = dataset_scanner.to_table()

# Generate Dataframe from Arow Table
```

```
nyc_df = nyc_table.to_pandas()

# Apply Filter
filtered_df = nyc_df[
    (nyc_df.total_amount > 100) &
    (nyc_df.year >2014)]

# Apply Projection
res = filtered_df[["total_amount", "passenger_count","year"]]

# Transform Result back to an Arrow Table
new_table = pa.Table.from_pandas(res)
```

Name	Time (s)	Peak memory usage (GBs)
DuckDB	0.05	0.3
Pandas	146.91	248

The difference in times between DuckDB and Pandas is a combination of all the integration benefits we explored in this article. In DuckDB the filter pushdown is applied to perform partition elimination (i.e., we skip reading the Parquet files where the year is <= 2014). The filter pushdown is also used to eliminate unrelated row_groups (i.e., row groups where the total amount is always <= 100). Due to our projection pushdown, Arrow only has to read the columns of interest from the Parquet files, which allows it to read only 4 out of 20 columns. On the other hand, Pandas is not capable of automatically pushing down any of these optimizations, which means that the full dataset must be read. **This results in the 4 orders of magnitude difference in query execution time.**

In the table above, we also depict the comparison of peak memory usage between DuckDB (Streaming) and Pandas (Fully-Materializing). In DuckDB, we only need to load the row group of interest into memory. Hence our memory usage is low. We also have constant memory usage since we only have to keep one of these row groups in-memory at a time. Pandas, on the other hand, has to fully materialize all Parquet files when executing the query. Because of this, we see a constant steep increase in its memory consumption. **The total difference in memory consumption of the two solutions is around 3 orders of magnitude.**

Conclusion and Feedback

In this blog post, we mainly showcased how to execute queries on Arrow datasets with DuckDB. There are additional libraries that can also consume the Arrow format but they have different purposes and capabilities.

If you encounter any problems when using our integration, please open an issue in either DuckDB's issue tracker or Arrow's issue tracker, depending on which library has a problem.

DuckDB Time Zones: Supporting Calendar Extensions

Publication date: 2022-01-06

Author: Richard Wesley

TL;DR: The DuckDB ICU extension now provides time zone support.

Time zone support is a common request for temporal analytics, but the rules are complex and somewhat arbitrary. The most well supported library for locale-specific operations is the International Components for Unicode (ICU). DuckDB already provided collated string comparisons using ICU via an extension (to avoid dependencies), and we have now connected the existing ICU calendar and time zone functions to the main code via the new `TIMESTAMP WITH TIME ZONE` (or `TIMESTAMPTZ` for short) data type. The ICU extension is pre-bundled in DuckDB's Python client and can be optionally installed in the remaining clients.

In this post, we will describe how time works in DuckDB and what time zone functionality has been added.

What Is Time?

> Quote People assume that time is a strict progression of cause to effect, but actually from a non-linear, non-subjective viewpoint it's more like a big ball of wibbly wobbly timey wimey stuff.
>
> -- Doctor Who: Blink

Time in databases can be very confusing because the way we talk about time is itself confusing. Local time, GMT, UTC, time zones, leap years, proleptic Gregorian calendars – it all looks like a big mess. But if you step back, modeling time is actually fairly simple, and can be reduced to two pieces: instants and binning.

Instants

You will often hear people (and documentation) say that database time is stored in UTC. This is sort of right, but it is more accurate to say that databases store *instants*. An instant is a point in universal time, and they are usually given as a count of some time increment from a fixed point in time (called the *epoch*). In DuckDB, the fixed point is the Unix epoch `1970-01-01 00:00:00 +00:00`, and the increment is microseconds (µs). (Note that to avoid confusion we will be using ISO 8601 y-m-d notation in this post to denote instants.) In other words, a `TIMESTAMP` column contains instants.

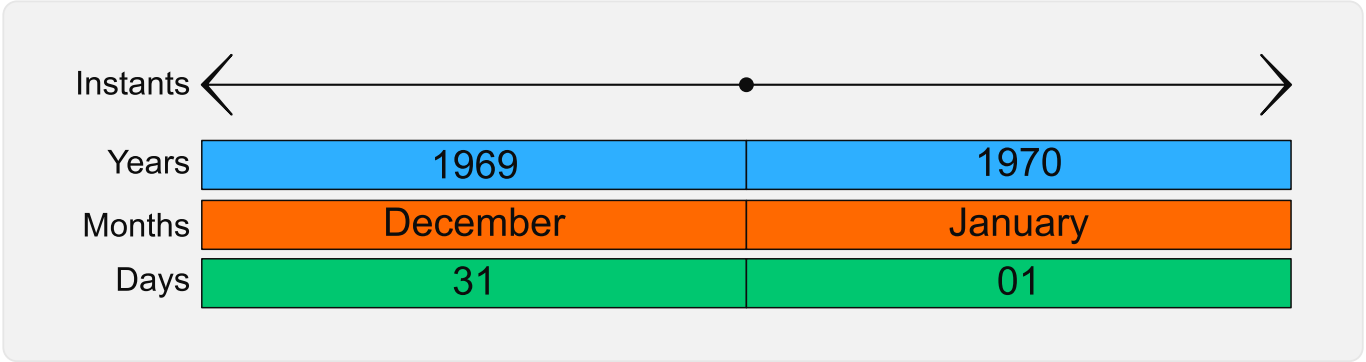
There are three other temporal types in SQL:

- `DATE` – an integral count of days from a fixed date. In DuckDB, the fixed date is `1970-01-01`, again in UTC.
- `TIME` – a (positive) count of microseconds up to a single day
- `INTERVAL` – a set of fields for counting time differences. In DuckDB, intervals count months, days and microseconds. (Months are not completely well-defined, but when present, they represent 30 days.)

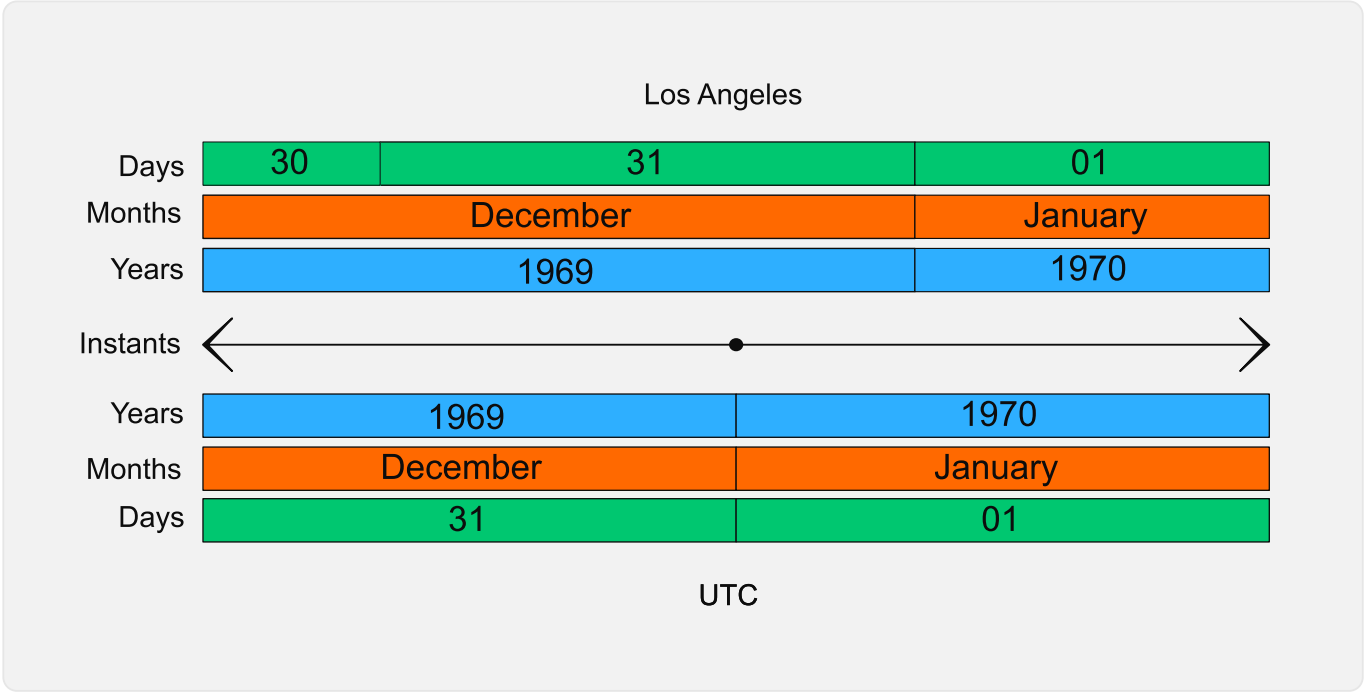
None of these other temporal types except `TIME` can have a `WITH TIME ZONE` modifier (and shorter `TZ` suffix), but to understand what that modifier means, we first need to talk about *temporal binning*.

Temporal Binning

Instants are pretty straightforward – they are just a number – but binning is the part that trips people up. Binning is probably a familiar idea if you have worked with continuous data: You break up a set of values into ranges and map each value to the range (or *bin*) that it falls into. Temporal binning is just doing this to instants:

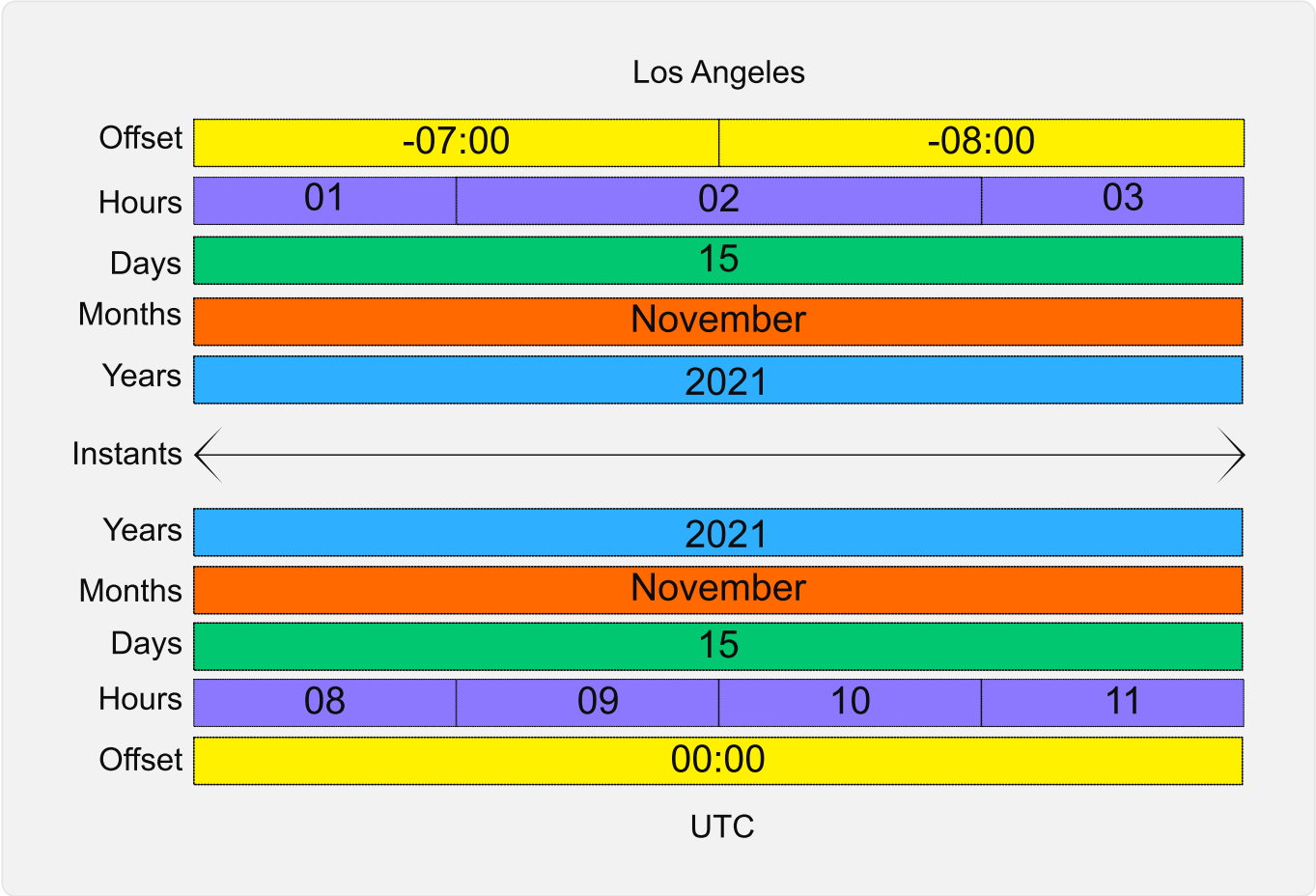


Temporal binning systems are often called *calendars*, but we are going to avoid that term for now because calendars are usually associated with dates, and temporal binning also includes rules for time. These time rules are called *time zones*, and they also impact where the day boundaries used by the calendar fall. For example, here is what the binning for a second time zone looks like at the epoch:



The most confusing thing about temporal binning is that there is more than one way to bin time, and it is not always obvious what binning should be used. For example, what I mean by “today” is a bin of instants often determined by where I live. Every instant that is part of my “today” goes in that bin. But notice that I qualified “today” with “where I live”, and that qualification determines what binning system is being used. But “today” could also be determined by “where the events happened”, which would require a different binning to be applied.

The biggest temporal binning problem most people run into occurs when daylight savings time changes. This example contains a daylight savings time change where the “hour” bin is two hours long! To distinguish the two hours, we needed to include another bin containing the offset from UTC:



As this example shows, in order to bin the instants correctly, we need to know the binning rules that apply. It also shows that we can't just use the built in binning operations, because they don't understand daylight savings time.

Naïve Timestamps

Instants are sometimes created from a string format using a local binning system instead of an instant. This results in the instants being offset from UTC, which can cause problems with daylight savings time. These are called *naïve* timestamps, and they may constitute a data cleaning problem.

Cleaning naïve timestamps requires determining the offset for each timestamp and then updating the value to be an instant. For most values, this can be done with an inequality join against a table containing the correct offsets, but the ambiguous values may need to be fixed by hand. It may also be possible to correct the ambiguous values by assuming that they were inserted in order and looking for "backwards jumps" using window functions.

A simple way to avoid this situation going forward is to add the UTC offset to non-UTC strings: `2021-07-31 07:20:15 -07:00`. The DuckDB VARCHAR cast operation parses these offsets correctly and will generate the corresponding instant.

Time Zone Data Types

The SQL standard defines temporal data types qualified by `WITH TIME ZONE`. This terminology is confusing because it seems to imply that the time zone will be stored with the value, but what it really means is "bin this value using the session's `TimeZone` setting". Thus a `TIMESTAMPTZ` column also stores instants, but expresses a "hint" that it should use a specific binning system.

There are a number of operations that can be performed on instants without a binning system:

- Comparing;

- Sorting;
- Increment (µs) difference;
- Casting to and from regular `TIMESTAMP`s.

These common operations have been implemented in the main DuckDB code base, while the binning operations have been delegated to extensions such as ICU.

One small difference between the display of the new `WITH TIME ZONE` types and the older types is that the new types will be displayed with a `+00` UTC offset. This is simply to make the type differences visible in command line interfaces and for testing. Properly formatting a `TIMESTAMPTZ` for display in a locale requires using a binning system.

ICU Temporal Binning

DuckDB already uses an ICU extension for collating strings for a particular locale, so it was natural to extend it to expose the ICU calendar and time zone functionality.

ICU Time Zones

The first step for supporting time zones is to add the `TimeZone` setting that should be applied. DuckDB extensions can define and validate their own settings, and the ICU extension now does this:

```
-- Load the extension
-- This is not needed in Python or R, as the extension is already installed
LOAD icu;

-- Show the current time zone. The default is set to ICU's current time zone.
SELECT * FROM duckdb_settings() WHERE name = 'TimeZone';
```

```
TimeZone    Europe/Amsterdam    The current time zone    VARCHAR
```

```
-- Choose a time zone.
SET TimeZone = 'America/Los_Angeles';

-- Emulate Postgres' time zone table
SELECT name, abbrev, utc_offset
FROM pg_timezone_names()
ORDER BY 1
LIMIT 5;
```

```
ACT ACT 09:30:00
AET AET 10:00:00
AGT AGT -03:00:00
ART ART 02:00:00
AST AST -09:00:00
```

ICU Temporal Binning Functions

Databases like DuckDB and Postgres usually provide some temporal binning functions such as `YEAR` or `DATE_PART`. These functions are part of a single binning system for the conventional (proleptic Gregorian) calendar and the UTC time zone. Note that casting to a string is a binning operation because the text produced contains bin values.

Because timestamps that require custom binning have a different data type, the ICU extension can define additional functions with bindings to `TIMESTAMPTZ`:

- `+` – Add an `INTERVAL` to a timestamp
- `-` – Subtract an `INTERVAL` from a timestamp
- `AGE` – Compute an `INTERVAL` describing the months/days/microseconds between two timestamps (or one timestamp and the current instant).
- `DATE_DIFF` – Count part boundary crossings between two timestamp

- DATE_PART – Extract a named timestamp part. This includes the part alias functions such as YEAR.
- DATE_SUB – Count the number of complete parts between two timestamp
- DATE_TRUNC – Truncate a timestamp to the given precision
- LAST_DAY – Returns the last day of the month
- MAKE_TIMESTAMPTZ – Constructs a TIMESTAMPTZ from parts, including an optional final time zone specifier.

We have not implemented these functions for TIMETZ because this type has limited utility, but it would not be difficult to add in the future. We have also not implemented string formatting/casting to VARCHAR because the type casting system is not yet extensible, and the current ICU build we are using does not embed this data.

ICU Calendar Support

ICU can also perform binning operations for some non-Gregorian calendars. We have added support for these calendars via a `Calendar` setting and the `icu_calendar_names` table function:

```
LOAD icu;

-- Show the current calendar. The default is set to ICU's current locale.
SELECT * FROM duckdb_settings() WHERE name = 'Calendar';
```

```
Calendar    gregorian    The current calendar    VARCHAR
```

```
-- List the available calendars
SELECT DISTINCT name FROM icu_calendar_names()
ORDER BY 1 DESC LIMIT 5;
```

```
roc
persian
japanese
iso8601
islamic-umalqura
```

```
-- Choose a calendar
SET Calendar = 'japanese';

-- Extract the current Japanese era number using Tokyo time
SET TimeZone = 'Asia/Tokyo';

SELECT
     era('2019-05-01 00:00:00+10'::TIMESTAMPTZ),
     era('2019-05-01 00:00:00+09'::TIMESTAMPTZ);
```

```
235  236
```

Caveats

ICU has some differences in behavior and representation from the DuckDB implementation. These are hopefully minor issues that should only be of concern to serious time nerds.

- ICU represents instants as millisecond counts using a DOUBLE. This makes it lose accuracy far from the epoch (e.g., around the first millennium)
- ICU uses the Julian calendar for dates before the Gregorian change on `1582-10-15` instead of the proleptic Gregorian calendar. This means that dates prior to the changeover will differ, although ICU will give the date as actually written at the time.
- ICU computes ages by using part increments instead of using the length of the earlier month like DuckDB and Postgres.

Future Work

Temporal analysis is a large area, and while the ICU time zone support is a big step forward, there is still much that could be done. Some of these items are core DuckDB improvements that could benefit all temporal binning systems and some expose more ICU functionality. There is also the prospect for writing other custom binning systems via extensions.

DuckDB Features

Here are some general projects that all binning systems could benefit from:

- Add a `DATE_ROLL` function that emulates the ICU calendar `roll` operation for "rotating" around a containing bin;
- Making casting operations extensible so extensions can add their own support;

ICU Functionality

ICU is a very rich library with a long pedigree, and there is much that could be done with the existing library:

- Create a more general `MAKE_TIMESTAPTZ` variant that takes a `STRUCT` with the parts. This could be useful for some non-Gregorian calendars.
- Extend the embedded data to contain locale temporal information (such as month names) and support formatting (`to_char`) and parsing (`to_timestamp`) of local dates. One issue here is that the ICU date formatting language is more sophisticated than the Postgres language, so multiple functions might be required (e.g., `icu_to_char`);
- Extend the binning functions to take per-row calendar and time zone specifications to support row-level temporal analytics such as "what time of day did this happen"?

Separation of Concerns

Because the time zone data type is defined in the main code base, but the calendar operations are provided by an extension, it is now possible to write application-specific extensions with custom calendar and time zone support such as:

- Financial 4-4-5 calendars;
- ISO week-based years;
- Table-driven calendars;
- Astronomical calendars with leap seconds;
- Fun calendars, such as Shire Reckoning and French Republican!

Conclusion and Feedback

In this blog post, we described the new DuckDB time zone functionality as implemented via the ICU extension. We hope that the functionality provided can enable temporal analytic applications involving time zones. We also look forward to seeing any custom calendar extensions that our users dream up!

Last but not least, if you encounter any problems when using our integration, please open an issue in DuckDB's issue tracker!

Parallel Grouped Aggregation in DuckDB

Publication date: 2022-03-07

Authors: Hannes Mühleisen, Mark Raasveldt

TL;DR: DuckDB has a fully parallelized aggregate hash table that can efficiently aggregate over millions of groups.

Grouped aggregations are a core data analysis command. It is particularly important for large-scale data analysis ("OLAP") because it is useful for computing statistical summaries of huge tables. DuckDB contains a highly optimized parallel aggregation capability for fast and scalable summarization.

Jump straight to the benchmarks?

Introduction

GROUP BY changes the result set cardinality – instead of returning the same number of rows of the input (like a normal SELECT), GROUP BY returns as many rows as there are groups in the data. Consider this (weirdly familiar) example query:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_extendedprice),
    avg(l_quantity)
FROM
    lineitem
GROUP BY
    l_returnflag,
    l_linestatus;
```

GROUP BY is followed by two column names, `l_returnflag` and `l_linestatus`. Those are the columns to compute the groups on, and the resulting table will contain all combinations of the same column that occur in the data. We refer to the columns in the GROUP BY clause as the "grouping columns" and all occurring combinations of values therein as "groups". The SELECT clause contains four (not five) expressions: References to the grouping columns, and two aggregates: the `sum` over `l_extendedprice` and the `avg` over `l_quantity`. We refer to those as the "aggregates". If executed, the result of this query looks something like this:

l_returnflag	l_linestatus	sum(l_extendedprice)	avg(l_quantity)
N	O	114935210409.19	25.5
R	F	56568041380.9	25.51
A	F	56586554400.73	25.52
N	F	1487504710.38	25.52

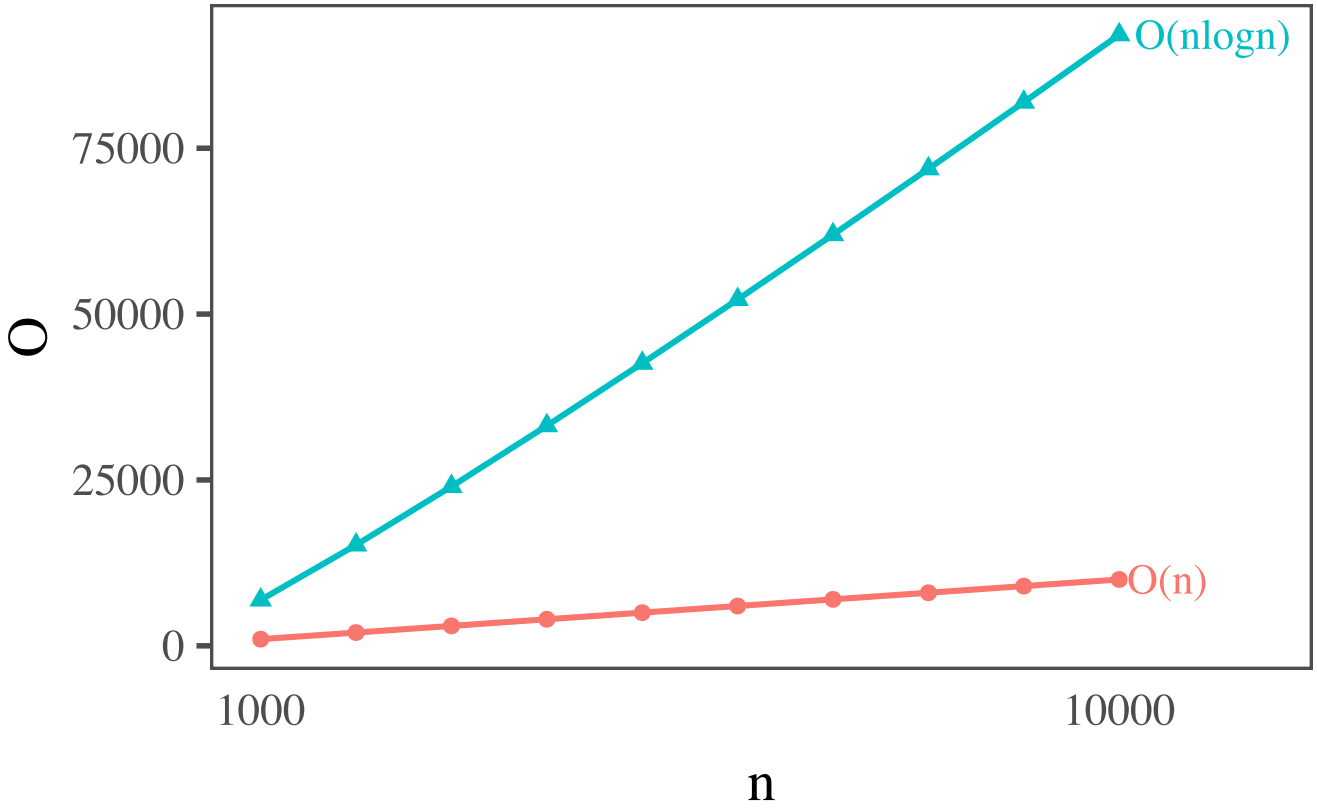
In general, SQL allows only columns that are mentioned in the GROUP BY clause to be part of the SELECT expressions directly, all other columns need to be subject to one of the aggregate functions like `sum`, `avg` etc. There are many more aggregate functions depending on which SQL system you use.

How should a query processing engine compute such an aggregation? There are many design decisions involved, and we will discuss those below and in particular the decisions made by DuckDB. The main issue when computing grouping results is that the groups can occur in the input table in any order. Were the input already sorted on the grouping columns, computing the aggregation would be trivial, as we

could just compare the current values for the grouping columns with the previous ones. If a change occurs, the next group begins and a new aggregation result needs to be computed. Since the sorted case is easy, one straightforward way of computing grouped aggregates is to sort the input table on the grouping columns first, and then use the trivial approach. But sorting the input is unfortunately still a computationally expensive operation despite our best efforts. In general, sorting has a computational complexity of `O(nlogn)` with n being the number of rows sorted.

Hash Tables for Aggregation

A better way is to use a hash table. Hash tables are a foundational data structure in computing that allow us to find entries with a computational complexity of `O(1)`. A full discussion on how hash tables work is far beyond the scope of this post. Below we try to focus on a very basic description and considerations related to aggregate computation.



To add n rows to a hash table we are looking at a complexity of `O(n)`, much, much better than `O(nlogn)` for sorting, especially when n goes into the billions. The figure above illustrates how the complexity develops as the table size increases. Another big advantage is that we do not have to make a sorted copy of the input first, which is going to be just as large as the input. Instead, the hash table will have at most as many entries as there are groups, which can be (and usually are) dramatically fewer than input rows. The overall process is thus this: Scan the input table, and for each row, update the hash table accordingly. Once the input is exhausted, we scan the hash table to provide rows to upstream operators or the query result directly.

Collision Handling

So, hash table it is then! We build a hash table on the input with the groups as keys and the aggregates as the entries. Then, for every input row, we compute a hash of the group values, find the entry in the hash table, and either create or update the aggregate states with the values from the row? Its unfortunately not that simple: Two rows with *different* values for the grouping columns may result in a hash that points to the *same* hash table entry, which would lead to incorrect results.

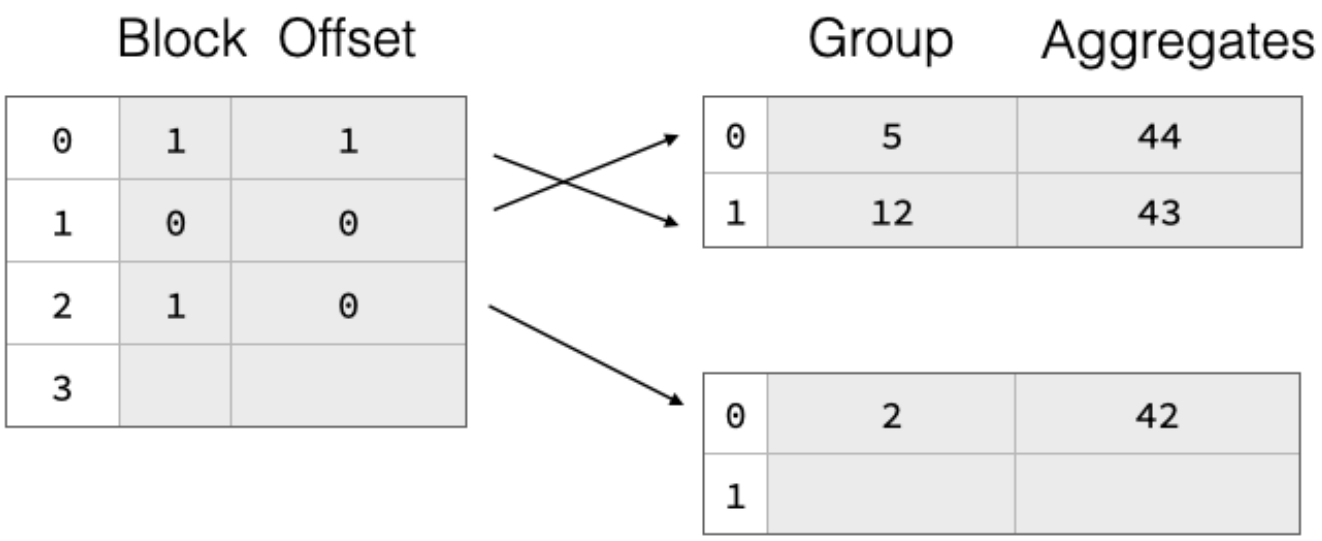
There are two main approaches to work around this problem: “Chaining” or “linear probing”. With chaining, we do not keep the aggregate values in the hash table directly, but rather keep a list of group values and aggregates. If grouping values points to a hash table entry with

an empty list, the new group and the aggregates are simply added. If grouping values point to an existing list, we check for every list entry whether the grouping values match. If so, we update the aggregates for that group. If not, we create a new list entry. In linear probing there are no such lists, but on finding an existing entry, we will compare the grouping values, and if they match we will update the entry. If they do not match, we move one entry down in the hash table and try again. This process finishes when either a matching group entry has been found or an empty hash table entry is found. While theoretically equivalent, computer hardware architecture will favor linear probing because of cache locality. Because linear probing walks the hash table entries *linearly*, the next entry will very likely be in the CPU cache and hence access is faster. Chaining will generally lead to random access and much worse performance on modern hardware architectures. We have therefore adopted linear probing for our aggregate hash table.

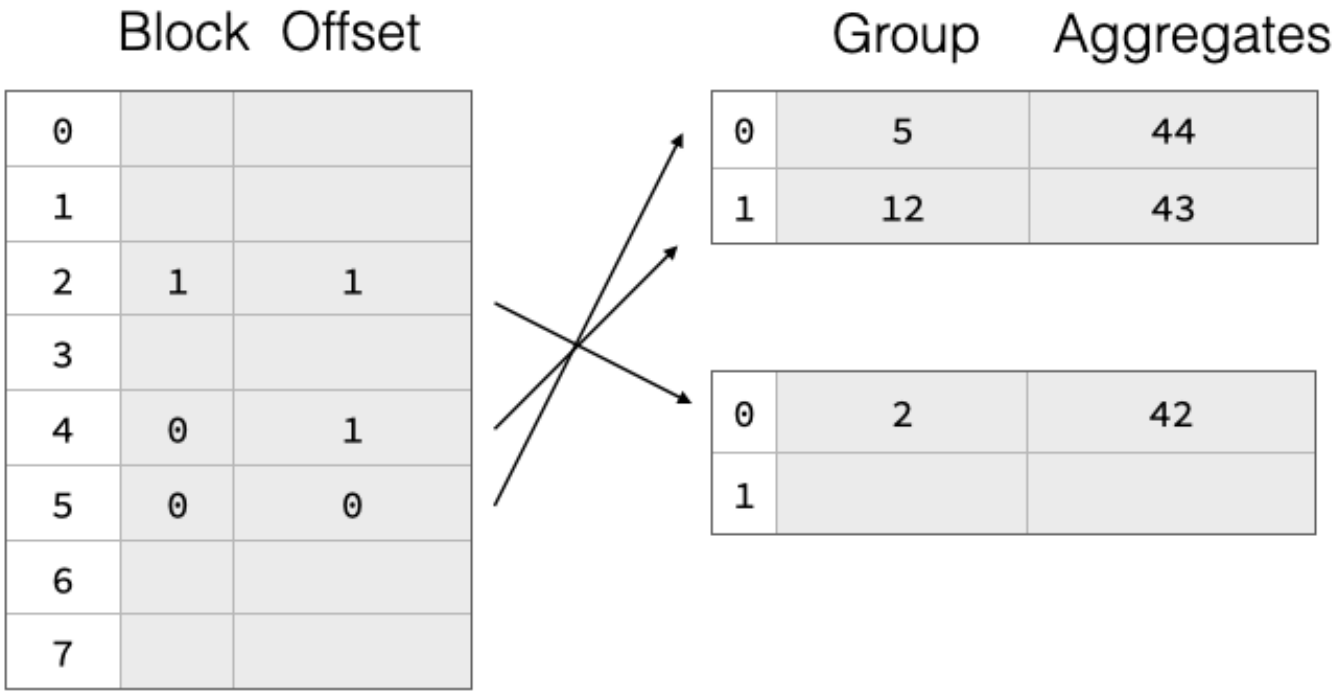
Both chaining and linear probing will degrade in theoretical lookup performance from O(1) to O(n) wrt hash table size if there are too many collisions, i.e., too many groups hashing to the same hash table entry. A common solution to this problem is to resize the hash table once the "fill ratio" exceeds some threshold, e.g., 75% is the default for Java's `HashMap`. This is particularly important as we do not know the amount of groups in the result before starting the aggregation. Neither do we assume to know the amount of rows in the input table. We thus start with a fairly small hash table and resize it once the fill ratio exceeds a threshold. The basic hash table structure is shown in the figure below, the table has four slots 0-4. There are already three groups in the table, with group keys 12, 5 and 2. Each group has aggregate values (e.g., from a `SUM`) of 43 etc.

	Group	Aggregates
0	12	43
1	5	44
2	2	42
3		

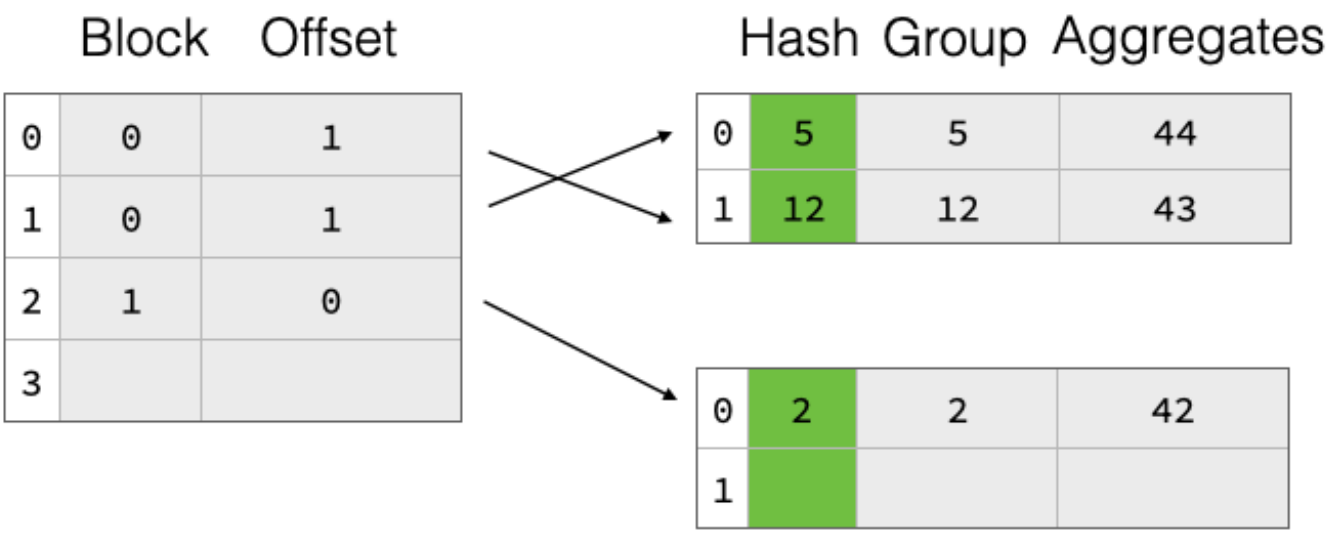
A big challenge with the resize of a partially filled hash table after the resize, all the groups are in the wrong place and we would have to move everything, which will be very expensive.



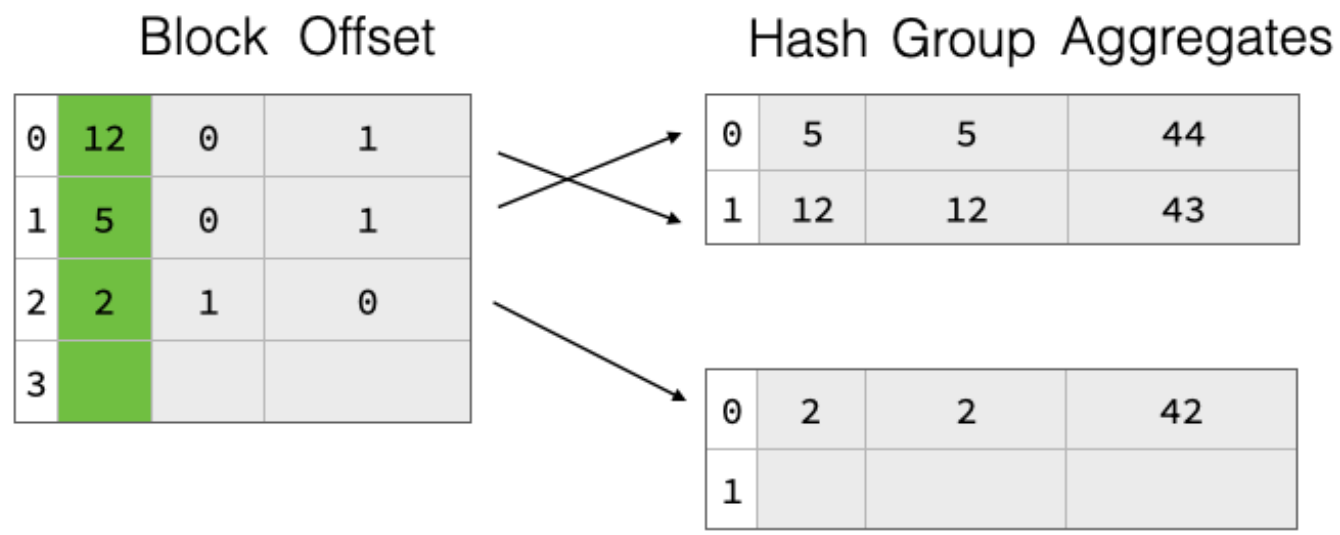
To support resize efficiently, we have implemented a two-part aggregate hash table consisting of a separately-allocated pointer array which points into payload blocks that contain grouping values and aggregate states for each group. The pointers are not actual pointers but symbolic, they refer to a block ID and a row offset within said block. This is shown in the figure above, the hash table entries are split over two payload blocks. On resize, we throw away the pointer array and allocate a bigger one. Then, we read all payload blocks again, hash the group values, and re-insert pointers to them into the new pointer array. The group data thus remains unchanged, which greatly reduces the cost of resizing the hash table. This can be seen in the figure below, where we double the pointer array size but the payload blocks remain unchanged.



The naive two-part hash table design would require a re-hashing of *all* group values on resize, which can be quite expensive especially for string values. To speed this up, we also write the raw hash of the group values to the payload blocks for every group. Then, during resize, we don't have to re-hash the groups but can just read them from the payload blocks, compute the new offset into the pointer array, and insert there.



The two-part hash table has a big drawback when looking up entries: There is no ordering between the pointer array and the group entries in the payload blocks. Hence, following the pointer creates random access in the memory hierarchy. This will lead to unnecessary stalls in the computation. To mitigate this issue, we extend the memory layout of the pointer array to include some (1 or 2) bytes from the group hash in addition to the pointer to the payload value. This way, linear probing can first compare the hash bits in the pointer array with the current group hash and decide whether it's worth following the payload pointer or not. This can potentially continue for every group in the pointer chain. Only when the hash bits match we have to actually follow the pointer and compare the actual groups. This optimization greatly reduces the amount of times the pointer to the payload blocks has to be followed and thereby reduces the amount of random accesses into memory which are directly related to overall performance. It has the nice side-effect of also greatly reducing full group comparisons which can also be expensive, e.g., when aggregating on groups that contain strings.



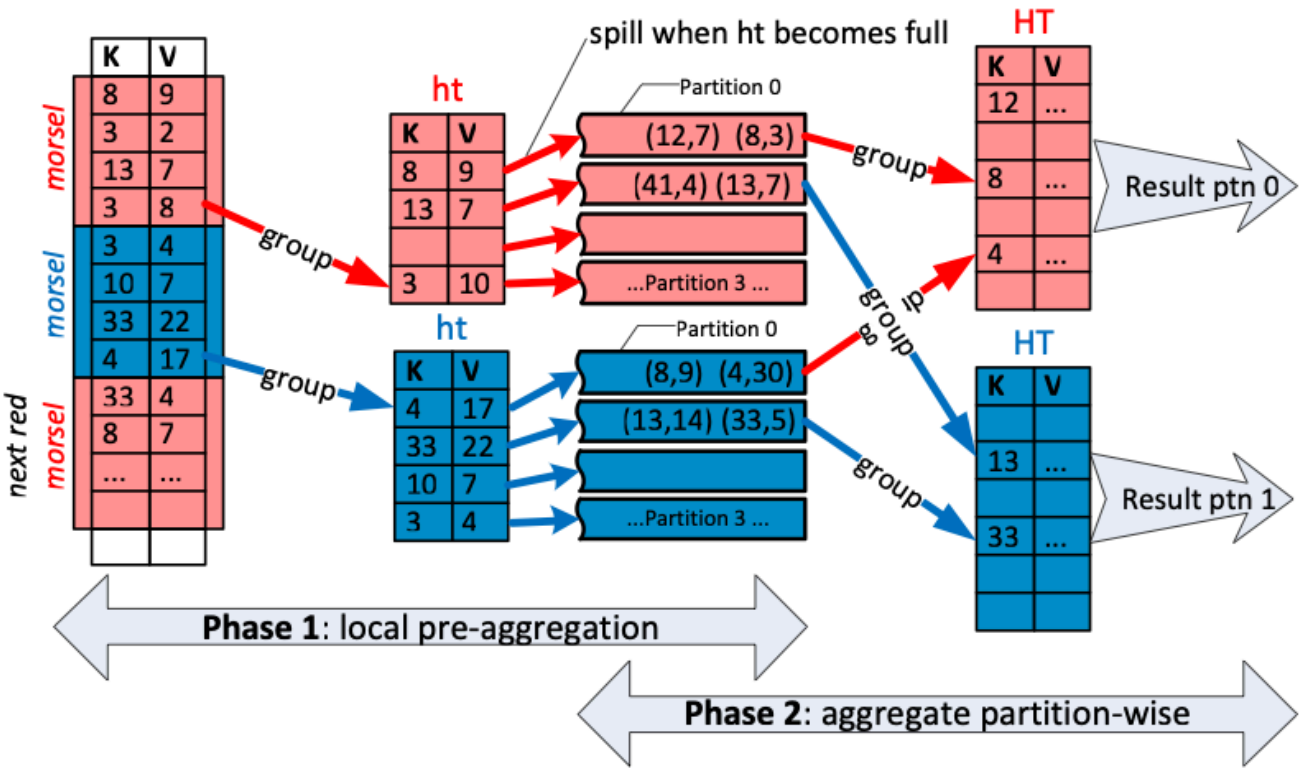
Another (smaller) optimization here concerns the width of the pointer array entries. For small hash tables with few entries, we do not need many bits to encode the payload block offset pointers. DuckDB supports both 4 byte and 8 byte pointer array entries.

For most aggregate queries, the vast majority of query processing time is spent looking up hash table entries, which is why it's worth spending time on optimizing them. If you're curious, the code for all this is in the DuckDB repo, `aggregate_hashtable.cpp`. There is another optimization for when we know that there are only a few distinct groups from column statistics, the perfect hash aggregate, but that's for another post. But we're not done here just yet.

Parallel Aggregation

While we now have an aggregate hash table design that should do fairly well for grouped aggregations, we still have not considered the fact that DuckDB automatically parallelizes all queries to use multiple hardware threads ("CPUs"). How does parallelism work together with hash tables? In general, the answer is unfortunately: "Badly". Hash tables are delicate structures that don't handle parallel modifications well. For example, imagine one thread would want to resize the hash table while another wants to add some new group data to it. Or how should we handle multiple threads inserting new groups at the same time for the same entry? One could use locks to make sure that only one thread at a time is using the table, but this would mostly defeat parallelizing the query. There has been plenty of research into concurrency-friendly hash tables but the short summary is that it's still an open issue.

It is possible to let each thread read data from downstream operators and build individual, local hash tables and merge those together later from a single thread. This works quite nicely if there are few groups like in the example at the top of this post. If there are few groups, a single thread can merge many thread-local hash tables without creating a bottleneck. However, it's entirely possible there are as many groups as there are input rows, for this tends to happen a lot when someone groups on a column that would be a candidate for a primary key, e.g., `observation_number`, `timestamp` etc. What is thus needed is a parallel merge of the parallel hash tables. We adopt a method from Leis et al.: Each thread builds not one, but multiple *partitioned* hash tables based on a radix-partitioning on the group hash.



The key observation here is that if two groups have a different hash value, they cannot possibly be the same. Because of this property, it is possible to use the hash values to create fully independent partitions of the groups without requiring any communication between threads as long as all the threads use the same partitioning scheme (see Phase 1 in the above diagram).

After all the local hash tables have been constructed, we assign individual partitions to each worker thread and merge the hash tables within that partition together (Phase 2). Because the partitions were created using the radix partitioning scheme on the hash, all worker threads can independently merge the hash tables within their respective partitions. The result is correct because each group goes into a single partition and that partition only.

One interesting detail is that we never need to build a final (possibly giant) hash table that holds all the groups because the radix group partitioning ensures that each group is localized to a partition.

There are two additional optimizations for the parallel partitioned hash table strategy: 1) We only start partitioning once a single thread's aggregate hash table exceeds a fixed limit of entries, currently set to 10 000 rows. This is because using a partitioned hash table is not free.

For every row added, we have to figure out which partition it should go into, and we have to merge everything back together at the end. For this reason, we will not start partitioning until the parallelization benefit outweighs the cost. Since the partitioning decision is individual to each thread, it may well be possible only some threads start partitioning. If that is the case, we will need to partition the hash tables of the threads that have not done so before starting merging them. This is a fully thread-local operation however and does not interfere with parallelism. 2) We will stop adding values to a hash table once its pointer array exceeds a certain threshold. Every thread then builds multiple sets of potentially partitioned hash tables. This is because we do not want the pointer array to become arbitrarily large. While this potentially creates duplicate entries for the same group in multiple hash tables, this is not problematic because we merge them all later anyway. This optimization works particularly well on data sets that have many distinct groups, but have group values that are clustered in the input in some manner. For example, when grouping by day in a data set that is ordered on date.

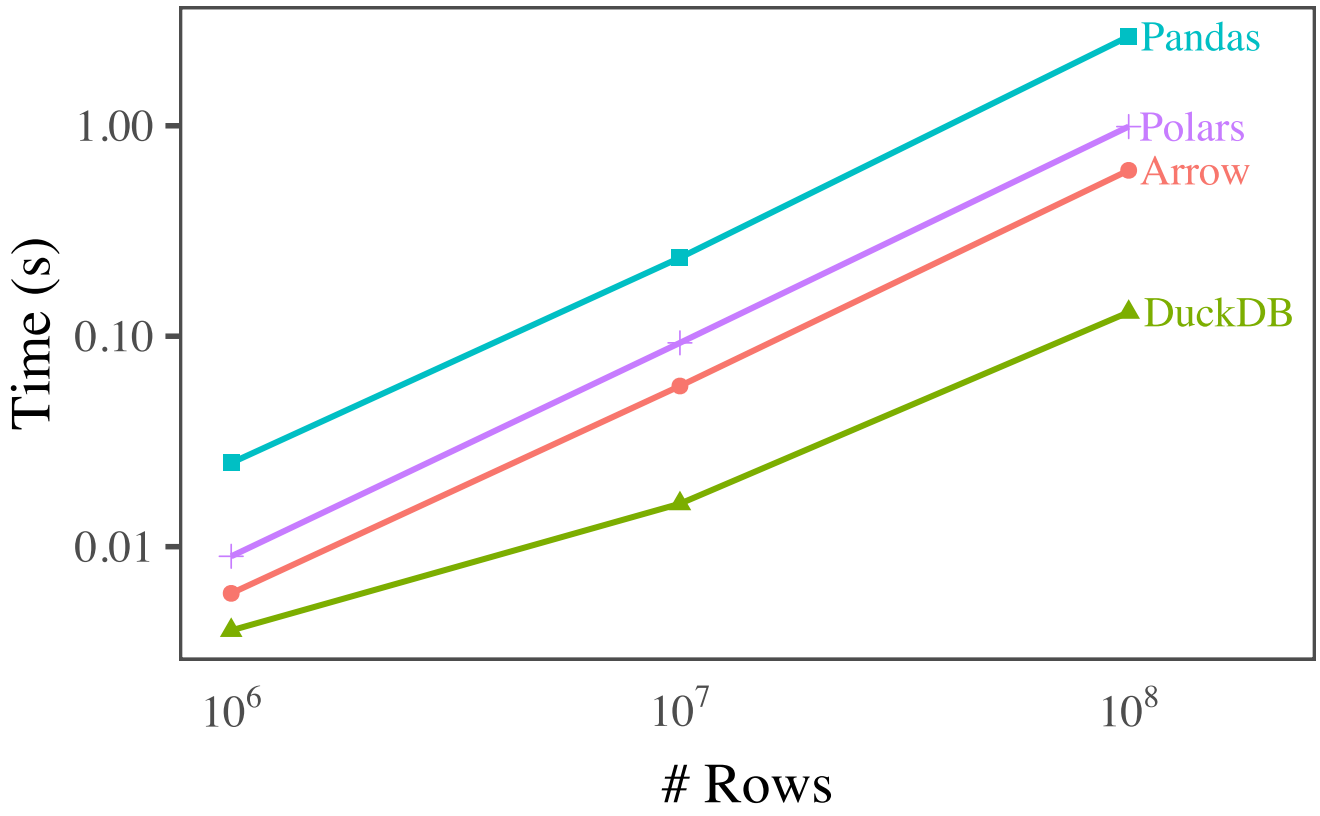
There are some kinds of aggregates which cannot use the parallel and partitioned hash table approach. While it is trivial to parallelize a sum, because the sum of the overall result is just the sum of the individual results, this is fairly impossible for computations like `median`, which DuckDB also supports. Also for this reason, DuckDB also supports `approx_quantile`, which *is* parallelizable.

Experiments

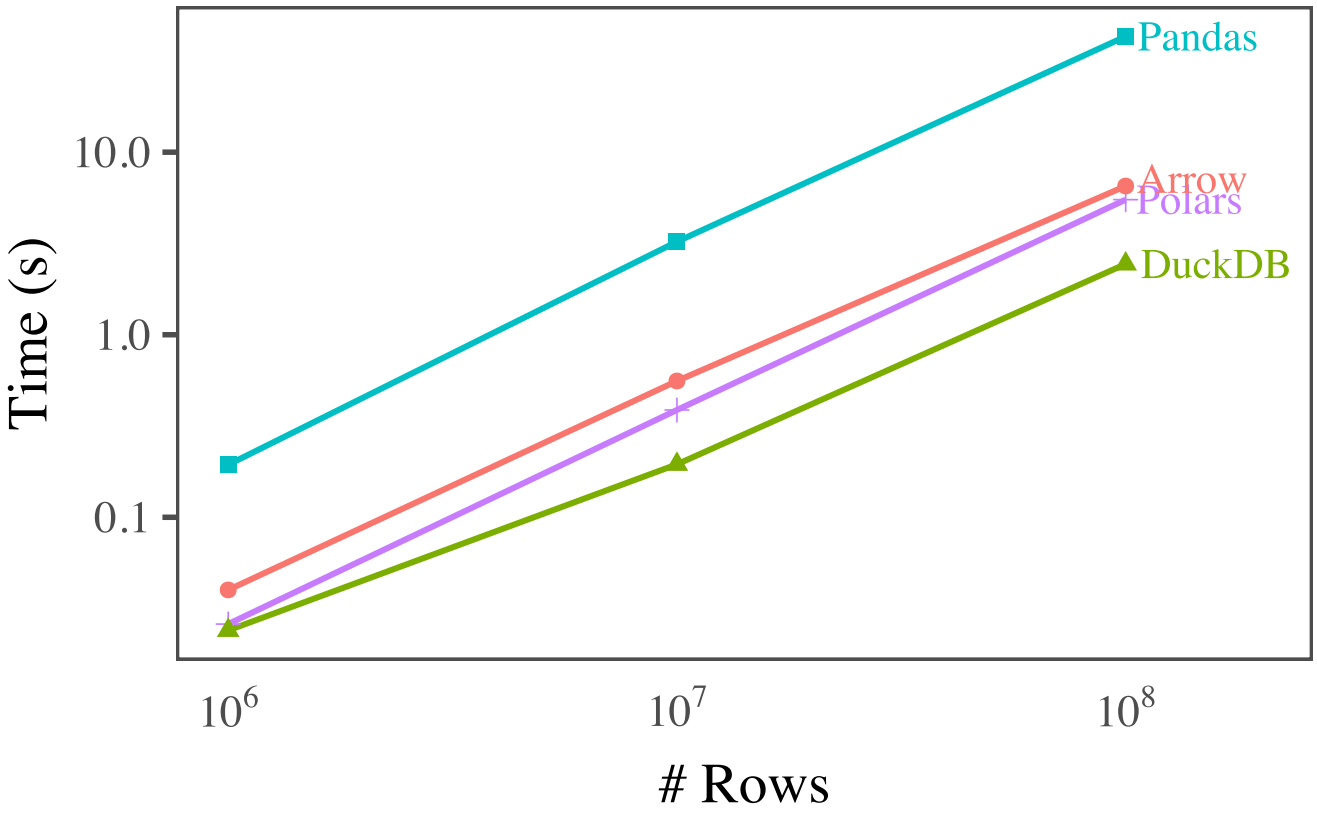
Putting all this together, it's now time for some performance experiments. We will compare DuckDB's aggregation operator as described above with the same operator in various Python data wrangling libraries. The other contenders are Pandas, Polars and Arrow. Those are chosen since they can all execute an aggregation operator on Pandas DataFrames without converting into some other storage format first, just like DuckDB.

For our benchmarks, we generate a synthetic dataset with a pre-defined number of groups over two integer columns and some random integer data to aggregate. The entire dataset is shuffled before the experiments to prevent taking advantage of the clustered nature of the synthetically generated data. For each group, we compute two aggregates, sum of the data column and a simple count. The SQL version of this aggregation would be `SELECT g1, g2, sum(d), count(*) FROM dft GROUP BY g1, g2 LIMIT 1;`. In the experiments below, we vary the dataset size and the amount of groups in them. This should nicely show the scaling behavior of the aggregation.

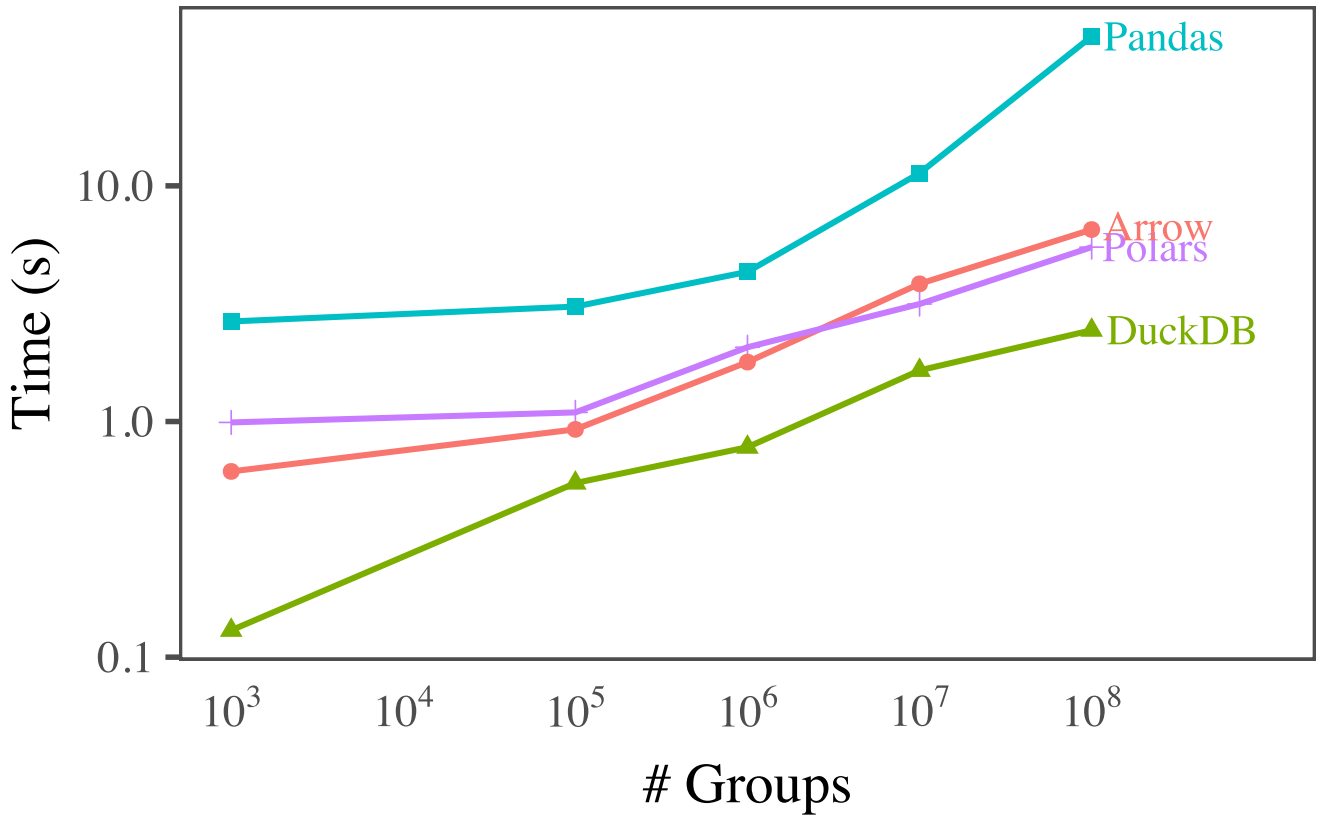
Because we are not interested in measuring the result set materialization time which would be significant for millions of groups, we follow the aggregation with an operator that only retrieves the first row. This does not change the complexity of the aggregation at all, since it needs to collect all data before producing even the first result row, since there might be data in the very last input data row that changes results for the first result. Of course this would be fairly unrealistic in practice, but it should nicely isolate the behavior of the aggregation operator only, since a `head(1)` operation on three columns should be fairly cheap and constant in execution time.



We measure the elapsed wall clock time required to complete each aggregation. To account for minor variation, we repeat each measurement three times and report the median time required. All experiments were run on a 2021 MacBook Pro with a ten-core M1 Max processor and 64 GB of RAM. Our data generation benchmark script is available online and we invite interested readers to re-run the experiment on their machines.



Now let's discuss some results. We start with varying the amount of rows in the table between one million and 100 millions. We repeat the experiment for both a fixed (small) group count of 1000 and when the amount of groups is equal to the amount of rows. Results are plotted as a *log-log plot*, we can see how DuckDB consistently outperforms the other systems, with the single-threaded Pandas being slowest, Polars and Arrow being generally similar.



For the next experiment, we fix the amount of rows at 100M (the largest size we experimented with) and show the full behavior when increasing the group size. We can see again how DuckDB consistently exhibits good scaling behavior when increasing group size, because it can effectively parallelize all phases of aggregation as outlined above. If you are interested in how we generated those plots, the plotting script is available, too.

Conclusion

Data analysis pipelines using mostly aggregation spend the vast majority of their execution time in the aggregate hash table, which is why it is worth spending an ungodly amount of human time optimizing them. We have some ideas for future work on this, for example we would like to extend our work when comparing sorting keys to comparing groups in the aggregate hash table. We also would like to add capabilities of dynamically choosing the amount of partitions a thread uses based on dynamic observation of the created hash table, e.g., if partitions are imbalanced we could use more bits to do so. Another large area of future work is to make our aggregate hash table work with out-of-core operations, where an individual hash table no longer fits in memory, this is particularly problematic when merging. And of course there are always opportunities to fine-tune an aggregation operator, and we are continuously improving DuckDBs aggregation operator.

If you want to work on cutting edge data engineering like this that will be used by thousands of people, consider contributing to DuckDB or join us at DuckLabs in Amsterdam!

Friendlier SQL with DuckDB

Publication date: 2022-05-04

Author: Alex Monahan

TL;DR: DuckDB offers several extensions to the SQL syntax. For a full list of these features, see the Friendly SQL documentation page.

An elegant user experience is a key design goal of DuckDB. This goal guides much of DuckDB's architecture: it is simple to install, seam-

less to integrate with other data structures like Pandas, Arrow, and R Dataframes, and requires no dependencies. Parallelization occurs automatically, and if a computation exceeds available memory, data is gracefully buffered out to disk. And of course, DuckDB's processing speed makes it easier to get more work accomplished.

However, SQL is not famous for being user-friendly. DuckDB aims to change that! DuckDB includes both a Relational API for dataframe-style computation, and a highly Postgres-compatible version of SQL. If you prefer dataframe-style computation, we would love your feedback on our roadmap. If you are a SQL fan, read on to see how DuckDB is bringing together both innovation and pragmatism to make it easier to write SQL in DuckDB than anywhere else. Please reach out on GitHub or Discord and let us know what other features would simplify your SQL workflows. Join us as we teach an old dog new tricks!

SELECT * EXCLUDE

A traditional SQL SELECT query requires that requested columns be explicitly specified, with one notable exception: the * wildcard. SELECT * allows SQL to return all relevant columns. This adds tremendous flexibility, especially when building queries on top of one another. However, we are often interested in *almost* all columns. In DuckDB, simply specify which columns to EXCLUDE:

```
SELECT * EXCLUDE (jar_jar_binks, midichlorians) FROM star_wars;
```

Now we can save time repeatedly typing all columns, improve code readability, and retain flexibility as additional columns are added to underlying tables.

DuckDB's implementation of this concept can even handle exclusions from multiple tables within a single statement:

```
SELECT
    sw.* EXCLUDE (jar_jar_binks, midichlorians),
    ff.* EXCLUDE cancellation
FROM star_wars sw, firefly ff;
```

SELECT * REPLACE

Similarly, we often wish to use all of the columns in a table, aside from a few small adjustments. This would also prevent the use of * and require a list of all columns, including those that remain unedited. In DuckDB, easily apply changes to a small number of columns with REPLACE:

```
SELECT
    * REPLACE (movie_count+3 AS movie_count, show_count*1000 AS show_count)
FROM star_wars_owned_by_disney;
```

This allows views, CTE's, or sub-queries to be built on one another in a highly concise way, while remaining adaptable to new underlying columns.

GROUP BY ALL

A common cause of repetitive and verbose SQL code is the need to specify columns in both the SELECT clause and the GROUP BY clause. In theory this adds flexibility to SQL, but in practice it rarely adds value. DuckDB now offers the GROUP BY we all expected when we first learned SQL – just GROUP BY ALL columns in the SELECT clause that aren't wrapped in an aggregate function!

```
SELECT
    systems,
    planets,
    cities,
    cantinas,
    sum(scum + villainy) AS total_scum_and_villainy
FROM star_wars_locations
GROUP BY ALL;
-- GROUP BY systems, planets, cities, cantinas
```

Now changes to a query can be made in only one place instead of two! Plus this prevents many mistakes where columns are removed from a SELECT list, but not from the GROUP BY, causing duplication.

Not only does this dramatically simplify many queries, it also makes the above EXCLUDE and REPLACE clauses useful in far more situations. Imagine if we wanted to adjust the above query by no longer considering the level of scum and villainy in each specific cantina:

```
SELECT
    * EXCLUDE (cantinas, booths, scum, villainy),
    sum(scum + villainy) AS total_scum_and_villainy
FROM star_wars_locations
GROUP BY ALL;
-- GROUP BY systems, planets, cities
```

Now that is some concise and flexible SQL! How many of your GROUP BY clauses could be re-written this way?

ORDER BY ALL

Another common cause for repetition in SQL is the ORDER BY clause. DuckDB and other RDBMSs have previously tackled this issue by allowing queries to specify the numbers of columns to ORDER BY (For example, ORDER BY 1, 2, 3). However, frequently the goal is to order by all columns in the query from left to right, and maintaining that numeric list when adding or subtracting columns can be error prone. In DuckDB, simply ORDER BY ALL:

```
SELECT
    age,
    sum(civility) AS total_civility
FROM star_wars_universe
GROUP BY ALL
ORDER BY ALL;
-- ORDER BY age, total_civility
```

This is particularly useful when building summaries, as many other client tools automatically sort results in this manner. DuckDB also supports ORDER BY ALL DESC to sort each column in reverse order, and options to specify NULLS FIRST or NULLS LAST.

Column Aliases in WHERE / GROUP BY / HAVING

In many SQL dialects, it is not possible to use an alias defined in a SELECT clause anywhere but in the ORDER BY clause of that statement. This commonly leads to verbose CTE's or subqueries in order to utilize those aliases. In DuckDB, a non-aggregate alias in the SELECT clause can be immediately used in the WHERE and GROUP BY clauses, and aggregate aliases can be used in the HAVING clause, even at the same query depth. No subquery needed!

```
SELECT
    only_imperial_storm_troopers_are_so_precise AS nope,
    turns_out_a_parsec_is_a_distance AS very_speedy,
    sum(mistakes) AS total_oops
FROM oops
WHERE
    nope = 1
GROUP BY
    nope,
    very_speedy
HAVING
    total_oops > 0;
```

Case Insensitivity While Maintaining Case

DuckDB allows queries to be case insensitive, while maintaining the specified case as data flows into and out of the system. This simplifies queries within DuckDB while ensuring compatibility with external libraries.

```
CREATE TABLE mandalorian AS SELECT 1 AS "THIS_IS_THE_WAY";
SELECT this_is_the_way FROM mandalorian;
```

THIS_IS_THE_WAY
1

Friendly Error Messages

Regardless of expertise, and despite DuckDB's best efforts to understand our intentions, we all make mistakes in our SQL queries. Many RDBMSs leave you trying to use the force to detect an error. In DuckDB, if you make a typo on a column or table name, you will receive a helpful suggestion about the most similar name. Not only that, you will receive an arrow that points directly to the offending location within your query.

```
SELECT * FROM star_trek;
```

```
Error: Catalog Error: Table with name star_trek does not exist!
Did you mean "star_wars"?
LINE 1: SELECT * FROM star_trek;
                      ^
```

(Don't worry, ducks and duck-themed databases still love some Trek as well).

DuckDB's suggestions are even context specific. Here, we receive a suggestion to use the most similar column from the table we are querying.

```
SELECT long_ago FROM star_wars;
```

```
Error: Binder Error: Referenced column "long_ago" not found in FROM clause!
Candidate bindings: "star_wars.long_long_ago"
LINE 1: SELECT long_ago FROM star_wars;
               ^
```

String Slicing

Even as SQL fans, we know that SQL can learn a thing or two from newer languages. Instead of using bulky `SUBSTRING` functions, you can slice strings in DuckDB using bracket syntax. As a note, SQL is required to be 1-indexed, so that is a slight difference from other languages (although it keeps DuckDB internally consistent and similar to other DBs).

```
SELECT 'I love you! I know'[:-3] AS nearly_soloed;
```

nearly_soloed
I love you! I k

Simple List and Struct Creation

DuckDB provides nested types to allow more flexible data structures than the purely relational model would allow, while retaining high performance. To make them as easy as possible to use, creating a `LIST` (array) or a `STRUCT` (object) uses simpler syntax than other SQL systems. Data types are automatically inferred.

```
SELECT
    ['A-Wing', 'B-Wing', 'X-Wing', 'Y-Wing'] AS starfighter_list,
    {name: 'Star Destroyer', common_misconceptions: 'Can''t in fact destroy a star'} AS star_destroyer_
facts;
```

List Slicing

Bracket syntax may also be used to slice a `LIST`. Again, note that this is 1-indexed for SQL compatibility.

```
SELECT
    starfighter_list[2:2] AS dont_forget_the_b_wing
FROM (SELECT ['A-Wing', 'B-Wing', 'X-Wing', 'Y-Wing'] AS starfighter_list);
```

dont_forget_the_b_wing
[B-Wing]

Struct Dot Notation

Use convenient dot notation to access the value of a specific key in a DuckDB STRUCT column. If keys contain spaces, double quotes can be used.

```
SELECT
    planet.name,
    planet."Amount of sand"
FROM (SELECT {name: 'Tatooine', 'Amount of sand': 'High'} AS planet);
```

Trailing Commas

Have you ever removed your final column from a SQL `SELECT` and been met with an error, only to find you needed to remove the trailing comma as well!? Never? Ok, Jedi... On a more serious note, this feature is an example of DuckDB's responsiveness to the community. In under 2 days from seeing this issue in a tweet (not even about DuckDB!), this feature was already built, tested, and merged into the primary branch. You can include trailing commas in many places in your query, and we hope this saves you from the most boring but frustrating of errors!

```
SELECT
    x_wing,
    proton_torpedoes,
    --targeting_computer
FROM luke_whats_wrong
GROUP BY
    x_wing,
    proton_torpedoes,
;
```

Function Aliases from Other Databases

For many functions, DuckDB supports multiple names in order to align with other database systems. After all, ducks are pretty versatile – they can fly, swim, and walk! Most commonly, DuckDB supports PostgreSQL function names, but many SQLite names are supported, as well as some from other systems. If you are migrating your workloads to DuckDB and a different function name would be helpful, please reach out – they are very easy to add as long as the behavior is the same! See our functions documentation for details.

```
SELECT
    'Use the Force, Luke'[:13] AS sliced_quote_1,
    substr('I am your father', 1, 4) AS sliced_quote_2,
    substring('Obi-Wan Kenobi, you''re my only hope', 17, 100) AS sliced_quote_3;
```

Auto-Increment Duplicate Column Names

As you are building a query that joins similar tables, you'll often encounter duplicate column names. If the query is the final result, DuckDB will simply return the duplicated column names without modifications. However, if the query is used to create a table, or nested in a subquery or Common Table Expression (where duplicate columns are forbidden by other databases!), DuckDB will automatically assign new names to the repeated columns to make query prototyping easier.

```
SELECT
    *
FROM (
    SELECT
        s1.tie_fighter,
        s2.tie_fighter
    FROM squadron_one s1
    CROSS JOIN squadron_two s2
    ) theyre_coming_in_too_fast;
```

tie_fighter	tie_fighter:1
green_one	green_two

Implicit Type Casts

DuckDB believes in using specific data types for performance, but attempts to automatically cast between types whenever necessary. For example, when joining between an integer and a varchar, DuckDB will automatically cast them to be the same type and complete the join successfully. A `List` or `IN` expression may also be created with a mixture of types, and they will be automatically cast as well. Also, `INTEGER` and `BIGINT` are interchangeable, and thanks to DuckDB's new storage compression, a `BIGINT` usually doesn't even take up any extra space! Now you can store your data as the optimal data type, but use it easily for the best of both!

```
CREATE TABLE sith_count_int AS SELECT 2::INTEGER AS sith_count;
CREATE TABLE sith_count_varchar AS SELECT 2::VARCHAR AS sith_count;

SELECT
    *
FROM sith_count_int s_int
JOIN sith_count_varchar s_char
  ON s_int.sith_count = s_char.sith_count;
```

sith_count	sith_count
2	2

Other Friendly Features

There are many other features of DuckDB that make it easier to analyze data with SQL!

DuckDB makes working with time easier in many ways, including by accepting multiple different syntaxes (from other databases) for the `INTERVAL` data type used to specify a length of time.

DuckDB also implements multiple SQL clauses outside of the traditional core clauses including the `SAMPLE` clause for quickly selecting a random subset of your data and the `QUALIFY` clause that allows filtering of the results of window functions (much like a `HAVING` clause does for aggregates).

The `DISTINCT ON` clause allows DuckDB to select unique combinations of a subset of the columns in a `SELECT` clause, while returning the first row of data for columns not checked for uniqueness.

Ideas for the Future

In addition to what has already been implemented, several other improvements have been suggested. Let us know if one would be particularly useful – we are flexible with our roadmap! If you would like to contribute, we are very open to PRs and you are welcome to reach out on GitHub or Discord ahead of time to talk through a new feature's design.

- Choose columns via regex
  - Decide which columns to select with a pattern rather than specifying columns explicitly
  - ClickHouse supports this with the `COLUMNS` expression
- Incremental column aliases
  - Refer to previously defined aliases in subsequent calculated columns rather than re-specifying the calculations
- Dot operators for JSON types
  - The JSON extension is brand new (see our documentation!) and already implements friendly `->` and `->>` syntax

Thanks for checking out DuckDB! May the Force be with you...

Range Joins in DuckDB

Publication date: 2022-05-27

Author: Richard Wesley

TL;DR: DuckDB has fully parallelized range joins that can efficiently join millions of range predicates.

Range intersection joins are an important operation in areas such as temporal analytics, and occur when two inequality conditions are present in a join predicate. Database implementations often rely on slow `O(N^2)` algorithms that compare every pair of rows for these operations. Instead, DuckDB leverages its fast sorting logic to implement two highly optimized parallel join operators for these kinds of range predicates, resulting in 20-30× faster queries. With these operators, DuckDB can be used effectively in more time-series-oriented use cases.

Introduction

Joining tables row-wise is one of the fundamental and distinguishing operations of the relational model. A join connects two tables horizontally using some Boolean condition called a *predicate*. This sounds straightforward, but how fast the join can be performed depends on the expressions in the predicate. This has lead to the creation of different join algorithms that are optimized for different predicate types.

In this post, we will explain several join algorithms and their capabilities. In particular, we will describe a newly added "range join" algorithm that makes connecting tables on overlapping time intervals or multiple ordering conditions much faster.

Flight Data

No, this part isn't about ducks, but about air group flight statistics from the Battlestar Galactica reboot. We have a couple of tables we will be using: `Pilots`, `Crafts`, `Missions` and `Battles`. Some data was lost when the fleet dispersed, but hopefully this is enough to provide some "real life" examples!

The `Pilots` table contains the pilots and their data that does not change (name, call sign, serial number):

id	callsign	name	serial
1	Apollo	Lee Adama	234567
2	Starbuck	Kara Thrace	462753
3	Boomer	Sharon Valeri	312743
4	Kat	Louanne Katraine	244977
5	Hotdog	Brendan Costanza	304871
6	Husker	William Adama	204971
…	…	…	…

The `Crafts` table contains all the various fighting craft (ignoring the "Ship Of Theseus" problem of recycled parts!):

id	type	tailno
1	Viper	N7242C
2	Viper	2794NC
3	Raptor	312
4	Blackbird	N9999C
…	…	…

The `Missions` table contains all the missions flown by pilots. Missions have a `begin` and `end` time logged with the flight deck. We will use some common pairings (and an unusual mission at the end where Commander Adama flew his old Viper):

pid	cid	begin	end
2	2	3004-05-04 13:22:12	3004-05-04 15:05:49
1	2	3004-05-04 10:00:00	3004-05-04 18:19:12
3	3	3004-05-04 13:33:52	3004-05-05 19:12:21
6	1	3008-03-20 08:14:37	3008-03-20 10:21:15
…	…	…	…

The `Battles` table contains the time window of each battle with the Cylons.

battle	begin	end
Fall of the Colonies	3004-05-04 13:21:45	3004-05-05 02:47:16
Red Moon	3004-05-28 07:55:27	3004-05-28 08:12:19
Tylium Asteroid	3004-06-09 09:00:00	3004-06-09 11:14:29
Resurrection Ship	3004-10-28 22:00:00	3004-10-28 23:47:05
…	…	…

These last two tables (`Missions` and `Battles`) are examples of *state tables*. An object in a state table has a state that runs between two time points. For the battles, the state is just yes/no. For the missions, the state is a pilot/craft combination.

Equality Predicates

The most common type of join involves comparing one or more pairs of expressions for equality, often a primary key and a foreign key. For example, if we want a list of the craft flown by the pilots, we can join the `Pilots` table to the `Craft` table through the `Missions` table:

```
SELECT callsign, count(*), tailno
FROM Pilots p, Missions m, Crafts c
WHERE p.id = m.pid
  AND c.id = m.cid
GROUP BY ALL
ORDER BY 2 DESC;
```

This will give us a table like:

callsign	count(*)	tailno
Starbuck	127	2794NC
Boomer	55	R1234V
Apollo	3	N7242C
Husker	1	N7242C
...	...	...

Range Predicates

The thing to notice in this example is that the conditions joining the tables are equalities connected with ANDs. But relational joins can be defined using *any* Boolean predicate – even ones without equality or AND.

One common operation in temporal databases is intersecting two state tables. Suppose we want to find the time intervals when each pilot was engaged in combat so we can compute combat hours for seniority? Vipers are launched quickly, but not before the battle has started, and there can be malfunctions or pilots may be delayed getting to the flight deck.

```
SELECT callsign, battle,
    greatest(m.begin, b.begin) AS begin,
    least(m.end, b.end) AS end
FROM Pilots p, Missions m, Crafts c, Battles b
WHERE m.begin < b.end
  AND b.begin < m.end
  AND p.id = m.pid
  AND c.id = m.cid;
```

This join creates a set of records containing the call sign and period in combat for each pilot. It handles the case where a pilot returns for a new craft, excludes patrol flights, and even handles the situation when a patrol flight turns into combat! This is because intersecting state tables this way produces a *joint state table* – an important temporal database operation. Here are a few rows from the result:

callsign	battle	begin	end
Starbuck	Fall of the Colonies	3004-05-04 13:22:12	3004-05-04 15:05:49
Apollo	Fall of the Colonies	3004-05-04 13:21:45	3004-05-04 18:19:12
Boomer	Fall of the Colonies	3004-05-04 13:33:52	3004-05-05 02:47:16
...	...	...	...

Apollo was already in flight when the first Cylon attack came, so the query puts his `begin` time for the battle at the start of the battle, not when he launched for the decommissioning flyby. Starbuck and Boomer were scrambled after the battle started, but Boomer did not return until after the battle was effectively over, so her `end` time is moved back to the official end of the battle.

What is important here is that the join condition between the pilot/mission/craft relation and the battle table has no equalities in it. This kind of join is traditionally very expensive to compute, but as we will see, there are ways of speeding it up.

Infinite Time

One common problem with populating state tables is how to represent the open edges. For example, the begin time for the first state might not be known, or the current state may not have ended yet.

Often such values are represented by `NULL`s, but this complicates the intersection query because comparing with `NULL` yields `NULL`. This issue can be worked around by using `coalesce(end, <large timestamp>)`, but that adds a computation to every row, most of

which don't need it. Another approach is to just use `<large timestamp>` directly instead of the `NULL`, which solves the expression computation problem but introduces an arbitrary time value. This value may give strange results when used in computations.

DuckDB provides a third alternative from Postgres that can be used for these situations: infinite time values. Infinite time values will compare as expected, but arithmetic with them will produce `NULL`s or infinities, indicating that the computation is not well defined.

Common Join Algorithms

To see why these joins can be expensive, let's start by looking at the two most common join algorithms.

Hash Joins

Joins with at least one equality condition ANDed to the rest of the conditions are called *equi-joins*. They are usually implemented using a hash table like this:

```
hashes = {}
for b in build:
    hashes[b.pk] = b

result = []
for p in probe:
    result.append((p, hashes[p.fk], ))
```

The expressions from one side (the *build* side) are computed and hashed, then the corresponding expressions from the other side (the *probe* side) are looked up in the hash table and checked for a match.

We can modify this a bit when only *some* of the ANDed conditions are equalities by checking the other conditions once we find the equalities in the hash table. The important point is that we can use a hash table to make the join run time `O(N)`. This modification is a general technique that can be used with any join algorithm which reduces the possible matches.

Nested Loop Joins

Since relational joins can be defined using *any* Boolean predicate – even one without equality or AND, hash joins do not always work. The join algorithm of last resort in these situations is called a *Nested Loop Join* (or NLJ for short), and consists of just comparing every row from the probe side with every row from the build side:

```
result = []
for p in probe:
    for b in build
        if compare(p, b):
            result.append((p, b, ))
```

This is `O(M x N)` in the number of rows, which can be very slow if the tables are large. Even worse, most practical analytic queries (such as the combat hours example above) will not return anything like this many results, so a lot of effort may be wasted. But without an algorithm that is tuned for a kind of predicate, this is what we would have to use.

Range Joins

When we have a range comparison (one of <, <= >, >=) as one of the join conditions, we can take advantage of the ordering it implies by sorting the input relations on some of the join conditions. Sorting is `O(N log N)`, which suggests that this could be faster than an NLJ, and indeed this turns out to be the case.

Piecewise Merge Join

Before the advent of hash joins, databases would often sort the join inputs to find matches. For equi-joins, a repeated binary search would then find the matching values on the build side in `O(M log N)` time. This is called a *Merge Join*, and it runs faster than `O(M x N)`, but not as fast as the `O(N)` time of a hash join. Still, in the case where we have a single range comparison, the binary search lets us find the first match for a probe value. We can then find all the remaining matches by looking after the first one.

If we also sort the probe side, we can even know where to start the search for the next probe value because it will be after where we found the previous value. This is how *Piecewise Merge Join* (PWMJ) works: We sort the build side so that the values are ordered by the predicate (either ASC or DESC), then sort each probe chunk the same way so we can quickly scan through sets of values to find possible matches. This can be significantly faster than NLJ for these types of queries. If there are more join conditions, we can then check the generated matches to make sure all conditions are met because once again the sorting has significantly reduced the number of checks that have to be made.

Inequality Join (IEJoin)

For two range conditions (like the combat pay query), there are even faster algorithms available. We have recently added a new join called IEJoin, which sorts on two predicates to really speed things up.

The way that IEJoin works is to first sort both tables on the values for the first condition and merge the two sort keys into a combined table that tracks the two input tables' row numbers. Next, it sorts the positions in the combined table on the second range condition. It can then quickly scan for matches that pass both conditions. And just like for hash joins, we can check any remaining conditions because we have hopefully significantly reduced the number pairs we have to test.

Walk Through Because the algorithm is a bit tricky, let's step through a small example. (If you are reading the paper, this is a simplified version of the “Union Arrays” optimization from §4.3, but I find this version of the algorithm is much easier to understand than the version in §3.1.) We are going to look at Qp from the paper, which is a self join on the table “West”:

West	t_id	time	cost	cores
s1	404	100	6	4
s2	498	140	11	2
s3	676	80	10	1
s4	742	90	5	4

We are looking for pairs of billing ids where the second id had a shorter time than the first, but a higher cost:

```
SELECT s1.t_id, s2.t_id AS t_id2
FROM west s1, west s2
WHERE s1.time > s2.time
  AND s1.cost < s2.cost;
```

There are two pairs that meet this criteria:

t_id	t_id2
404	676
742	676

(This is an example of another kind of double range query where we are looking for anomalies.)

First, we sort both input tables on the first condition key (`time`). (We sort DESC because we want the values to satisfy the join condition (>) from left to right.)

Because they are sorted the same way, we can merge the condition keys from the sorted tables into a new table called `L1` after marking each row with the table it came from (using negative row numbers to indicate the right table):

L1	s2	s2	s1	s1	s4	s4	s3	s3
time	140	140	100	100	90	90	80	80
cost	11	11	6	6	5	5	10	10
rid	1	-1	2	-2	3	-3	4	-4

The `rid` column lets us map rows in `L1` back to the original table.

Next, we build a second table `L2` with the second condition key (`cost`) and the row positions (P) of `L1` (not the row numbers from the original tables!) We sort `L2` on `cost` (DESC again this time because now we want the join condition to hold from right to left):

L2	s2	s2	s3	s3	s1	s1	s4	s4
cost	11	11	10	10	6	6	5	5
P	0	1	6	7	2	3	4	5

The sorted column of `L1` row positions is called the *permutation array*, and we can use it to find the corresponding position of the `time` value for a given `cost`.

At this point we have two tables (`L1` and `L2`), each sorted on one of the join conditions and pointing back to the tables it was derived from. Moreover, the sort orders have been chosen so that the condition holds from left to right (resp. right to left). Since the conditions are transitive, this means that whenever we have a value that satisfies a condition at a point in the table, it also satisfies it for everything to the right (resp. left)!

With this setup, we can scan `L2` from left to right looking for rows that match both conditions using two indexes:

- `i` iterates across `L2` from left to right;
- `off2` tracks `i` and is used to identify `costs` that satisfy the join condition compared to `i`. (Note that for loose inequalities, this could be to the right of `i`);

We use a bitmap B to track which rows in `L1` that the `L2` scan has already identified as satisfying the `cost` condition compared to the `L2` scan position `i`.

Because we only want matches between one left and one right row, we can skip matches where the `rid`s have different signs. To leverage this observation, we only process values of `i` that are in the left hand table (`rid[P[i]]` is positive), and we only mark bits for rows in the right hand table (`rid[P[i]]` is negative). In this example, the right side rows are the odd numbered values in P (which are conveniently also the odd values of `i`), which makes them easy to track in the example.

For the other rows, here is what happens:

i	off2	cost[i]	cost[off2]	P[i]	rid[P[i]]	B	Result
0	0	11	11	0	1	`00000000`	[]
2	0..2	10	11..10	6	4	`01000000`	[]
4	2..4	6	10..6	2	2	`01000001`	[{s4, s3}]
6	4..6	5	6..5	4	3	`01010001`	[{s1, s3}]

Whenever we find `costs` that satisfy the condition to the left of the scan location (between `off2` and `i`), we use `P[off2]` to mark the bits in B corresponding to those positions in `L1` that reference right side rows. This records that the `cost` condition is satisfied for those

rows. Then whenever we have a position `P[i]` in `L1`, we can scan B to the right to find values that also satisfy the `cost` condition. This works because everything to the right of `P[i]` in `L1` satisfies the `price` condition thanks the sort order of `L1` and the transitivity of the comparison operations.

In more detail:

1. When `i` and `off2` are `0`, the `cost` condition < is not satisfied, so nothing happens;
2. When `i` is `1`, we are looking at a row from the right side of the join, so we skip it and move on;
3. When `i` is 2, we are now looking at a row from the left side, so we bring `off2` forward until the `cost` condition fails, marking B where it succeeds at `P[1] = [1]`;
4. We then scan the `time` values in `L1` right from position `P[i=2] = 6` and find no matches in B;
5. When `i` is 4, we bring `off2` forward again, marking B at `P[3] = [7]`;
6. We then scan `time` from position 2 and find matches at `[6,7]`, one of which (6) is from the right side table;
7. When `i` is 6, we bring `off2` forward again, marking B at `P[5] = [3]`;
8. We then scan `time` from position 4 and again find matches at `[6,7]`;
9. Finally, when `i` runs off the end, we have no new `cost` values, so nothing happens;

What makes this fast is that we only have to check a few bits to find the matches. When we do need to perform comparisons, we can use the fast radix comparison code from our sorting code, which doesn't require special templated versions for every data type. This not only reduces the code size and complexity, it "future-proofs" it against new data types.

Further Details That walk through is a slightly simplified, single threaded version of the actual algorithm. There are a few more details that may be of interest:

- Scanning large, mostly empty bit maps can be slow, so we use the Bloom filter optimization from §4.2.
- The published algorithm assumes that there are no duplicate `L1` values in either table. To handle the general case, we use an exponential search to find the first `L1` value that satisfies the predicate with respect to the current position and scan right from that point;
- We also adapted the distributed Algorithm 3 from §5 by joining pairs of the sorted blocks generated by the sort code on separate threads. This allows us to fully parallelize the operator by first using parallel sorting and then by breaking up the join into independent pieces;
- Breaking up the pieces for parallel execution also allows us to spool join blocks that are not being processed to disk, making the join scalable.

Special Joins

One of the nice things about IEJoin is that it is very general and implements a number of more specialized join types reasonably efficiently. For example, the state intersection query above is an example of an *interval join* where we are looking to join on the intersection of two intervals.

Another specialized join that can be accelerated with `IEJoin` is a *band join*. This can be used to join values that are "close" to each other

```
SELECT r.id, s.id
FROM r, s
WHERE r.value - s.value BETWEEN a AND b;
```

This translates into a double inequality join condition:

```
SELECT r.id, s.id
FROM r, s
WHERE s.value + a <= r.value AND r.value <= s.value + b;
```

which is exactly the type of join expression that IEJoin handles.

Performance

So how fast is the IEJoin? It is so fast that it is difficult to compare it to the previous range join algorithms because the improvements are so large that the other algorithms do not complete in a reasonable amount of time!

Simple Measurements

To give an example, here are the run times for a 100K self join of some employee tax and salary data, where the goal is to find the 1001 pairs of employees where one has a higher salary but the other has a higher tax rate:

```
SELECT
    r.id,
    s.id
FROM Employees r
JOIN Employees s
    ON r.salary < s.salary
    AND r.tax > s.tax;
```

Algorithm	Time (s)
NLJ	21.440
PWMJ	38.698
IEJoin	0.280

Another example is a self join to find 3772 overlapping events in a 30K event table:

```
SELECT
    r.id,
    s.id
FROM events r
JOIN events s
    ON r.start <= s.end
    AND r.end >= s.start
    AND r.id <> s.id;
```

Algorithm	Time (s)
NLJ	6.985
PWMJ	4.780
IEJoin	0.226

In both cases we see performance improvements of 20-100×, which is very helpful when you run a lot of queries like these!

Optimization Measurements

A third example demonstrates the importance of the join pair filtering and exponential search optimizations. The data is a state table of library circulation data from another interval join paper, and the query is a point-in-period temporal query used to generate Figure 4d:

```
SELECT x, count(*) AS y
FROM books,
    (SELECT x FROM range('2013-01-01'::TIMESTAMP, '2014-01-01'::TIMESTAMP, INTERVAL 1 DAY) tbl(x)) dates
WHERE checkout <= x AND x <= return
GROUP BY ALL
ORDER BY 1;
```

The result is a count of the number of books checked out at midnight on each day. These are the runtimes on an 18 core iMac Pro:

Improvement	Time	CPU
Unoptimized	> 30 m	~100%
Filtering	119.76 s	269%
Exponential	11.21 s	571%

The query joins a 35M row table with a 365 row table, so most of the data comes from the left hand side. By avoiding setting bits for the matching rows in the left table, we eliminate almost all `L1` checks. This dramatically reduces the runtime and improved the CPU utilization.

The data also has a large number of rows corresponding to books that were checked out at the start of the year, which all have the same `checkout` date. Searching left linearly in the first block to find the first match for the scan resulted in repeated runs of ~120K comparisons. This caused the runtime to be completely dominated by processing the first block. By reducing the number of comparisons for these rows from an average of ~60K to 16, the runtime dropped by a factor of 10 and the CPU utilization doubled.

Conclusion and Feedback

In this blog post, we explained the new DuckDB range join improvements provided by the new IEJoin operator. This should greatly improve the response time of state table joins and anomaly detection joins. We hope this makes your DuckDB experience even better – and please let us know if you run into any problems! Feel free to reach out on our GitHub page, or our Discord server.

Persistent Storage of Adaptive Radix Trees (ART) in DuckDB

Publication date: 2022-07-27

Author: Pedro Holanda

TL;DR: DuckDB uses Adaptive Radix Tree (ART) Indexes to enforce constraints and to speed up query filters. Up to this point, indexes were not persisted, causing issues like loss of indexing information and high reload times for tables with data constraints. We now persist ART Indexes to disk, drastically diminishing database loading times (up to orders of magnitude), and we no longer lose track of existing indexes. This blog post contains a deep dive into the implementation of ART storage, benchmarks, and future work. Finally, to better understand how our indexes are used, I'm asking you to answer the following survey. It will guide us when defining our future roadmap.

DuckDB uses ART Indexes to keep primary key (PK), foreign key (FK), and unique constraints. They also speed up point-queries, range queries (with high selectivity), and joins. Before the bleeding edge version (or V0.4.1, depending on when you are reading this post), DuckDB did not persist ART indexes on disk. When storing a database file, only the information about existing PKs and FKs would be stored, with all other indexes being transient and non-existing when restarting the database. For PKs and FKs, they would be fully reconstructed when reloading the database, creating the inconvenience of high-loading times.

A lot of scientific work has been published regarding ART Indexes, most notably on synchronization, cache-efficiency, and evaluation. However, up to this point, no public work exists on serializing and buffer managing an ART Tree. Some say that Hyper, the database in Tableau, persists ART indexes, but again, there is no public information on how that is done.

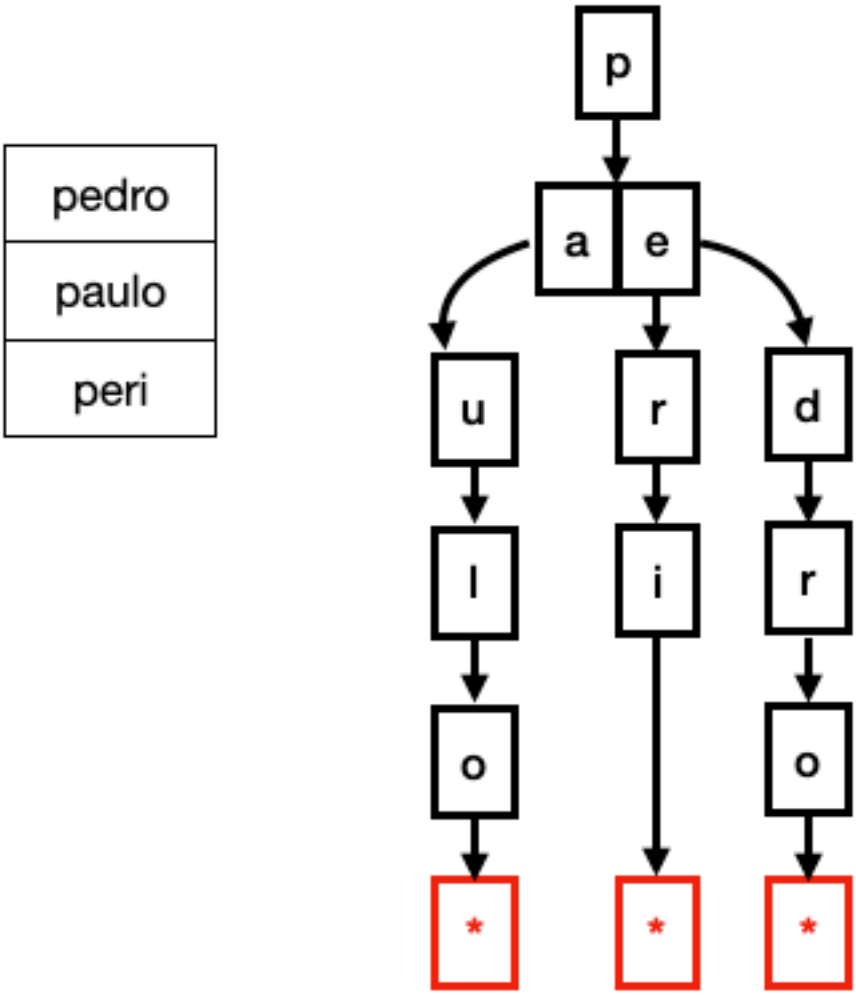
This blog post will describe how DuckDB stores and loads ART indexes. In particular, how the index is lazily loaded (i.e., an ART node is only loaded into memory when necessary). In the ART Index Section, we go through what an ART Index is, how it works, and some examples. In the ART in DuckDB Section, we explain why we decided to use an ART index in DuckDB where it is used and discuss the problems of not persisting ART indexes. In the ART Storage Section, we explain how we serialize and buffer manage ART Indexes in DuckDB. In the Benchmarks Section, we compare DuckDB v0.4.0 (before ART Storage) with the bleeding edge version of DuckDB. We demonstrate the difference in the loading costs of PKs and FKs in both versions and the differences between lazily loading an ART index and accessing a fully loaded ART Index. Finally, in the Road Map section, we discuss the drawbacks of our current implementations and the plans on the list of ART index goodies for the future.

ART Index

Adaptive Radix Trees are, in essence, tries that apply vertical and horizontal compression to create compact index structures.

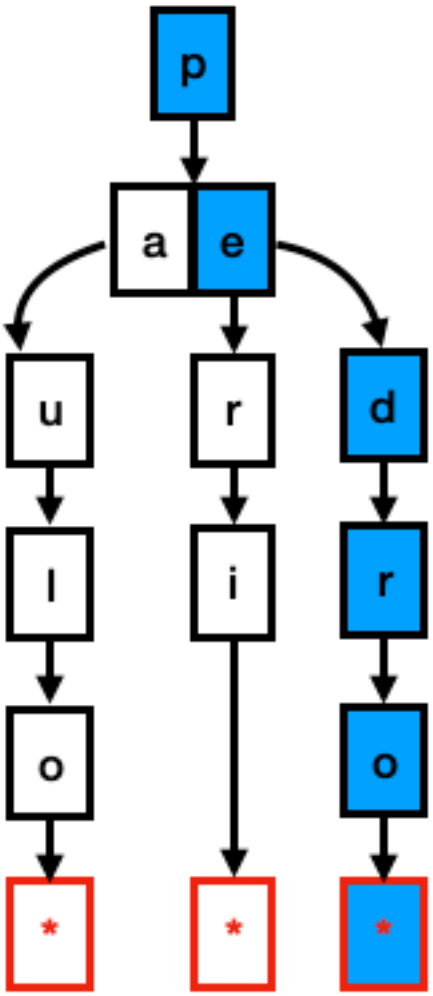
Trie

Tries are tree data structures, where each tree level holds information on part of the dataset. They are commonly exemplified with strings. In the figure below, you can see a trie representation of a table containing the strings “pedro”, “paulo” and “peri”. The root node represents the first character “p” with children “a” (from paulo) and “e” (from pedro and peri), and so on.



To perform lookups on a trie, you must match each character of the key to the current level of the trie. For example, if you search for pedro, you must check the root contains the letter p. If it does, you check if any of its children contains the letter e, up to the point you reach a leaf

node containing the pointer to the tuple that holds this string. (See figure below).



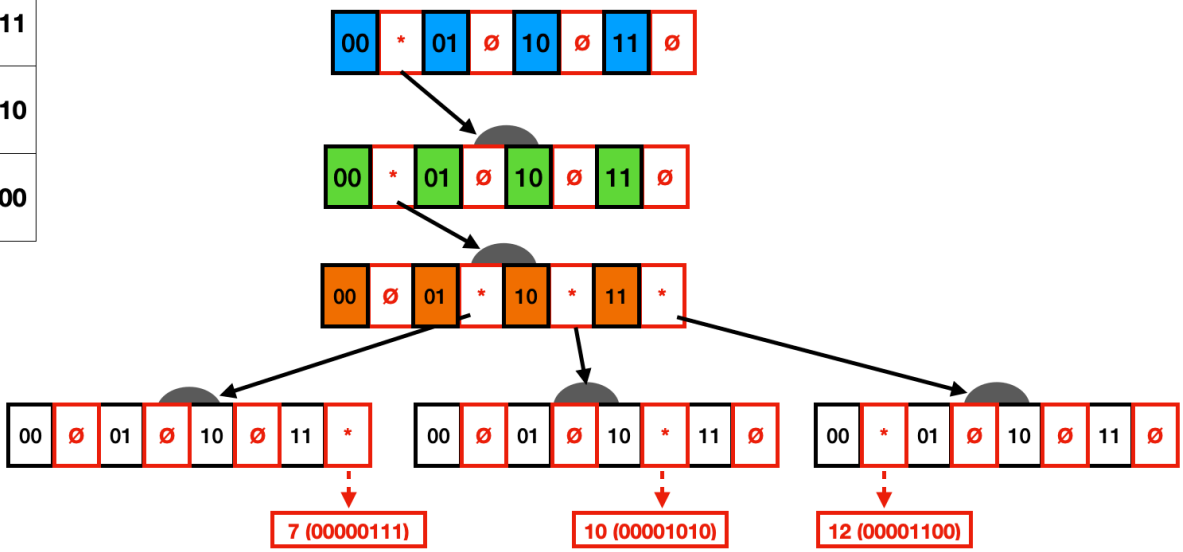
The main advantage of tries is that they have O(k) lookups, meaning that in the worst case, the lookup cost will equal the length of the strings.

In reality, tries can also be used for numeric data types. However, storing them character by character-like strings would be wasteful. Take, for example, the `UBIGINT` data type. In reality, `UBIGINT` is a `uint64_t` which takes 64 bits (i.e., 8 bytes) of space. The maximum value of a `uint64_t` is `18,446,744,073,709,551,615`. Hence if we represented it, like in the example above, we would need 17 levels on the trie. In practice, tries are created on a bit fan-out, which tells how many bits are represented per level of the trie. A `uint64_t` trie with 8-bit fan-out would have a maximum of 8 levels, each representing a byte.

To have more realistic examples, from this point onwards, all depictions in this post will be with bit representations. In DuckDB, the fan-out is always 8 bits. However, for simplicity, the following examples in this blog post will have a fan-out of 2 bits.

In the example below, we have a trie that indexes the values 7, 10, and 12. You can also see the binary representation of each value on the table next to them. Each node consists of the bits 0 and 1, with a pointer next to them. This pointer can either be set (represented by $*$) or null (represented by $\emptyset$). Similar to the string trie we had before, each level of the trie will represent two bits, with the pointer next to these bits pointing to their children. Finally, the leaves point to the actual data.

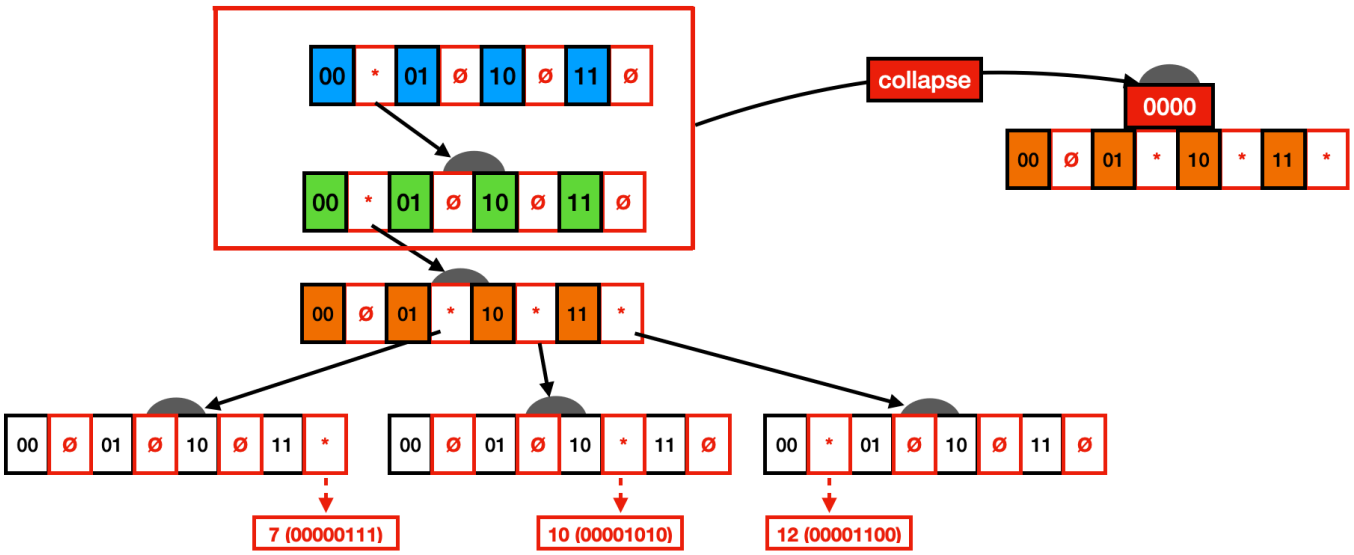
7	00000111
10	00001010
12	00001100



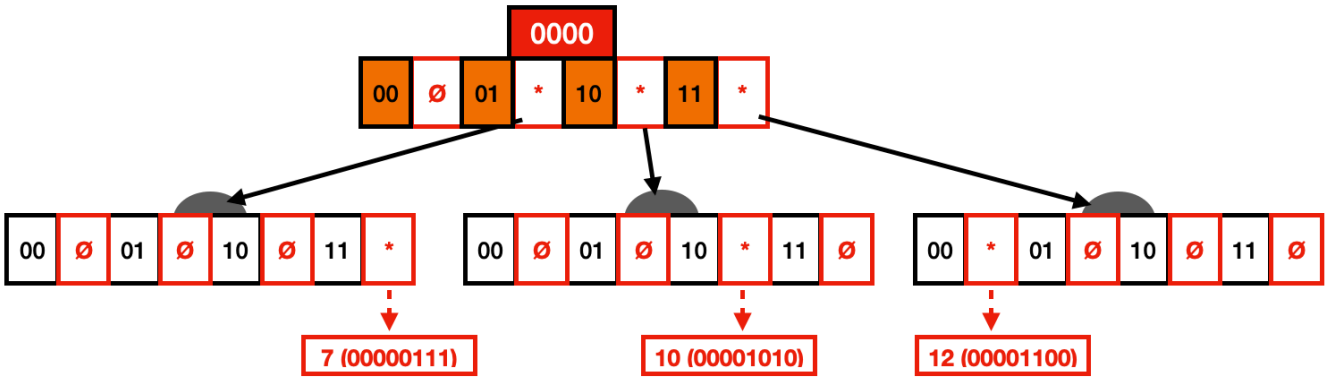
One can quickly notice that this trie representation is wasteful on two different fronts. First, many nodes only have one child (i.e., one path), which could be collapsed by vertical compression (i.e., Radix Tree). Second, many nodes have null pointers, storing space without any information in them, which could be resolved with horizontal compression.

Vertical Compression (i.e., Radix Trees)

The basic idea of vertical compression is that we collapse paths with nodes that only have one child. To support this, nodes store a prefix variable containing the collapsed path to that node. You can see a representation of this in the figure below. For example, one can see that the first four nodes have only one child. These nodes can be collapsed to the third node (i.e., the first one that bifurcates) as a prefix path. When performing lookups, the key must match all values included in the prefix path.



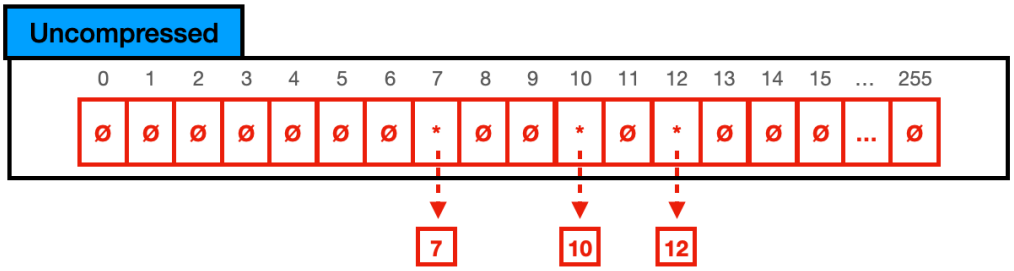
Below you can see the resulting trie after vertical compression. This trie variant is commonly known as a Radix Tree. Although a lot of wasted space has already been saved with this trie variant, we still have many nodes with unset pointers.



Horizontal Compression (i.e., ART)

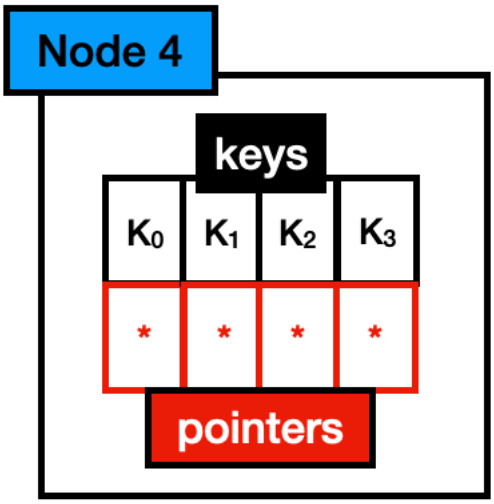
To fully understand the design decisions behind ART indexes, we must first extend the 2-bit fan-out to 8-bits, the commonly found fan-out for database systems.

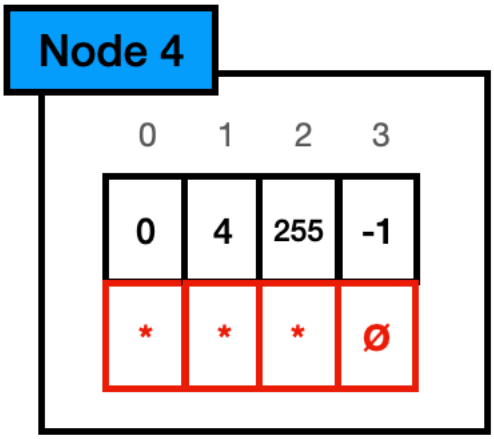
7	00000**111**
10	00001**010**
12	00001**100**



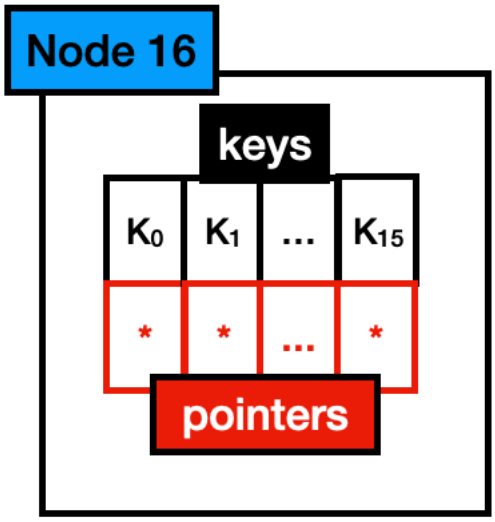
Below you can see the same nodes as before in a trie node of 8 bits. In reality, these nodes will store (2^8) 256 pointers, with the key being the array position of the pointer. In the case depicted by this example, we have a node with (256 pointers * 8 bytes) 2048 byte size while only actually utilizing 24 bytes (3 pointers * 8 bytes), which means that 2016 bytes are entirely wasted. To avoid this situation, ART indexes are composed of 4 different node types that depend on how full the current node is. Below I quickly describe each node with a graphical representation of them. In the graphical representation, I present a conceptual visualization of the node and an example with keys 0,4 and 255.

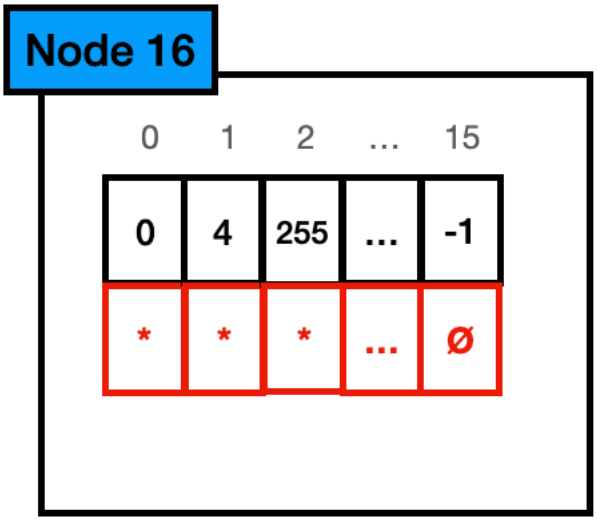
Node 4: Node 4 holds up to 4 different keys. Each key is stored in a one-byte array, with one pointer per key. With its total size being 40 bytes (4*1 + 4*8). Note that the pointer array is aligned with the key array (e.g., key 0 is in position 0 of the keys array, hence its pointer is in position 0 of the pointers array)



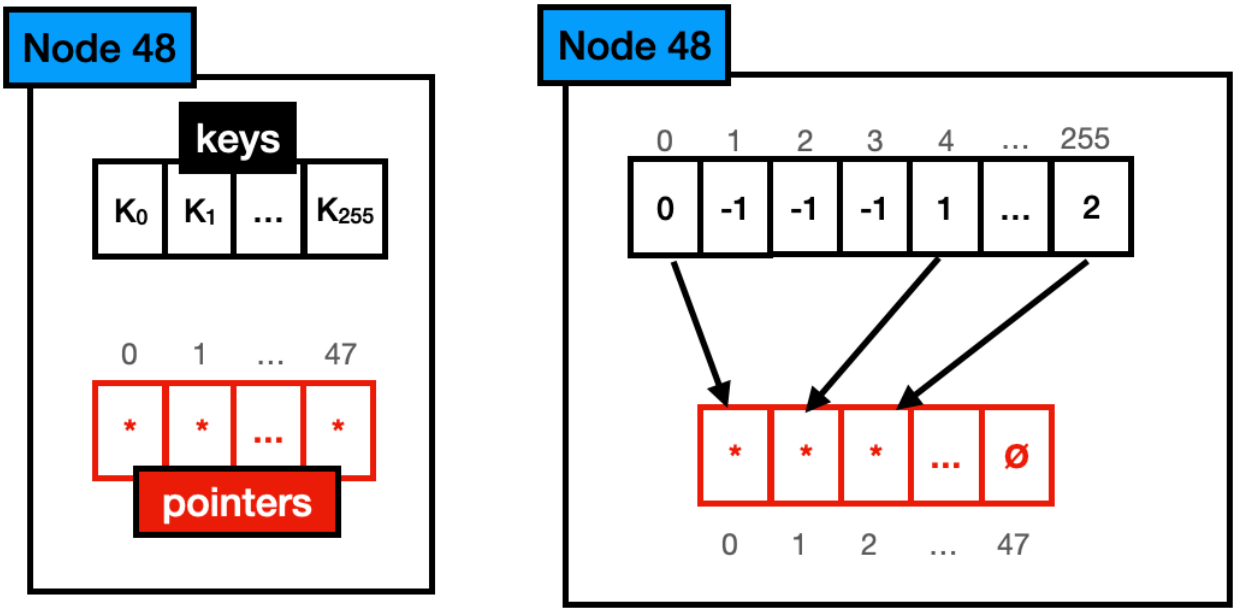


Node 16: Node 16 holds up to 16 different keys. Like node 4, each key is stored in a one-byte array, with one pointer per key. With its total size being 144 bytes (16*1 + 16*8). Like Node 4, the pointer array is aligned with the key array.

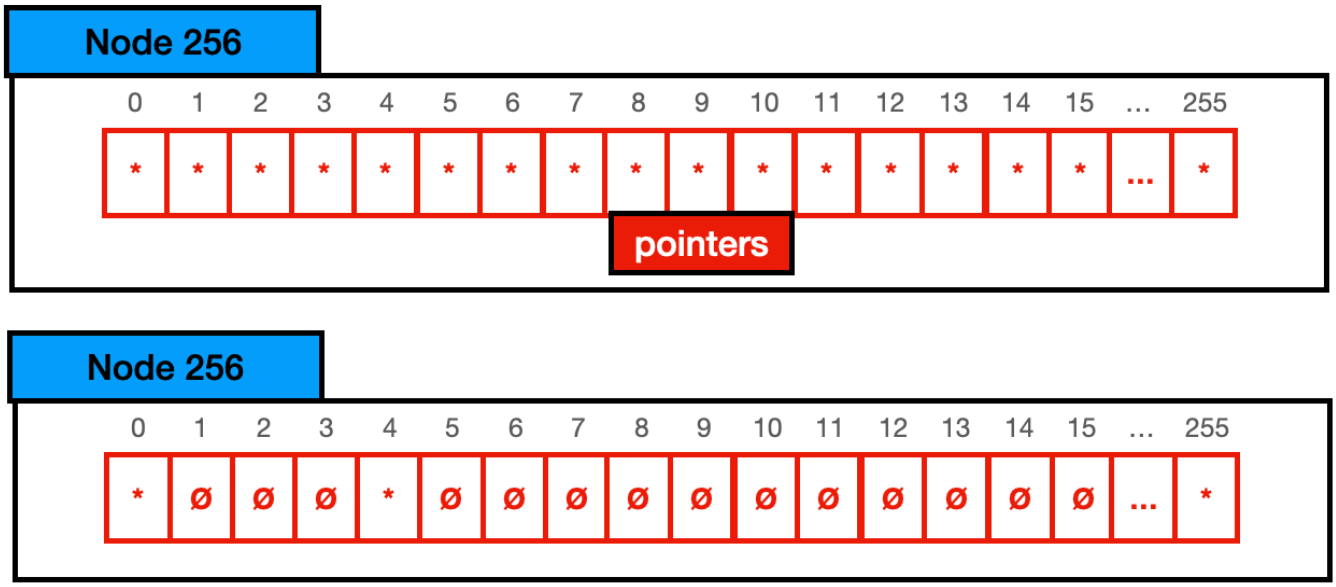




Node 48: Node 48 holds up to 48 different keys. When a key is present in this node, the one-byte array position representing that key will hold an index into the pointer array that points to the child of that key. Its total size is 640 bytes (256*1 + 48*8). Note that the pointer array and the key array are not aligned anymore. The key array points to the position in the pointer array where the pointer of that key is stored (e.g., the key 255 in the key array is set to 2 because the position 2 of the pointer array points to the child pertinent to that key).

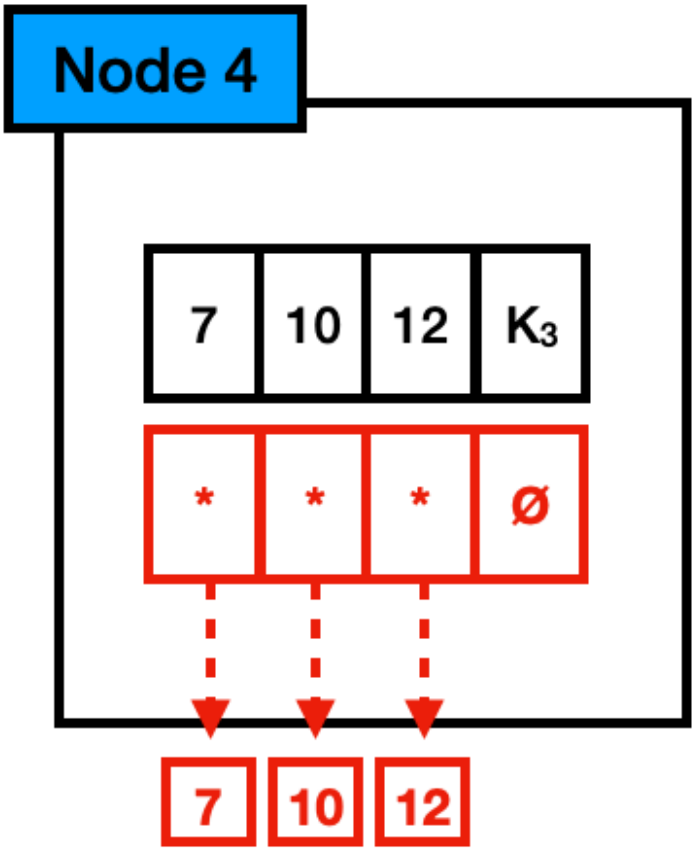


Node 256: Node 256 holds up to 256 different keys, hence all possible values in the distribution. It only has a pointer vector, if the pointer is set, the key exists, and it points to its child. Its total size is 2048 bytes (256 pointers * 8 bytes).



For the example in the previous section, we could use a Node 4 instead of a Node 256 to store the keys, since we only have 3 keys present. Hence it would look like the following:

7	00000111
10	00001010
12	00001100



ART in DuckDB

When considering which index structure to implement in DuckDB, we wanted a structure that could be used to keep PK/FK/Unique constraints while also being able to speed up range queries and Joins. Database systems commonly implement hash tables for constraint checks and B+ trees for range queries. However, we saw in ART Indexes an opportunity to diminish the code complexity by having one data structure for two use cases. The main characteristics that ART Index provides that we take advantage of are:

1. **Compact structure.** Since the ART internal nodes are rather small, they can fit in CPU caches, being a more cache-conscious structure than B+ trees.
2. **Fast point queries.** The worst case for an ART point query is O(k), which is sufficiently fast for constraint checking.
3. **No dramatic regression on insertions.** Many hash table variants must be rebuilt when they reach a certain size. In practice, one insert might cause a significant regression in time, with a query suddenly taking orders of magnitude more time to complete, with no apparent reason for the user. In the ART, inserts might cause node growths (e.g., a Node 4 might grow to a Node 16), but these are inexpensive.
4. **Ability to run range queries.** Although the ART does not run range queries as fast as B+ trees since it must perform tree traversals, where the B+ tree can scan leaf nodes sequentially, it still presents an advantage over hash tables since these types of queries can be done (Some might argue that you can use hash tables for range queries, but meh). This allows us to efficiently use ART for highly selective range queries and index joins.
5. **Maintainability.** Using one structure for both constraint checks and range queries instead of two is more code efficient and maintainable.

What Is It Used For?

As said previously, ART indexes are mainly used in DuckDB on three fronts.

1. Data Constraints. Primary key, Foreign Keys, and Unique constraints are all maintained by an ART Index. When inserting data in a tuple with a constraint, this will effectively try to perform an insertion in the ART index and fail if the tuple already exists.

```
CREATE TABLE integers (i INTEGER PRIMARY KEY);
-- Insert unique values into ART
INSERT INTO integers VALUES (3), (2);
-- Insert conflicting value in ART will fail
INSERT INTO integers VALUES (3);

CREATE TABLE fk_integers (
    j INTEGER,
    FOREIGN KEY (j) REFERENCES integers(i)
 );
-- This insert works normally
INSERT INTO fk_integers VALUES (2), (3);
-- This fails after checking the ART in integers
INSERT INTO fk_integers VALUES (4);
```

2. Range Queries. Highly selective range queries on indexed columns will also use the ART index underneath.

```
CREATE TABLE integers (i INTEGER PRIMARY KEY);
-- Insert unique values into ART
INSERT INTO integers VALUES (3), (2), (1), (8) , (10);
-- Range queries (if highly selective) will also use the ART index
SELECT * FROM integers WHERE i >= 8;
```

3. Joins. Joins with a small number of matches will also utilize existing ART indexes.

```
-- Optionally you can always force index joins with the following pragma
PRAGMA force_index_join;

CREATE TABLE t1 (i INTEGER PRIMARY KEY);
CREATE TABLE t2 (i INTEGER PRIMARY KEY);
-- Insert unique values into ART
INSERT INTO t1 VALUES (3), (2), (1), (8), (10);
INSERT INTO t2 VALUES (3), (2), (1), (8), (10);
-- Joins will also use the ART index
SELECT * FROM t1 INNER JOIN t2 ON (t1.i = t2.i);
```

4. Indexes over expressions. ART indexes can also be used to quickly look up expressions.

```
CREATE TABLE integers (i INTEGER, j INTEGER);

INSERT INTO integers VALUES (1, 1), (2, 2), (3, 3);

-- Creates index over the i + j expression
CREATE INDEX i_index ON integers USING ART((i + j));

-- Uses ART index point query
SELECT i FROM integers WHERE i + j = 2;
```

ART Storage

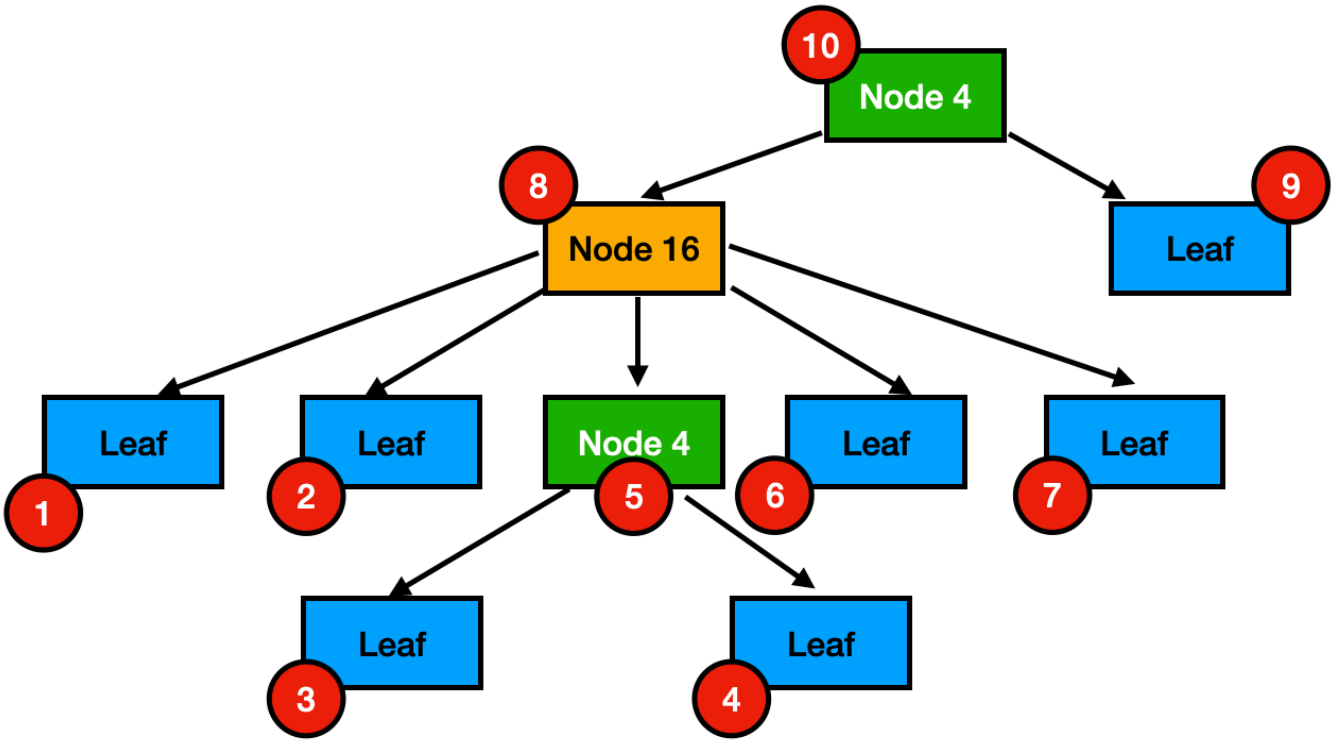
There are two main constraints when storing ART indexes:

1. The index must be stored in an order that allows for lazy-loading. Otherwise, we would have to fully load the index, including nodes that might be unnecessary to queries that would be executed in that session.
2. It must not increase the node size. Otherwise, we diminish the cache-conscious effectiveness of the ART index.

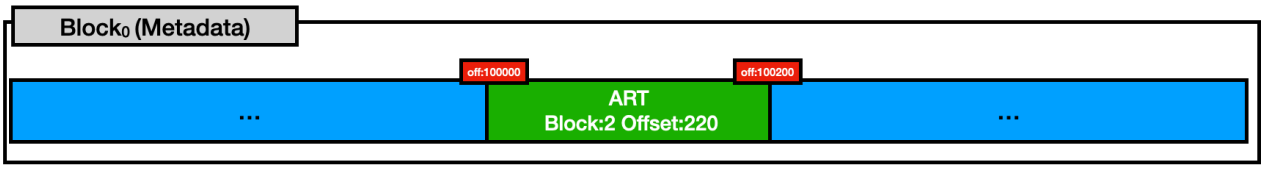
Post-Order Traversal

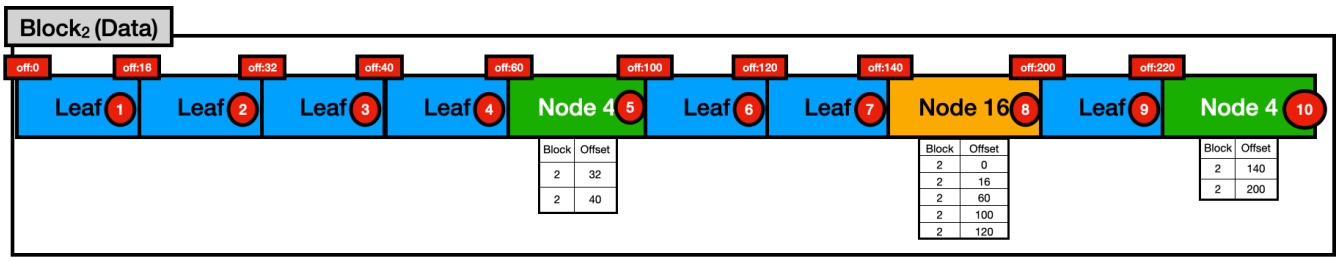
To allow for lazy loading, we must store all children of a node, collect the information of where each child is stored, and then, when storing the actual node, we store the disk information of each of its children. To perform this type of operation, we do a post-order traversal.

The post-order traversal is shown in the figure below. The circles in red represent the numeric order where the nodes will be stored. If we start from the root node (i.e., Node 4 with storage order 10), we must first store both children (i.e., Node 16 with storage order 8 and the Leaf with storage order 9). This goes on recursively for each of its children.



The figure below shows an actual representation of what this would look like in DuckDB's block format. In DuckDB, data is stored in 256 kB contiguous blocks, with some blocks reserved for metadata and some for actual data. Each block is represented by an `id`. To allow for navigation within a block, they are partitioned by byte offsets hence each block contains 256,000 different offsets





In this example, we have `Block 0` that stored some of our database metadata. In particular, between offsets 100,000 and 100,200 we store information pertinent to one ART index. This will store information on the index (e.g., name, constraints, expression) and the `<Block,Offset>` position of its root node.

For example, let's assume we are doing a lookup of the key with `row_ids` stored in the Leaf with storage order 1. We would start by loading the Art Root node on `<Block:2, Offset:220>`, by checking the keys stored in that Node, we would then see we must load the Node 16 at `<Block:2, Offset:140>`, and then finally our Leaf at `<Block:0, Offset:0>`. That means that for this lookup, only these

3 nodes were loaded into memory. Subsequent access to these nodes would only require memory access, while access to different nodes (e.g., Leaf storage order 2) would still result in disk access.

One major problem with implementing this (de)serialization process is that now we not only have to keep information about the memory address of pointers but also if they are already in memory and if not, what's the `<Block,Offset>` position they are stored.

If we stored the Block Id and Offset in new variables, it would dramatically increase the ART node sizes, diminishing its effectiveness as a cache-conscious data structure.

Take Node 256 as an example. The cost of holding 256 pointers is 2048 bytes (256 pointers * 8 bytes). Let's say we decide to store the Block Information on a new array like the following:

```
struct BlockPointer {
    uint32_t block_id;
    uint32_t offset;
}

class Node256 : public Node  {
    // Pointers to the child nodes
    Node* children[256];
    BlockPointer block_info[256];
}
```

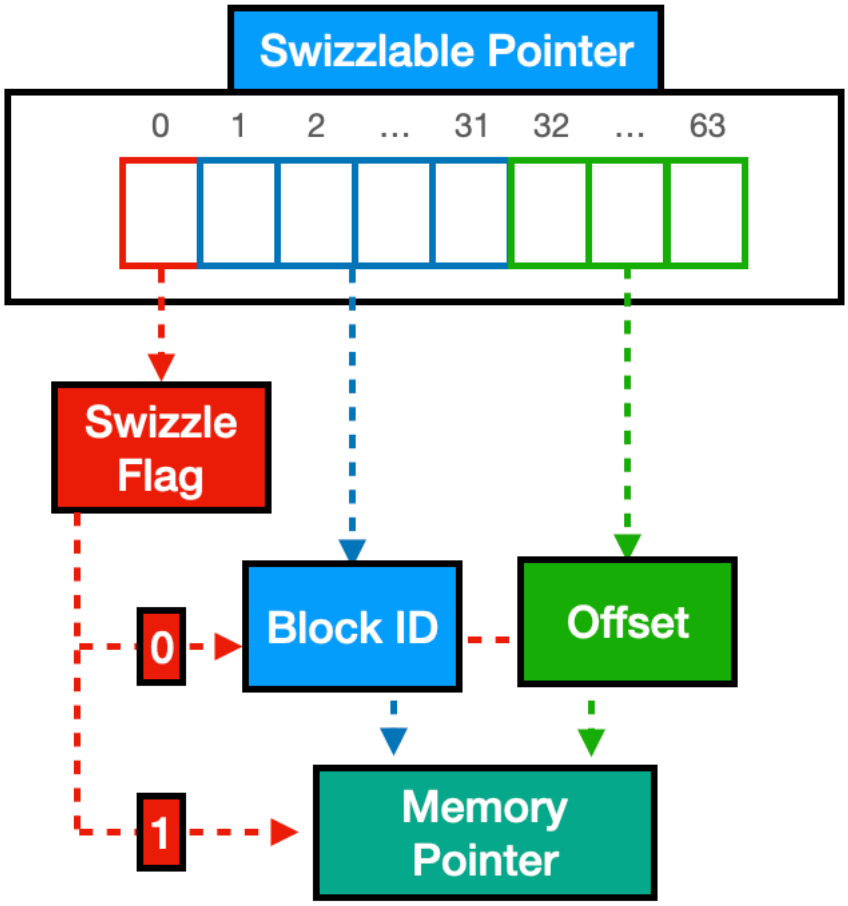
Node 256 would increase 2048 bytes (256 * (4+4)), causing it to double its current size to 4096 bytes.

Pointer Swizzling

To avoid the increase in sizes of ART nodes, we decided to implement Swizzlable Pointers and use them instead of regular pointers.

The idea is that we don't need all 64 bits (i.e., 48 bits give you an address space of 256 terabytes, supporting any of the current architectures, see the related discussion on Stack Overflow and the "64-bit computing" Wikipedia page) in a pointer to point to a memory address. Hence we can use the most significant bit as a flag (i.e., the Swizzle Flag). If the swizzle flag is set, the value in our Swizzlable Pointer is a memory address for the Node. Otherwise, the variable stores the Block Information of where the Node is stored. In the latter case, we use the following 31 bits to store the Block ID and the remaining 32 bits to store the offset.

In the following figure, you can see a visual representation of DuckDB's Swizzlable Pointer.



Benchmarks

To evaluate the benefits and disadvantages of our current storage implementation, we run a benchmark (Available at this Colab Link), where we create a table containing 50,000,000 integral elements with a primary key constraint on top of them.

```
con = duckdb.connect("vault.db")
con.execute("CREATE TABLE integers (x INTEGER PRIMARY KEY)")
con.execute("INSERT INTO integers SELECT * FROM range(50000000)")
```

We run this benchmark on two different versions of DuckDB, one where the index is not stored (i.e., v0.4.0), which means it is always in memory and fully reconstructed at a database restart, and another one where the index is stored (i.e., bleeding-edge version), using the lazy-loading technique described previously.

Storing Time

We first measure the additional cost of serializing our index.

```
cur_time = time.time()
con.close()
print("Storage time: " + str(time.time() - cur_time))
```

Storage Time

Name	Time (s)
Reconstruction	8.99
Storage	18.97

We can see storing the index is 2x more expensive than not storing the index. The reason is that our table consists of one column with 50,000,000 `int32_t` values. However, when storing the ART, we also store 50,000,000 `int64_t` values for their respective `row_ids` in the leaves. This increase in the elements is the main reason for the additional storage cost.

Load Time

We now measure the loading time of restarting our database.

```
cur_time = time.time()
con = duckdb.connect("vault.db")
print("Load time: " + str(time.time() - cur_time))
```

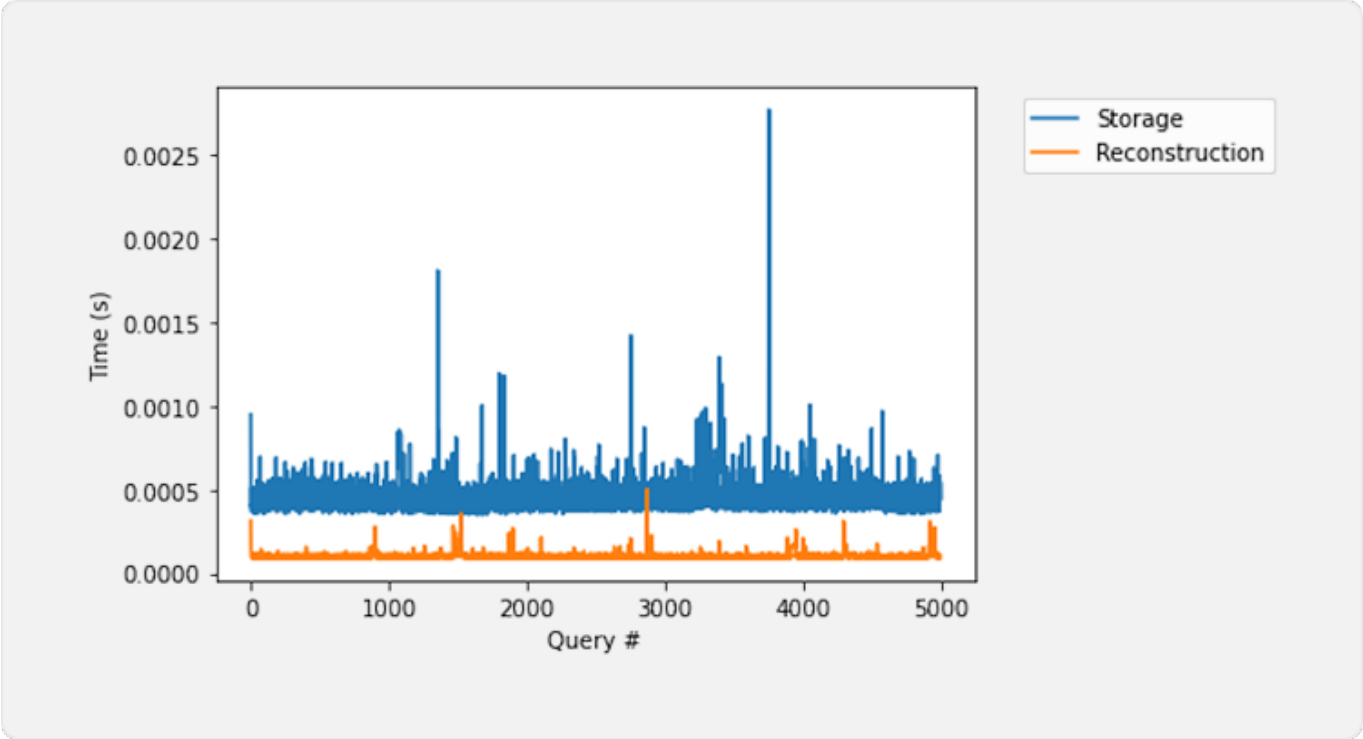
Name	Time (s)
Reconstruction	7.75
Storage	0.06

Here we can see a two-order magnitude difference in the times of loading the database. This difference is basically due to the complete reconstruction of the ART index during loading. In contrast, in the `Storage` version, only the metadata information about the ART index is loaded at this point.

Query Time (Cold)

We now measure the cold query time (i.e., the Database has just been restarted, which means that in the `Storage` version, the index does not exist in memory yet.) of running point queries on our index. We run 5000 point queries, equally spaced on 10000 elements in our distribution. We use this value to always force the point queries to load a large number of unused nodes.

```
times = []
for i in range (0, 50000000, 10000):
  cur_time = time.time()
  con.execute("SELECT x FROM integers WHERE x = " + str(i))
  times.append(time.time() - cur_time)
```

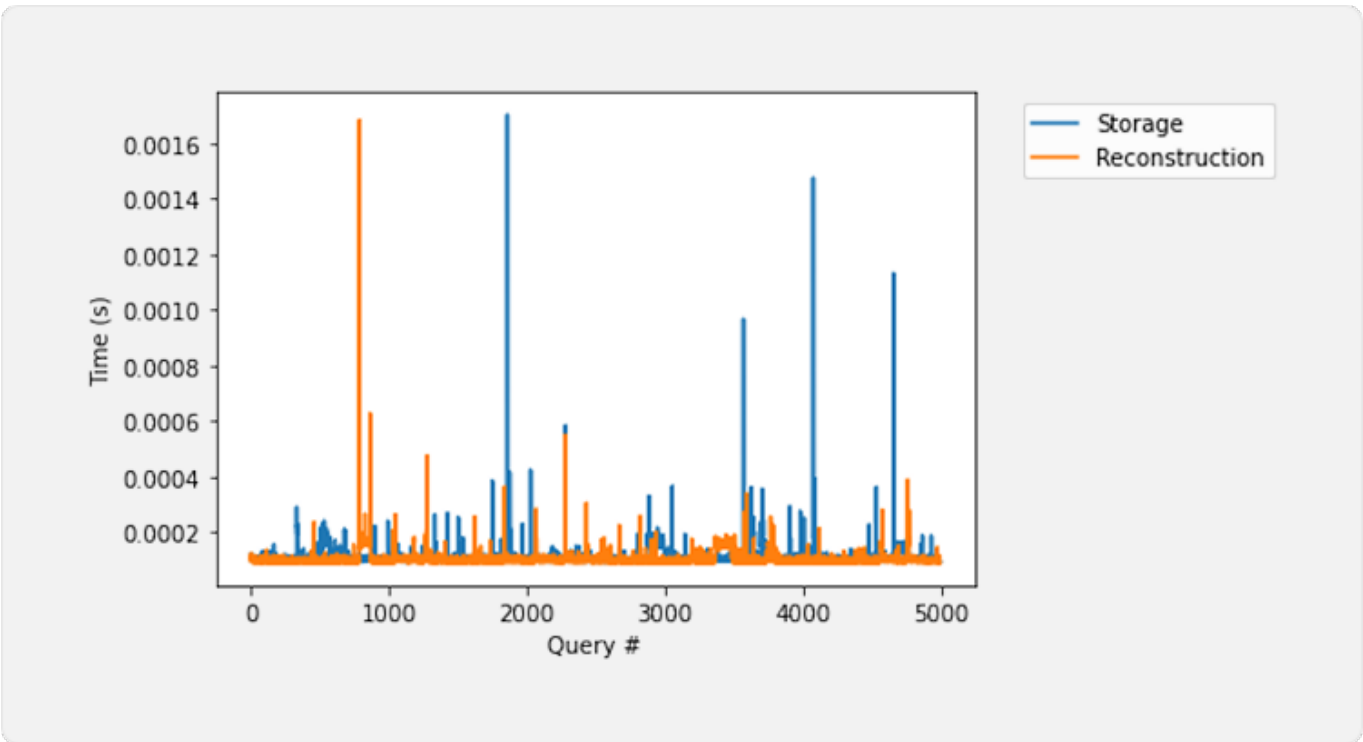


In general, each query is 3x more expensive in the persisted storage format. This is due to two main reasons:

- Creating the nodes. In the storage version, we do create the nodes lazily, which means that, for each node, all parameters must be allocated, and values like keys and prefixes are loaded.
- Block pinning. At every node, we must pin and unpin the blocks where they are stored.

Query Time (Hot)

In this experiment, we execute the same queries as in the previous section.



The times in both versions are comparable since all the nodes in the storage version are already set in memory. In conclusion, when stored indexes are in active use, they present similar performance to fully in-memory indexes.

Future Work

ART index storage has been a long-standing issue in DuckDB, with multiple users claiming it was a missing feature that created an impediment for them to use DuckDB. Although now storing and lazily loading ART indexes is possible, there are many future paths we can still pursue to make the ART-Index more performant. Here I list what I believe are the most important next steps:

1. **Caching pinned blocks.** In our current implementation, blocks are constantly pinned and unpinned, even though blocks can store multiple nodes and are most likely reused continuously through lookups. Smartly caching them will result in drastic savings for queries that trigger node loading.
2. **Bulk loading.** Our ART-Index currently does not support bulk loading. This means that nodes will be constantly resized when creating an index over a column since elements will be inserted one by one. If we bulk-load the data, we can know exactly which Nodes we must create for that dataset, hence avoiding these frequent resizes.
3. **Bulk insertion.** When performing bulk insertion, a similar problem as bulk-loading would happen. A possible solution would be to create a new ART index with Bulk-Loading and then merge it with the existing ART index.
4. **Vectorized lookups/inserts.** DuckDB utilized a vectorized execution engine. However, both our ART lookups and inserts still follow a tuple-at-a-time model.
5. **Updatable index storage.** In our current implementation, ART-Indexes are fully invalidated from disk and stored again. This causes an unnecessary increase in storage time on subsequent storage. Updating nodes directly into disk instead of entirely rewriting the index could drastically decrease future storage costs. In other words, indexes are constantly completely stored at every checkpoint.
6. **Combined pointer/row id leaves.** Our current leaf node format allows for storing values that are repeated over multiple tuples. However, since ART indexes are commonly used to keep unique key constraints (e.g., Primary Keys), and a unique `row_id` fits in the same pointer size space, a lot of space can be saved by using the child pointers to point to the actual `row_id` instead of creating an actual leaf node that only stores one `row_id`.

Roadmap

> Quote It's tough to make predictions, especially about the future.
>
> -- Yogi Berra

ART Indexes are a core part of both constraint enforcement and keeping access speed up in DuckDB. And as depicted in the previous section, there are many distinct paths we can take in our bag of ART goodies, with advantages for completely different use cases.

Querying Postgres Tables Directly from DuckDB

Publication date: 2022-09-30

Author: Hannes Mühleisen

TL;DR: DuckDB can now directly query tables stored in PostgreSQL and speed up complex analytical queries without duplicating data.

Introduction

PostgreSQL is the world's most advanced open source database (self-proclaimed). From its interesting beginnings as an academic DBMS, it has evolved over the past 30 years into a fundamental workhorse of our digital environment.

PostgreSQL is designed for traditional transactional use cases, "OLTP", where rows in tables are created, updated and removed concurrently, and it excels at this. But this design decision makes PostgreSQL far less suitable for analytical use cases, "OLAP", where large chunks of tables are read to create summaries of the stored data. Yet there are many use cases where both transactional and analytical use cases

are important, for example when trying to gain the latest business intelligence insights into transactional data.

There have been some attempts to build database management systems that do well on both workloads, "HTAP", but in general many design decisions between OLTP and OLAP systems are hard trade-offs, making this endeavour difficult. Accepting that one size does not fit all after all, systems are often separated, with the transactional application data living in a purpose-built system like PostgreSQL, and a copy of the data being stored in an entirely different DBMS. Using a purpose-built analytical system speeds up analytical queries by several orders of magnitude.

Unfortunately, maintaining a copy of the data for analytical purposes can be problematic: The copy will immediately be outdated as new transactions are processed, requiring a complex and non-trivial synchronization setup. Storing two copies of the database also will require twice the storage space. For example, OLTP systems like PostgreSQL traditionally use a row-based data representation, and OLAP systems tend to favor a chunked-columnar data representation. You can't have both without maintaining a copy of the data with all the issues that brings with it. Also, the SQL syntaxes between whatever OLAP system you're using and Postgres may differ quite significantly.

But the design space is not as black and white as it seems. For example, the OLAP performance in systems like DuckDB does not only come from a chunked-columnar on-disk data representation. Much of DuckDB's performance comes from its vectorized query processing engine that is custom-tuned for analytical queries. What if DuckDB was able to somehow *read data stored in PostgreSQL*? While it seems daunting, we have embarked on a quest to make just this possible.

To allow for fast and consistent analytical reads of Postgres databases, we designed and implemented the "Postgres Scanner". This scanner leverages the *binary transfer mode* of the Postgres client-server protocol (See the Implementation Section for more details.), allowing us to efficiently transform and use the data directly in DuckDB.

Among other things, DuckDB's design is different from conventional data management systems because DuckDB's query processing engine can run on nearly arbitrary data sources without needing to copy the data into its own storage format. For example, DuckDB can currently directly run queries on Parquet files, CSV files, SQLite files, Pandas, R and Julia data frames as well as Apache Arrow sources. This new extension adds the capability to directly query PostgreSQL tables from DuckDB.

Usage

The Postgres Scanner DuckDB extension source code is available on GitHub, but it is directly installable through DuckDB's new binary extension installation mechanism. To install, just run the following SQL query once:

```
INSTALL postgres_scanner;
```

Then, whenever you want to use the extension, you need to first load it:

```
LOAD postgres_scanner;
```

To make a Postgres database accessible to DuckDB, use the POSTGRES_ATTACH command:

```
CALL postgres_attach('dbname=myshinydb');
```

`postgres_attach` takes a single required string parameter, which is the `libpq` connection string. For example you can pass `'dbname=myshinydb'` to select a different database name. In the simplest case, the parameter is just `''`. There are three additional named parameters to the function:

- `source_schema` the name of a non-standard schema name in Postgres to get tables from. Default is `public`.
- `overwrite` whether we should overwrite existing views in the target schema, default is `false`.
- `filter_pushdown` whether filter predicates that DuckDB derives from the query should be forwarded to Postgres, defaults to `false`. See below for a discussion of what this parameter controls.

The tables in the database are registered as views in DuckDB, you can list them with

```
PRAGMA show_tables;
```

Then you can query those views normally using SQL. Again, no data is being copied, this is just a virtual view on the tables in your Postgres database.

If you prefer to not attach all tables, but just query a single table, that is possible using the POSTGRES_SCAN and POSTGRES_SCAN_PUSHDOWN table-producing functions directly, e.g.

```
SELECT * FROM postgres_scan('dbname=myshinydb', 'public', 'mytable');
SELECT * FROM postgres_scan_pushdown('dbname=myshinydb', 'public', 'mytable');
```

Both functions take three unnamed string parameters, the `libpq` connection string (see above), a Postgres schema name and a table name. The schema name is often `public`. As the name suggests, the variant with "pushdown" in the name will perform selection pushdown as described below.

The Postgres scanner will only be able to read actual tables, views are not supported. However, you can of course recreate such views within DuckDB, the syntax should be exactly the same!

Implementation

From an architectural perspective, the Postgres Scanner is implemented as a plug-in extension for DuckDB that provides a so-called table scan function (`postgres_scan`) in DuckDB. There are many such functions in DuckDB and in extensions, such as the Parquet and CSV readers, Arrow readers etc.

The Postgres Scanner uses the standard `libpq` library, which it statically links in. Ironically, this makes the Postgres Scanner easier to install than the other Postgres clients. However, Postgres' normal client-server protocol is quite slow, so we spent quite some time optimizing this. As a note, DuckDB's SQLite Scanner does not face this issue, as SQLite is also an in-process database.

We actually implemented a prototype direct reader for Postgres' database files, but while performance was great, there is the issue that committed but not yet checkpointed data would not be stored in the heap files yet. In addition, if a checkpoint was currently running, our reader would frequently overtake the checkpointer, causing additional inconsistencies. We abandoned that approach since we want to be able to query an actively used Postgres database and believe that consistency is important. Another architectural option would have been to implement a DuckDB Foreign Data Wrapper (FDW) for Postgres similar to duckdb_fdw but while this could improve the protocol situation, deployment of a postgres extension is quite risky on production servers so we expect few people will be able to do so.

Instead, we use the rarely-used *binary transfer mode* of the Postgres client-server protocol. This format is quite similar to the on-disk representation of Postgres data files and avoids some of the otherwise expensive to-string and from-string conversions. For example, to read a normal `int32` from the protocol message, all we need to do is to swap byte order (`ntohl`).

The Postgres scanner connects to PostgreSQL and issues a query to read a particular table using the binary protocol. In the simplest case (see optimizations below), to read a table called `lineitem`, we internally run the query:

```
COPY (SELECT * FROM lineitem) TO STDOUT (FORMAT binary);
```

This query will start reading the contents of `lineitem` and write them directly to the protocol stream in binary format.

Parallelization

DuckDB supports automatic intra-query parallelization through pipeline parallelism, so we also want to parallelize scans on Postgres tables: Our scan operator opens multiple connections to Postgres, and reads subsets of the table from each. To efficiently split up reading the table, we use Postgres' rather obscure *TID Scan* (Tuple ID) operator, which allows a query to surgically read a specified range of tuple IDs from a table. The Tuple IDs have the form `(page, tuple)`. We parallelize our scan of a Postgres table based on database page ranges expressed in TIDs. Each scan task reads 1000 pages currently. For example, to read a table consisting of 2500 pages, we would start three scan tasks with TID ranges `[(0,0),(999,0)]`, `[(1000,0),(1999,0)]` and `[(2000,0),(UINT32_MAX,0)]`. Having an open bound for the last range is important because the number of pages (`relpages`) in a table in the `pg_class` table is merely an estimate. For a given page range (P_MIN, P_MAX), our query from above is thus extended to look like this:

```
COPY (
   SELECT *
   FROM lineitem
   WHERE ctid BETWEEN '(P_MIN,0)'::tid AND '(P_MAX,0)'::tid
) TO STDOUT (FORMAT binary);
```

This way, we can efficiently scan the table in parallel while not relying on the schema in any way. Because page size is fixed in Postgres, this also has the added bonus of equalizing the effort to read a subset of the page independent of the number of columns in each row.

"But wait!", you will say, according to the documentation the tuple ID is not stable and may be changed by operations such as `VACUUM ALL`. How can you use it for synchronizing parallel scans? This is true, and could be problematic, but we found a solution:

Transactional Synchronization

Of course a transactional database such as Postgres is expected to run transactions while we run our table scans for analytical purposes. Therefore we need to address concurrent changes to the table we are scanning in parallel. We solve this by first creating a new read-only transaction in DuckDB's bind phase, where query planning happens. We leave this transaction running until we are completely done reading the table. We use yet another little-known Postgres feature, `pg_export_snapshot()`, which allows us to get the current transaction context in one connection, and then import it into our parallel read connections using `SET TRANSACTION SNAPSHOT ...`. This way, all connections related to one single table scan will see the table state exactly as it appeared at the very beginning of our scan throughout the potentially lengthy read process.

Projection and Selection Push-Down

DuckDB's query optimizer moves selections (filters on rows) and projections (removal of unused columns) as low as possible in the query plan (push down), and even instructs the lowermost scan operators to perform those operations if they support them. For the Postgres scanner, we have implemented both push down variants. Projections are rather straightforward – we can immediately instruct Postgres to only retrieve the columns the query is using. This of course also reduces the number of bytes that need to be transferred, which speeds up queries. For selections, we construct a SQL filter expression from the pushed down filters. For example, if we run a query like `SELECT l_returnflag, l_linestatus FROM lineitem WHERE l_shipdate < '1998-09-02'` through the Postgres scanner, it would run the following queries:

```
COPY (
  SELECT
    "l_returnflag",
    "l_linestatus"
  FROM "public"."lineitem"
  WHERE
    ctid BETWEEN '(0,0)'::tid AND '(1000,0)'::tid AND
    ("l_shipdate" < '1998-09-02' AND "l_shipdate" IS NOT NULL)
  ) TO STDOUT (FORMAT BINARY);
-- and so on
```

As you can see, the projection and selection pushdown has expanded the queries ran against Postgres accordingly. Using the selection push-down is optional. There may be cases where running a filter in Postgres is actually slower than transferring the data and running the filter in DuckDB, for example when filters are not very selective (many rows match).

Performance

To investigate the performance of the Postgres Scanner, we ran the well-known TPC-H benchmark on DuckDB using its internal storage format, on Postgres also using its internal format and with DuckDB reading from Postgres using the new Postgres Scanner. We used DuckDB 0.5.1 and Postgres 14.5, all experiments were run on a MacBook Pro with an M1 Max CPU. The experiment script is available. We run "scale factor" 1 of TPCH, creating a dataset of roughly 1 GB with ca. 6 M rows in the biggest table, `lineitem`. Each of the 22 TPC-H benchmark queries was run 5 times, and we report the median run time in seconds. The time breakdown is given in the following table.

query	duckdb	duckdb/postgres	postgres
1	0.03	0.74	1.12
2	0.01	0.20	0.18
3	0.02	0.55	0.21
4	0.03	0.52	0.11
5	0.02	0.70	0.13
6	0.01	0.24	0.21
7	0.04	0.56	0.20

query	duckdb	duckdb/postgres	postgres
8	0.02	0.74	0.18
9	0.05	1.34	0.61
10	0.04	0.41	0.35
11	0.01	0.15	0.07
12	0.01	0.27	0.36
13	0.04	0.18	0.32
14	0.01	0.19	0.21
15	0.03	0.36	0.46
16	0.03	0.09	0.12
17	0.05	0.75	> 60.00
18	0.08	0.97	1.05
19	0.03	0.32	0.31
20	0.05	0.37	> 60.00
21	0.09	1.53	0.35
22	0.03	0.15	0.15

Stock Postgres is not able to finish queries 17 and 20 within a one-minute timeout because of correlated subqueries containing a query on the lineitem table. For the other queries, we can see that DuckDB with the Postgres Scanner not only finished all queries, it also was faster than stock Postgres on roughly half of them, which is astonishing given that DuckDB has to read its input data from Postgres through the client/server protocol as described above. Of course, stock DuckDB is still 10× faster with its own storage, but as discussed at the very beginning of this post this requires the data to be imported there first.

Other Use Cases

The Postgres Scanner can also be used to combine live Postgres data with pre-cached data in creative ways. This is especially effective when dealing with an append only table, but could also be used if a modified date column is present. Consider the following SQL template:

```
INSERT INTO my_table_duckdb_cache
SELECT * FROM postgres_scan('dbname=myshinydb', 'public', 'my_table')
WHERE incrementing_id_column > (SELECT max(incrementing_id_column) FROM my_table_duckdb_cache);

SELECT * FROM my_table_duckdb_cache;
```

This provides faster query performance with fully up to date query results, at the cost of data duplication. It also avoids complex data replication technologies.

DuckDB has built-in support to write query results to Parquet files. The Postgres scanner provides a rather simple way to write Postgres tables to Parquet files, it can even directly write to S3 if desired. For example,

```
COPY (SELECT * FROM postgres_scan('dbname=myshinydb', 'public', 'lineitem')) TO 'lineitem.parquet' (FORMAT
parquet);
```

Conclusion

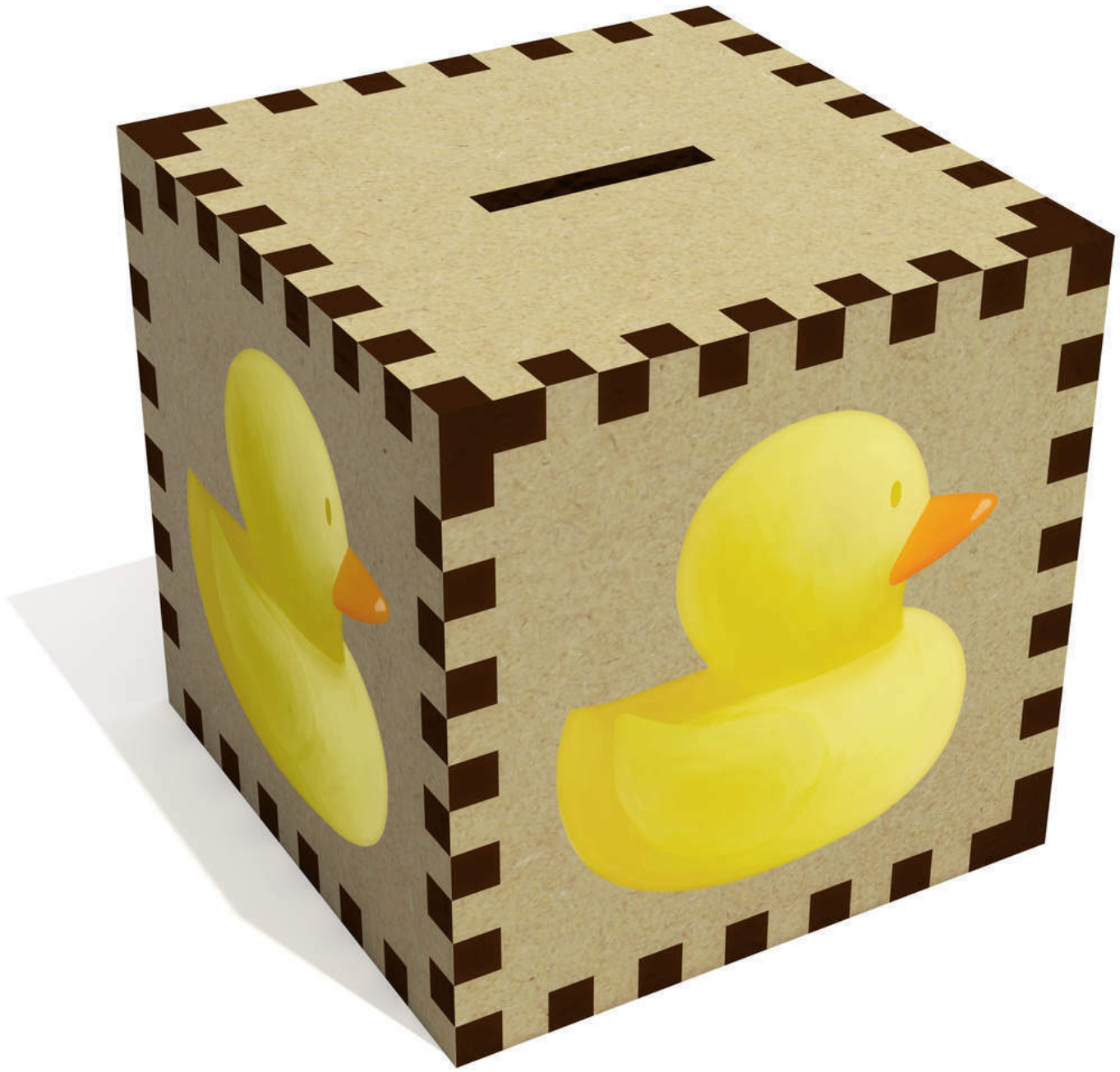
DuckDB's new Postgres Scanner extension can read PostgreSQL's tables while PostgreSQL is running and compute the answers to complex OLAP SQL queries often faster than PostgreSQL itself can without the need to duplicate data. The Postgres Scanner is currently in preview and we are curious to hear what you think. If you find any issues with the Postgres Scanner, please report them.

Modern Data Stack in a Box with DuckDB

Publication date: 2022-10-12

Author: Jacob Matson

TL;DR: A fast, free, and open-source Modern Data Stack (MDS) can now be fully deployed on your laptop or to a single machine using the combination of DuckDB, Meltano, dbt, and Apache Superset.



This post is a collaboration with Jacob Matson and cross-posted on dataduel.co.

Summary

There is a large volume of literature, e.g., 1 and 2, about scaling data pipelines. "Use Kafka! Build a lake house! Don't build a lake house, use Snowflake! Don't use Snowflake, use XYZ!" However, with advances in hardware and the rapid maturation of data software, there is a simpler approach. This article will light up the path to highly performant single node analytics with an MDS-in-a-box open source stack: Meltano, DuckDB, dbt, & Apache Superset on Windows using Windows Subsystem for Linux (WSL). There are many options within the MDS, so if you are using another stack to build an MDS-in-a-box, please share it with the community on the DuckDB Twitter, GitHub, or Discord, or the dbt slack! Or just stop by for a friendly debate about our choice of tools!

Motivation

What is the Modern Data Stack, and why use it? The MDS can mean many things (see examples and a historical perspective), but fundamentally it is a return to using SQL for data transformations by combining multiple best-in-class software tools to form a stack. A typical stack would include (at least!) a tool to extract data from sources and load it into a data warehouse, dbt to transform and analyze that data in the warehouse, and a business intelligence tool. The MDS leverages the accessibility of SQL in combination with software development best practices like git to enable analysts to scale their impact across their companies.

Why build a bundled Modern Data Stack on a single machine, rather than on multiple machines and on a data warehouse? There are many advantages!

- Simplify for higher developer productivity
- Reduce costs by removing the data warehouse
- Deploy with ease either locally, on-premise, in the cloud, or all 3
- Eliminate software expenses with a fully free and open-source stack
- Maintain high performance with modern software like DuckDB and increasingly powerful single-node compute instances
- Achieve self-sufficiency by completing an end-to-end proof of concept on your laptop
- Enable development best practices by integrating with GitHub
- Enhance security by (optionally) running entirely locally or on-premise

If you contribute to an open-source community or provide a product within the Modern Data Stack, there is an additional benefit!

- Increase adoption of your tool by providing a free and self-contained example stack
  - Dagster's example project uses DuckDB for this already!
  - Reach out on the DuckDB Twitter, GitHub, or Discord, or the dbt slack to share an example using your tool with the community!

Trade-offs

One key component of the MDS is the unlimited scalability of compute. How does that align with the MDS-in-a-box approach? Today, cloud computing instances can vertically scale significantly more than in the past (for example, 224 cores and 24 TB of RAM on AWS!). Laptops are more powerful than ever. Now that new OLAP tools like DuckDB can take better advantage of that compute, horizontal scaling is no longer necessary for many analyses! Also, this MDS-in-a-box can be duplicated with ease to as many boxes as needed if partitioned by data subject area. So, while infinite compute is sacrificed, significant scale is still easily achievable.

Due to this tradeoff, this approach is more of an "Open Source Analytics Stack in a box" than a traditional MDS. It sacrifices infinite scale for significant simplification and the other benefits above.

Choosing a Problem

Given that the NBA season is starting soon, a monte carlo type simulation of the season is both topical and well-suited for analytical SQL. This is a particularly great scenario to test the limits of DuckDB because it only requires simple inputs and easily scales out to massive numbers of records. This entire project is held in a GitHub repo, which you can find on GitHub.

Building the Environment

The detailed steps to build the project can be found in the repo, but the high-level steps will be repeated here. As a note, Windows Subsystem for Linux (WSL) was chosen to support Apache Superset, but the other components of this stack can run directly on any operating system. Thankfully, using Linux on Windows has become very straightforward.

1. Install Ubuntu 20.04 on WSL.
2. Upgrade your packages (`sudo apt update`).
3. Install python.
4. Clone the git repo.
5. Run `make build` and then `make run` in the terminal.
6. Create super admin user for Superset in the terminal, then login and configure the database.
7. Run test queries in superset to check your work.

Meltano as a Wrapper for Pipeline Plugins

In this example, Meltano pulls together multiple bits and pieces to allow the pipeline to be run with a single statement. The first part is the tap (extractor) which is 'tap-spreadsheets-anywhere'. This tap allows us to get flat data files from various sources. It should be noted that DuckDB can consume directly from flat files (locally and over the network), or SQLite and PostgreSQL databases. However, this tap was chosen to provide a clear example of getting static data into your database that can easily be configured in the meltano.yml file. Meltano also becomes more beneficial as the complexity of your data sources increases.

```
plugins:
  extractors:
  - name: tap-spreadsheets-anywhere
    variant: ets
    pip_url: git+https://github.com/ets/tap-spreadsheets-anywhere.git
# data sources are configured inside of this extractor
```

The next bit is the target (loader), 'target-duckdb'. This target can take data from any Meltano tap and load it into DuckDB. Part of the beauty of this approach is that you don't have to mess with all the extra complexity that comes with a typical database. DuckDB can be dropped in and is ready to go with zero configuration or ongoing maintenance. Furthermore, because the components and the data are co-located, networking is not a consideration and further reduces complexity.

```
  loaders:
  - name: target-duckdb
    variant: jwills
    pip_url: target-duckdb~=0.4
    config:
      filepath: /tmp/mdsbox.db
      default_target_schema: main
```

Next is the transformer: 'dbt-duckdb'. dbt enables transformations using a combination of SQL and Jinja templating for approachable SQL-based analytics engineering. The dbt adapter for DuckDB now supports parallel execution across threads, which makes the MDS-in-a-box run even faster. Since the bulk of the work is happening inside of dbt, this portion will be described in detail later in the post.

```
  transformers:
  - name: dbt-duckdb
    variant: jwills
    pip_url: dbt-core~=1.2.0 dbt-duckdb~=1.2.0
    config:
      path: /tmp/mdsbox.db
```

Lastly, Apache Superset is included as a Meltano utility to enable some data querying and visualization. Superset leverages DuckDB's SQLAlchemy driver, duckdb_engine, so it can query DuckDB directly as well.

```
  utilities:
  - name: superset
    variant: apache
    pip_url: apache-superset==1.5.0 markupsafe==2.0.1 duckdb-engine==0.6.4
```

With Superset, the engine needs to be configured to open DuckDB in “read-only” mode. Otherwise, only one query can run at a time (simultaneous queries will cause locks). This also prevents refreshing the Superset dashboard while the pipeline is running. In this case, the pipeline runs in under 8 seconds!

Wrangling the Data

The NBA schedule was downloaded from basketball-reference.com, and the Draft Kings win totals from Sept 27th were used for win totals. The schedule and win totals make up the entirety of the data required as inputs for this project. Once converted into CSV format, they were uploaded to the GitHub project, and the meltano.yml file was updated to reference the file locations.

Loading Sources

Once the data is on the web inside of GitHub, Meltano can pull a copy down into DuckDB. With the command `meltano run tap-spreadsheets-anywhere target-duckdb`, the data is loaded into DuckDB, and ready for transformation inside of dbt.

Building dbt Models

After the sources are loaded, the data is transformed with dbt. First, the source models are created as well as the scenario generator. Then the random numbers for that simulation run are generated – it should be noted that the random numbers are recorded as a table, not a view, in order to allow subsequent re-runs of the downstream models with the graph operators for troubleshooting purposes (i.e., `dbt run -s random_num_gen+`). Once the underlying data is laid out, the simulation begins, first by simulating the regular season, then the play-in games, and lastly the playoffs. Since each round of games has a dependency on the previous round, parallelization is limited in this model, which is reflected in the dbt DAG, in this case conveniently hosted on GitHub Pages.

There are a few more design choices worth calling out:

1. Simulation tables and summary tables were split into separate models for ease of use / transparency. So each round of the simulation has a sim model and an end model – this allows visibility into the correct parameters (conference, team, elo rating) to be passed into each subsequent round.
2. To prevent overly deep queries, 'reg_season_end' and 'playoff_sim_r1' have been materialized as tables. While it is slightly slower on build, the performance gains when querying summary tables (i.e., 'season_summary') are more than worth the slowdown. However, it should be noted that even for only 10k sims, the database takes up about 150 MB in disk space. Running at 100k simulations easily expands it to a few GB.

Connecting Superset

Once the dbt models are built, the data visualization can begin. An admin user must be created in superset in order to log in. The instructions for connecting the database can be found in the GitHub project, as well as a note on how to connect it in 'read only mode'.

There are 2 models designed for analysis, although any number of them can be used. 'season_summary' contains various summary statistics for the season, and 'reg_season_sim' contains all simulated game results. This second data set produces an interesting histogram chart. In order to build data visualizations in superset, the dataset must be defined first, the chart built, and lastly, the chart assigned to a dashboard.

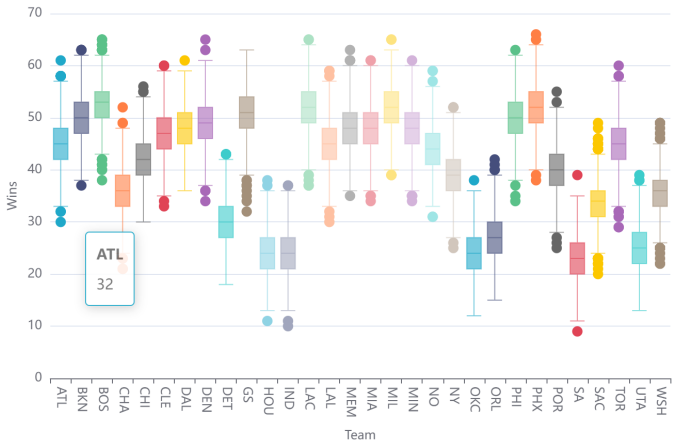
Below is an example Superset dashboard containing several charts based on this data. Superset is able to clearly summarize the data as well as display the level of variability within the monte carlo simulation. The duckdb_engine queries can be refreshed quickly when new simulations are run.

NBA Sim Dashboard Published

Season Summary

conf	team	avg_wins	vegas_wins	elo_vs_vegas
East	DET	29.7	29.5	-0.2
East	ORL	26.7	26.5	-0.2
East	IND	23.9	23.5	-0.4
West	LAC	52.0	52.5	0.5
West	PHX	51.9	52.5	0.6
West	GS	50.9	51.5	0.6
West	DEN	49.1	49.5	0.4
West	MIN	48.2	48.5	0.3
West	MEM	48.2	48.5	0.3
West	DAL	48.1	48.5	0.4
West	LAL	45.2	45.5	0.3
West	NO	44.3	44.5	0.2
West	POR	39.7	39.5	-0.2
West	SAC	33.7	33.5	-0.2
West	UTA	24.9	24.5	-0.4
West	HOU	24.0	23.5	-0.5
West	OKC	23.9	23.5	-0.4
West	SA	23.2	22.5	-0.7

Expected Wins by Team



expanded playoffs

avg_seed	conf	team	avg_wins	made_playoffs	made_conf_semis	made_conf_finals	made_finals	won_finals
2.9	East	BOS	52.7	9853	6682	4133	2520	1459
3.2	East	MIL	51.6	9764	6355	3691	2047	1128
4	East	PHI	49.9	9513	5444	2824	1456	735
4.3	East	BKN	49.9	9437	5332	2804	1399	706
5.1	East	MIA	48	8915	4418	2004	840	344
5.7	East	CLE	47.1	8496	3790	1701	733	308
6.6	East	TOR	45	7548	2902	1102	427	158
6.9	East	ATL	45.2	7081	2663	1027	386	144
8.4	East	CHI	42.1	4746	1443	479	147	42
9.6	East	NY	39.3	2780	645	168	35	9
11	East	CHA	36.4	979	191	35	7	3
11.1	East	WSH	35.4	833	129	31	3	2
13.1	East	DET	29.7	49	6	1	N/A	N/A
13.8	East	ORL	26.7	6	N/A	N/A	N/A	N/A
14.5	East	IND	23.9	N/A	N/A	N/A	N/A	N/A

Conclusion

The ecosystem around DuckDB has grown such that it integrates well with the Modern Data Stack. The MDS-in-a-box is a viable approach for smaller data projects, and would work especially well for read-heavy analytics. There were a few other learnings from this experiment. Superset dashboards are easy to construct, but they are not scriptable and must be built in the GUI (the paid hosted version, Preset, does support exporting as YAML). Also, while you can do monte carlo analysis in SQL, it may be easier to do in another language. However, this shows how far you can stretch the capabilities of SQL!

Next Steps

There are additional directions to take this project. One next step could be to Dockerize this workflow for even easier deployments. If you want to put together a Docker example, please reach out! Another adjustment to the approach could be to land the final outputs in Parquet files, and to read them with in-memory DuckDB connections. Those files could even be landed in an S3-compatible object store (and still read by DuckDB), although that adds complexity compared with the in-a-box approach! Additional MDS components could also be integrated for data quality monitoring, lineage tracking, etc.

Josh Wills is also in the process of making an interesting enhancement to dbt-duckdb! Using the sqlglot library, dbt-duckdb would be able to automatically transpile dbt models written using the SQL dialect of other databases (including Snowflake and BigQuery) to DuckDB. Imagine if you could test out your queries locally before pushing to production... Join the DuckDB channel of the dbt slack to discuss the possibilities!

Please reach out if you use this or another approach to build an MDS-in-a-box! Also, if you are interested in writing a guest post for the DuckDB blog, please reach out on Discord!

Lightweight Compression in DuckDB

Publication date: 2022-10-28

Author: Mark Raasveldt

TL;DR: DuckDB supports efficient lightweight compression that is automatically used to keep data size down without incurring high costs for compression and decompression.

When working with large amounts of data, compression is critical for reducing storage size and egress costs. Compression algorithms typically reduce data set size by **75-95%**, depending on how compressible the data is. Compression not only reduces the storage footprint of a data set, but also often **improves performance** as less data has to be read from disk or over a network connection.

Column store formats, such as DuckDB's native file format or Parquet, benefit especially from compression. That is because data within an individual column is generally very similar, which can be exploited effectively by compression algorithms. Storing data in row-wise format

results in interleaving of data of different columns, leading to lower compression rates.

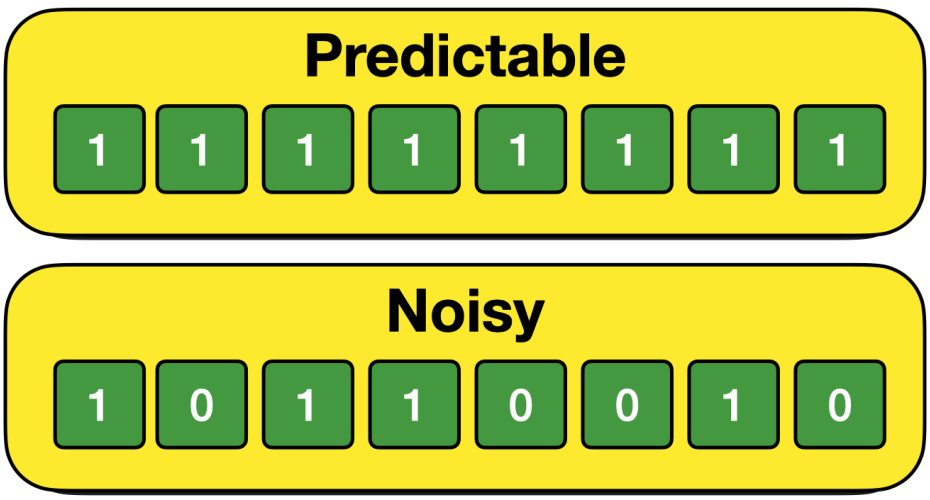
DuckDB added support for compression at the end of last year. As shown in the table below, the compression ratio of DuckDB has continuously improved since then and is still actively being improved. In this blog post, we discuss how compression in DuckDB works, and the design choices and various trade-offs that we have made while implementing compression for DuckDB's storage format.

Version	Taxi	On Time	`lineitem`	Notes	Date
DuckDB v0.2.8	15.3 GB	1.73 GB	0.85 GB	Uncompressed	July 2021
DuckDB v0.2.9	11.2 GB	1.25 GB	0.79 GB	RLE + Constant	September 2021
DuckDB v0.3.2	10.8 GB	0.98 GB	0.56 GB	Bitpacking	February 2022
DuckDB v0.3.3	6.9 GB	0.23 GB	0.32 GB	Dictionary	April 2022
DuckDB v0.5.0	6.6 GB	0.21 GB	0.29 GB	FOR	September 2022
DuckDB dev	4.8 GB	0.21 GB	0.17 GB	FSST + Chimp	now()
CSV	17.0 GB	1.11 GB	0.72 GB		
Parquet (Uncompressed)	4.5 GB	0.12 GB	0.31 GB		
Parquet (Snappy)	3.2 GB	0.11 GB	0.18 GB		
Parquet (Zstd)	2.6 GB	0.08 GB	0.15 GB		

Compression Intro

At its core, compression algorithms try to find patterns in a data set in order to store it more cleverly. **Compressibility** of a data set is therefore dependent on whether or not such patterns can be found, and whether they exist in the first place. Data that follows a fixed pattern can be compressed significantly. Data that does not have any patterns, such as random noise, cannot be compressed. Formally, the compressibility of a dataset is known as its entropy.

As an example of this concept, let us consider the following two data sets.



The constant data set can be compressed by simply storing the value of the pattern and how many times the pattern repeats (e.g., `1x8`). The random noise, on the other hand, has no pattern, and is therefore not compressible.

General-Purpose Compression Algorithms

The compression algorithms that most people are familiar with are *general-purpose compression algorithms*, such as Zip, Gzip or Zstd. General-purpose compression algorithms work by finding patterns in bits. They are therefore agnostic to data types, and can be used on any stream of bits. They can be used to compress files, but they can also be applied to arbitrary data sent over a socket connection.

General-purpose compression is flexible and very easy to set up. There are a number of high quality libraries available (such as Zstd, Snappy or LZ4) that provide compression, and they can be applied to any data set stored in any manner.

The downside of general-purpose compression is that (de)compression is generally expensive. While this does not matter if we are reading and writing from a hard disk or over a slow internet connection, the speed of (de)compression can become a bottleneck when data is stored in RAM.

Another downside is that these libraries operate as a *black box*. They operate on streams of bits, and do not reveal information of their internal state to the user. While that is not a problem if you are only looking to decrease the size of your data, it prevents the system from taking advantage of the patterns found by the compression algorithm during execution.

Finally, general-purpose compression algorithms work better when compressing large chunks of data. As illustrated in the table below, compression ratios suffer significantly when compressing small amounts of data. To achieve a good compression ratio, blocks of at least **256 kB** must be used.

Compression	1 kB	4 kB	16 kB	64 kB	256 kB	1 MB
zstd	1.72	2.1	2.21	2.41	2.54	2.73
lz4	1.29	1.5	1.52	1.58	1.62	1.64
gzip	1.7	2.13	2.28	2.49	2.62	2.67

This is relevant because the block size is the minimum amount of data that must be decompressed when reading a single row from disk. Worse, as DuckDB compresses data on a per-column basis, the block size would be the minimum amount of data that must be decompressed per column. With a block size of 256 kB, fetching a single row could require decompressing multiple megabytes of data. This can cause queries that fetch a low number of rows, such as `SELECT * FROM tbl LIMIT 5` or `SELECT * FROM tbl WHERE id = 42` to incur significant costs, despite appearing to be very cheap on the surface.

Lightweight Compression Algorithms

Another option for achieving compression is to use specialized lightweight compression algorithms. These algorithms also operate by finding patterns in data. However, unlike general-purpose compression, they do not attempt to find generic patterns in bitstreams. Instead, they operate by finding **specific patterns** in data sets.

By detecting specific patterns, specialized compression algorithms can be significantly more lightweight, providing much faster compression and decompression. In addition, they can be effective on much smaller data sizes. This allows us to decompress a few rows at a time, rather than requiring large blocks of data to be decompressed at once. These specialized compression algorithms can also offer efficient support for random seeks, making data access through an index significantly faster.

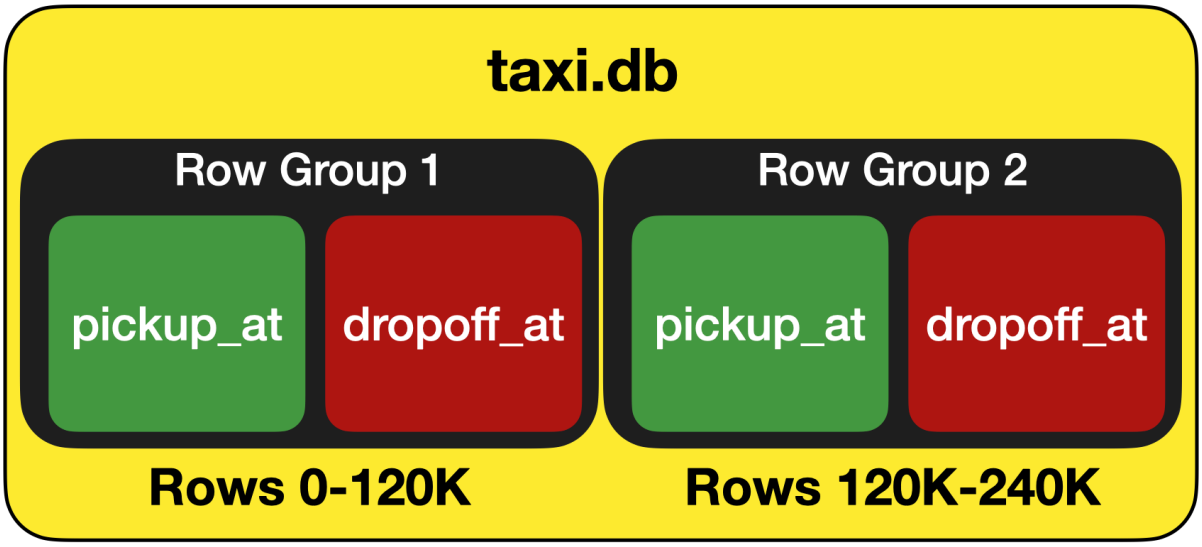
Lightweight compression algorithms also provide us with more fine-grained control over the compression process. This is especially relevant for us as DuckDB's file format uses fixed-size blocks in order to avoid fragmentation for workloads involving deletes and updates. The fine-grained control allows us to fill these blocks more effectively, and avoid having to guess how much compressed data will fit into a buffer.

On the flip side, these algorithms are ineffective if the specific patterns they are designed for do not occur in the data. As a result, individually, these lightweight compression algorithms are no replacement for general-purpose algorithms. Instead, multiple specialized algorithms must be combined in order to capture many different common patterns in data sets.

Compression Framework

Because of the advantages described above, DuckDB uses only specialized lightweight compression algorithms. As each of these algorithms work optimally on different patterns in the data, DuckDB's compression framework must first decide on which algorithm to use to store the data of each column.

DuckDB's storage splits tables into *Row Groups*. These are groups of `120K` rows, stored in columnar chunks called *Column Segments*. This storage layout is similar to Parquet – but with an important difference: columns are split into blocks of a fixed-size. This design decision was made because DuckDB's storage format supports in-place ACID modifications to the storage format, including deleting and updating rows, and adding and dropping columns. By partitioning data into fixed size blocks the blocks can be easily reused after they are no longer required and fragmentation is avoided.



The compression framework operates within the context of the individual *Column Segments*. It operates in two phases. First, the data in the column segment is *analyzed*. In this phase, we scan the data in the segment and find out the best compression algorithm for that particular segment. After that, the *compression* is performed, and the compressed data is written to the blocks on disk.

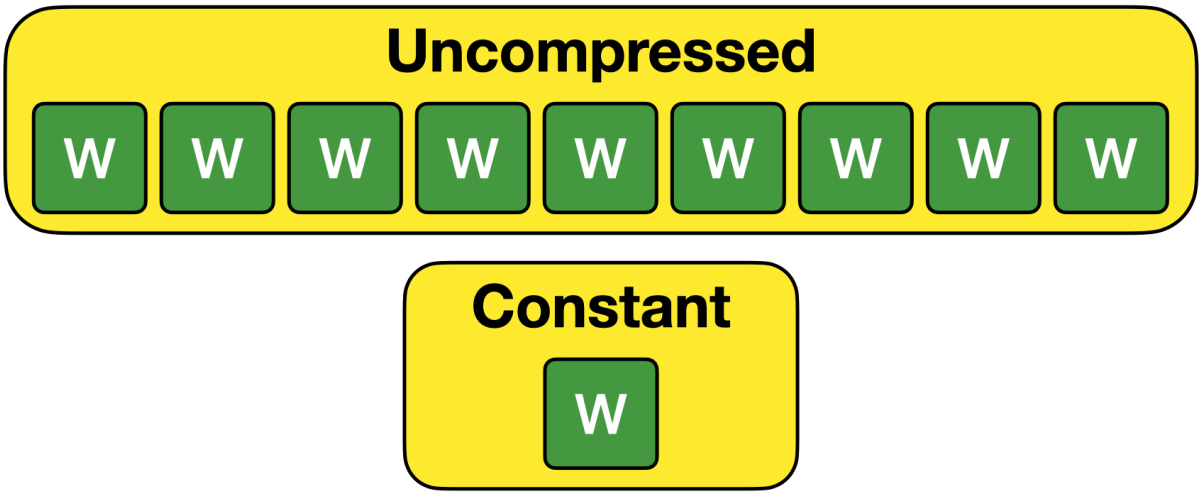
While this approach requires two passes over the data within a segment, this does not incur a significant cost, as the amount of data stored in one segment is generally small enough to fit in the CPU caches. A sampling approach for the analyze step could also be considered, but in general we value choosing the best compression algorithm and reducing file size over a minor increase in compression speed.

Compression Algorithms

DuckDB implements several lightweight compression algorithms, and we are in the process of adding more to the system. We will go over a few of these compression algorithms and how they work in the following sections.

Constant Encoding

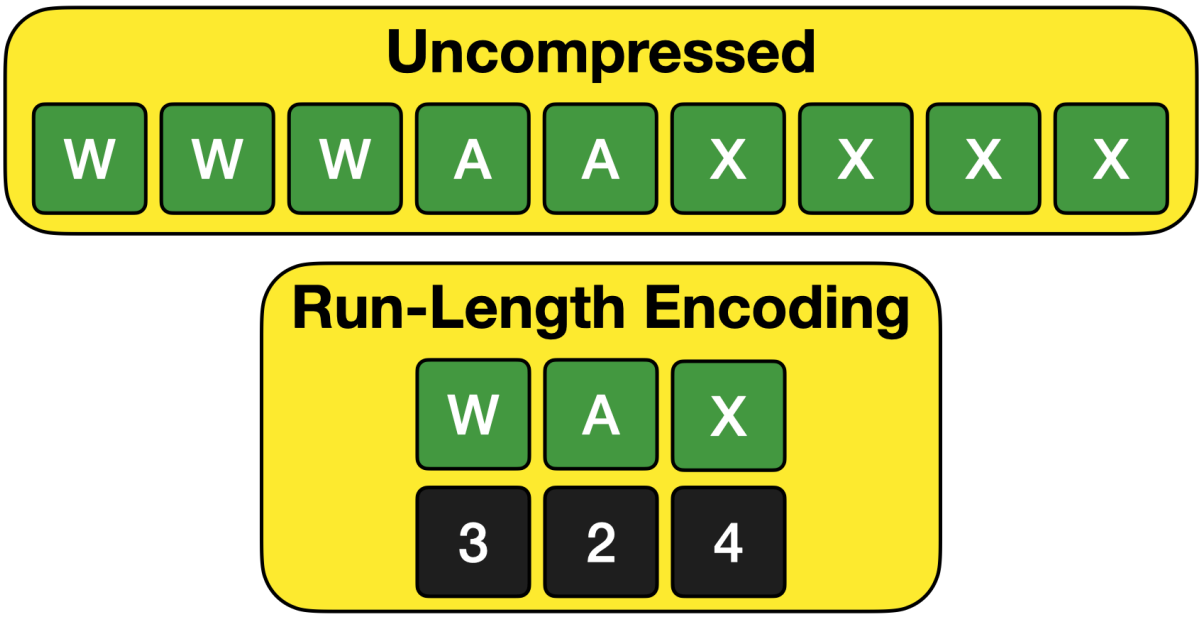
Constant encoding is the most straightforward compression algorithm in DuckDB. Constant encoding is used when every single value in a column segment is the same value. In that case, we store only that single value. This encoding is visualized below.



When applicable, this encoding technique leads to tremendous space savings. While it might seem like this technique is rarely applicable – in practice it occurs relatively frequently. Columns might be filled with `NULL` values, or have values that rarely change (such as e.g., a `year` column in a stream of sensor data). Because of this compression algorithm, such columns take up almost no space in DuckDB.

Run-Length Encoding (RLE)

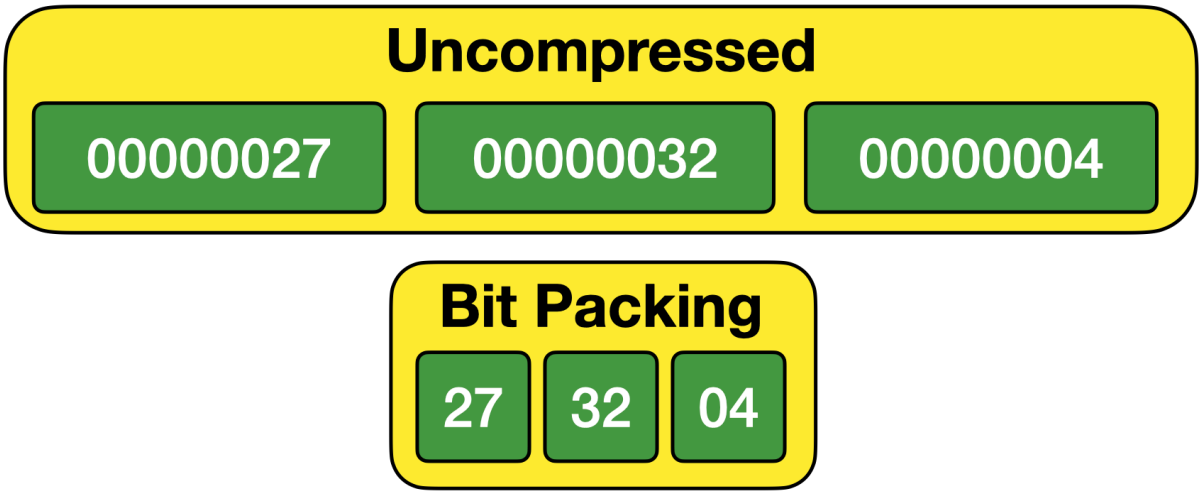
Run-length encoding (RLE) is a compression algorithm that takes advantage of repeated values in a dataset. Rather than storing individual values, the data set is decomposed into a pair of (value, count) tuples, where the count represents how often the value is repeated. This encoding is visualized below.



RLE is powerful when there are many repeating values in the data. This might occur when data is sorted or partitioned on a particular attribute. It is also useful for columns that have many missing (NULL) values.

Bit Packing

Bit Packing is a compression technique that takes advantage of the fact that integral values rarely span the full range of their data type. For example, four-byte integer values can store values from negative two billion to positive two billion. Frequently the full range of this data type is not used, and instead only small numbers are stored. Bit packing takes advantage of this by removing all of the unnecessary leading zeros when storing values. An example (in decimal) is provided below.

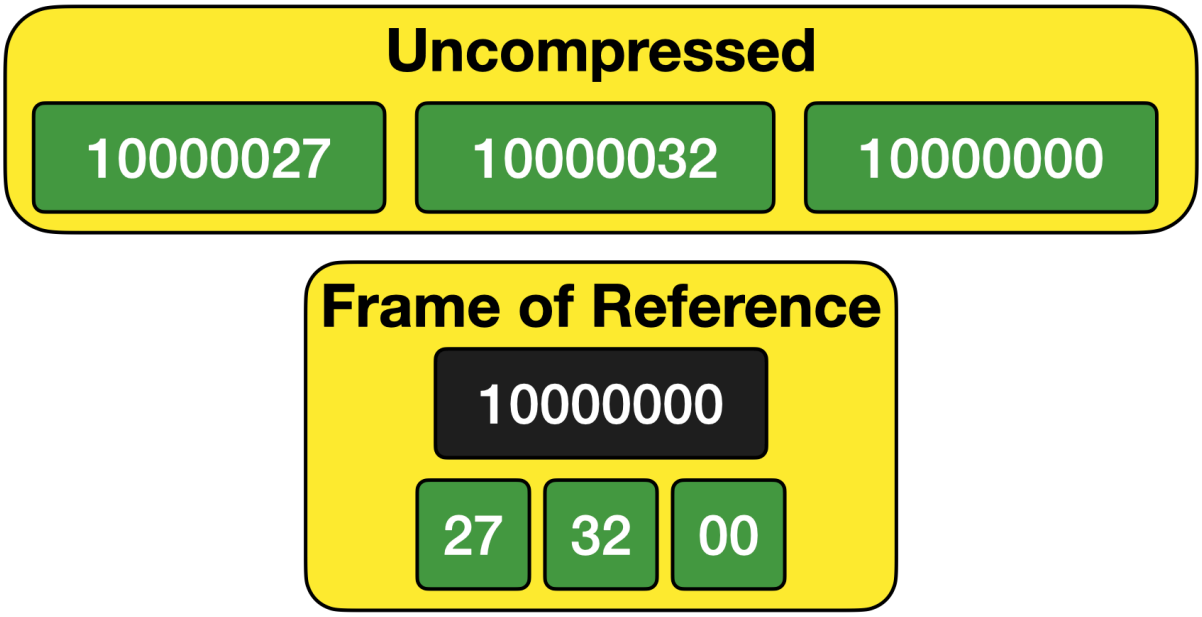


For bit packing compression, we keep track of the maximum value for every `1024` values. The maximum value determines the bit packing width, which is the number of bits necessary to store that value. For example, when storing a set of values with a maximum value of `32`, the bit packing width is `5` bits, down from the `32` bits per value that would be required to store uncompressed four-byte integers.

Bit packing is very powerful in practice. It is also convenient to users – as due to this technique there are no storage size differences between using the various integer types. A `BIGINT` column will be stored in the exact same amount of space as an `INTEGER` column. That relieves the user from having to worry about which integer type to choose.

Frame of Reference

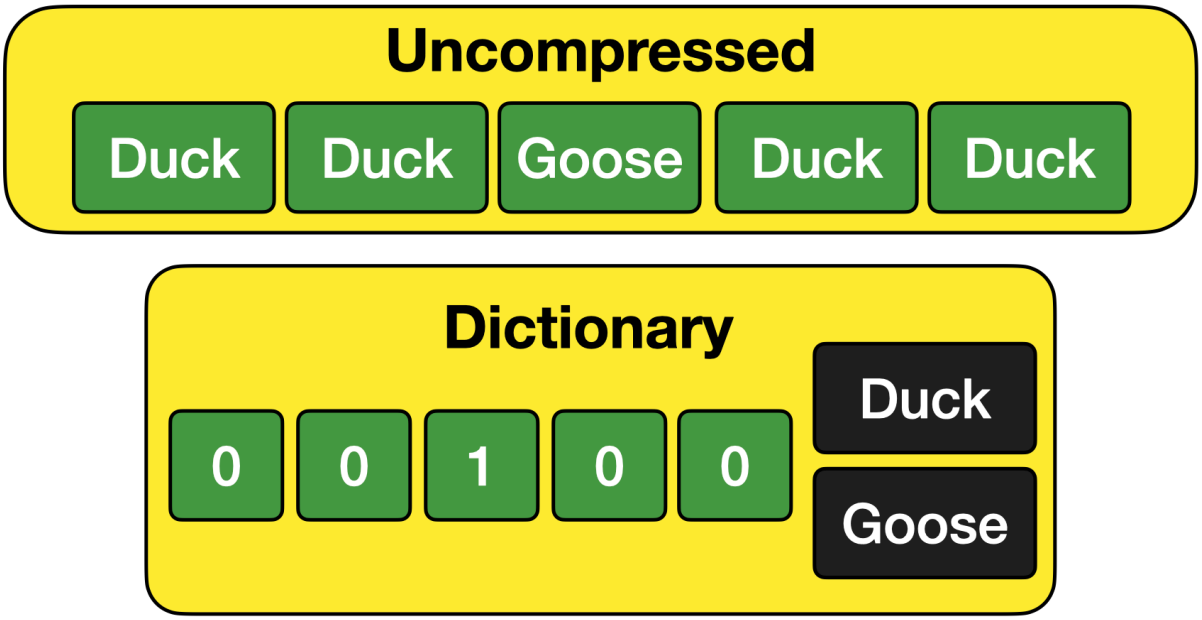
Frame of Reference encoding is an extension of bit packing, where we also include a frame. The frame is the minimum value found in the set of values. The values are stored as the offset from this frame. An example of this is given below.



While this might not seem particularly useful at a first glance, it is very powerful when storing dates and timestamps. That is because dates and timestamps are stored as Unix Timestamps in DuckDB, i.e., the offset since `1970-01-01` in either days (for dates) or microseconds (for timestamps). When we have a set of date or timestamp values, the absolute numbers might be very high, but the numbers are all very close together. By applying a frame before bit packing, we can often improve our compression ratio tremendously.

Dictionary Encoding

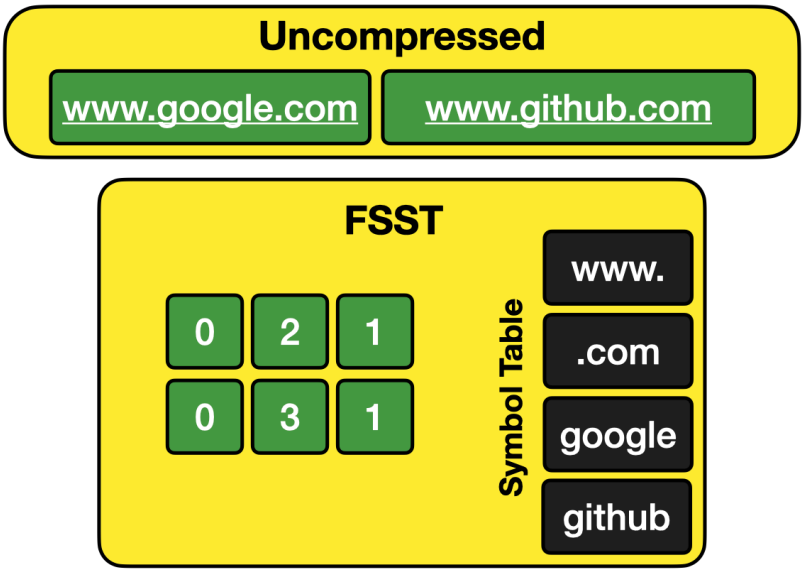
Dictionary encoding works by extracting common values into a separate dictionary, and then replacing the original values with references to said dictionary. An example is provided below.



Dictionary encoding is particularly efficient when storing text columns with many duplicate entries. The much larger text values can be replaced by small numbers, which can in turn be efficiently bit packed together.

FSST

Fast Static Symbol Table compression is an extension to dictionary compression, that not only extracts repetitions of entire strings, but also extracts repetitions *within* strings. This is effective when storing strings that are themselves unique, but have a lot of repetition within the strings, such as URLs or e-mail addresses. An image illustrating how this works is shown below.



For those interested in learning more, watch the talk by Peter Boncz here.

Chimp & Patas

Chimp is a very new compression algorithm that is designed to compress floating point values. It is based on Gorilla compression. The core idea behind Gorilla and Chimp is that floating point values, when XOR'd together, seem to produce small values with many trailing and leading zeros. These algorithms then work on finding an efficient way of storing the trailing and leading zeros.

After implementing Chimp, we have been inspired and worked on implementing Patas, which uses many of the same ideas but optimizes further for higher decompression speed. Expect a future blog post explaining these in more detail soon!

Inspecting Compression

The PRAGMA `storage_info` can be used to inspect the storage layout of tables and columns. This can be used to inspect which compression algorithm has been chosen by DuckDB to compress specific columns of a table.

```
SELECT * EXCLUDE (column_path, segment_id, start, stats, persistent, block_id, block_offset, has_updates)
FROM pragma_storage_info('taxi')
USING SAMPLE 10 ROWS
ORDER BY row_group_id;
```

row_group_id	column_name	column_id	segment_type	count	compression
4	extra	13	FLOAT	65536	Chimp
20	tip_amount	15	FLOAT	65536	Chimp
26	pickup_latitude	6	VALIDITY	65536	Constant
46	tolls_amount	16	FLOAT	65536	RLE
73	store_and_fwd_flag	8	VALIDITY	65536	Uncompressed
96	total_amount	17	VALIDITY	65536	Constant

row_group_id	column_name	column_id	segment_type	count	compression
111	total_amount	17	VALIDITY	65536	Constant
141	pickup_at	1	TIMESTAMP	52224	BitPacking
201	pickup_longitude	5	VALIDITY	65536	Constant
209	passenger_count	3	TINYINT	65536	BitPacking

Conclusion & Future Goals

Compression has been tremendously successful in DuckDB, and we have made great strides in reducing the storage requirements of the system. We are still actively working on extending compression within DuckDB, and are looking to improve the compression ratio of the system even further, both by improving our existing techniques and implementing several others. Our goal is to achieve compression on par with Parquet with Snappy, while using only lightweight specialized compression techniques that are very fast to operate on.

Announcing DuckDB 0.6.0

Publication date: 2022-11-14

Author: Mark Raasveldt

The DuckDB team is happy to announce the latest DuckDB version (0.6.0) has been released. This release of DuckDB is named "Oxyura" after the White-headed duck (Oxyura leucocephala) which is an endangered species native to Eurasia.

To install the new version, please visit the installation guide. Note that the release is still being rolled out, so not all artifacts may be published yet. The full release notes can be found on GitHub.

What's in 0.6.0

The new release contains many improvements to the storage system, general performance improvements, memory management improvements and new features. Below is a summary of the most impactful changes, together with the linked PRs that implement the features.

Storage Improvements

As we are working towards stabilizing the storage format and moving towards version 1.0, we have been actively working on improving our storage format, including many compression improvements.

Optimistic writing to disk. In previous DuckDB versions, the data of a single transaction was first loaded into memory, and would only be written to disk on a commit. While this works fine when data is loaded in batches that fit in memory, it does not work well when loading a lot of data in a single transaction, such as when ingesting one very large file into the system.

This version introduces optimistic writing to disk. When loading large data sets in a single transaction, data is compressed and streamed to the database file, even before the `COMMIT` has occurred. When the transaction is committed, the data will already have been written to disk, and no further writing has to happen. On a rollback, any optimistically written data is reclaimed by the system.

Parallel data loading. In addition to optimistically writing data to disk, this release includes support for parallel data loading into individual tables. This greatly improves performance of data loading on machines that have multiple cores (i.e., all modern machines).

Below is a benchmark comparing loading time of 150 million rows of the Taxi dataset from a Parquet file on an M1 Max with 10 cores:

Version	Load time
v0.5.1	91.4 s
v0.6.0	17.2 s

DuckDB supports two modes – the `order-preserving` and the `non-order-preserving` parallel data load.

The order-preserving load preserves the insertion order so that e.g., the first line in your CSV file is the first line in the DuckDB table. The non-order-preserving load does not offer such guarantees – and instead might re-order the data on load. By default the order-preserving load is used, which involves some extra book-keeping. The preservation of insertion order can be disabled using the `SET preserve_insertion_order = false`{:.language-sql .highlight} statement.

Compression Improvements

FSST. The Fast Static Symbol Table compression algorithm is introduced in this version. This state-of-the-art compression algorithm compresses data *inside* strings using a dictionary, while maintaining support for efficient scans and random look-ups. This greatly increases the compression ratio of strings that have many unique values but with common elements, such as e-mail addresses or URLs.

The compression ratio improvements of the TPC-H SF1 dataset are shown below:

Compression	Size
Uncompressed	761 MB
Dictionary	510 MB
FSST + Dictionary	251 MB

Chimp. The Chimp compression algorithm is included, which is the state-of-the-art in lightweight floating point compression. Chimp is an improved version of Gorillas, that achieves both a better compression ratio as well as faster decompression speed.

Patas. Patas is a novel floating point compression method that iterates upon the Chimp algorithm by optimizing for a single case in the Chimp algorithm. While Patas generally has a slightly lower compression ratio than Chimp, it has significantly faster decompression speed, almost matching uncompressed data in read speed.

The compression ratio of a dataset containing temperatures of cities stored as double (8-byte floating point numbers) is shown below:

Compression	Size
Uncompressed	25.4 MB
Chimp	9.7 MB
Patas	10.2 MB

Performance Improvements

DuckDB aims to have very high performance for a wide variety of workloads. As such, we are always working to improve performance for various workloads. This release is no different.

Parallel CSV Loading (Experimental). In this release we are launching a new experimental parallel CSV reader. This greatly improves the ingestion speed of large CSV files into the system. While we have done our best to make the parallel CSV reader robust – CSV parsing is a minefield as there is such a wide variety of different files out there – so we have marked the reader as experimental for now.

The parallel CSV reader can be enabled by setting the `experimental_parallel_csv` flag to true. We aim to make the parallel CSV reader the default reader in future DuckDB versions.

```
SET experimental_parallel_csv = true;
```

Below is the load time of a 720 MB CSV file containing the `lineitem` table from the TPC-H benchmark,

Variant	Load time
Single-threaded	3.5 s
Parallel	0.6 s

Parallel CREATE INDEX & Index Memory Management Improvements. Index creation is also sped up significantly in this release, as the `CREATE INDEX` statement can now be executed fully in parallel. In addition, the number of memory allocations done by the ART is greatly reduced through inlining of small structures which both reduces memory size and further improves performance.

The timings of creating an index on a single column with 16 million values is shown below.

Version	Create index time
v0.5.1	5.92 s
v0.6.0	1.38 s

Parallel count(DISTINCT). Aggregates containing `DISTINCT` aggregates, most commonly used for exact distinct count computation (e.g., `count(DISTINCT col)`) previously had to be executed in single-threaded mode. Starting with v0.6.0, DuckDB can execute these queries in parallel, leading to large speed-ups.

SQL Syntax Improvements

SQL is the primary way of interfacing with DuckDB – and DuckDB tries to have an easy to use SQL dialect. This release contains further improvements to the SQL dialect.

UNION Type. This release introduces the UNION type, which allows sum types to be stored and queried in DuckDB. For example:

```
CREATE TABLE messages (u UNION(num INTEGER, error VARCHAR));
INSERT INTO messages VALUES (42);
INSERT INTO messages VALUES ('oh my globs');
SELECT * FROM messages;
```

u
42
oh my globs

Sum types are strongly typed – but they allow a single value in a table to be represented as one of various types. The union page in the documentation contains more information on how to use this new composite type.

FROM-first. Starting with this release, DuckDB supports starting queries with the FROM clause instead of the SELECT clause. In fact, the SELECT clause is fully optional now, and defaults to SELECT *. That means the following queries are now valid in DuckDB:

```
-- SELECT clause is optional, SELECT * is implied (if not included)
FROM tbl;

-- first 5 rows of the table
FROM tbl LIMIT 5;

-- SELECT can be used after the FROM
FROM tbl SELECT l_orderkey;

-- insert all data from tbl1 into tbl2
INSERT INTO tbl2 FROM tbl1;
```

COLUMNS Expression. This release adds support for the COLUMNS expression, inspired by the ClickHouse syntax. The COLUMNS expression allows you to execute expressions or functions on multiple columns without having to duplicate the full expression.

```
CREATE TABLE obs (id INTEGER, val1 INTEGER, val2 INTEGER);
INSERT INTO obs VALUES (1, 10, 100), (2, 20, NULL), (3, NULL, 300);
SELECT min(COLUMNS(*)), count(*) FROM obs;
```

min(obs.id)	min(obs.val1)	min(obs.val2)	count_star()
1	10	100	3

The COLUMNS expression supports all star expressions, including the EXCLUDE and REPLACE syntax. In addition, the COLUMNS expression can take a regular expression as parameter:

```
SELECT COLUMNS('val[0-9]+') FROM obs;
```

val1	val2
10	100
20	NULL
NULL	300

List comprehension support. List comprehension is an elegant and powerful way of defining operations on lists. DuckDB now also supports list comprehension as part of its SQL dialect. For example, the query below now works:

```
SELECT [x + 1 for x in [1, 2, 3]] AS l;
```

l
[2, 3, 4]

Nested types and structures are very efficiently implemented in DuckDB, and are now also more elegant to work with.

Memory Management Improvements

When working with large data sets, memory management is always a potential pain point. By using a streaming execution engine and buffer manager, DuckDB supports many operations on larger than memory data sets. DuckDB also aims to support queries where *intermediate* results do not fit into memory by using disk-spilling techniques, and has support for an efficient out-of-core sort, out-of-core window functions and an out-of-core hash join.

This release further improves on that by greatly optimizing the out-of-core hash join, resulting in a much more graceful degradation in performance as the data exceeds the memory limit.

Memory limit (GB)	Old time (s)	New time (s)
10	1.97	1.96
9	1.97	1.97
8	2.23	2.22
7	2.23	2.44
6	2.27	2.39
5	2.27	2.32
4	2.81	2.45
3	5.60	3.20
2	7.69	3.28
1	17.73	4.35

jemalloc. In addition, this release bundles the jemalloc allocator with the Linux version of DuckDB by default, which fixes an outstanding issue where the standard `GLIBC` allocator would not return blocks to the operating system, unnecessarily leading to out-of-memory errors on the Linux version. Note that this problem does not occur on macOS or Windows, and as such we continue using the standard allocators there (at least for now).

Shell Improvements

DuckDB has a command line interface (CLI) that is adapted from SQLite's CLI, and therefore supports a similar interface to SQLite. All of the tables in this blog post have been generated using the `.mode markdown` in the CLI.

The DuckDB shell also offers several improvements over the SQLite shell, such as syntax highlighting, and this release includes a few new goodies.

DuckBox Rendering. This release includes a new `.mode duckbox` rendering that is used by default. This box rendering adapts to the size of the shell, and leaves out columns and rows to provide a better overview of a result. It very quickly renders large result sets by leaving out rows in the middle. That way, typing `SELECT * FROM tbl` in the shell no longer blows it up. In fact, this can now be used to quickly get a good feel of a dataset instead.

The number of rows that are rendered can be changed by using the `.maxrows X` setting, and you can switch back to the old rendering using the `.mode box` command.

```
SELECT * FROM '~/Data/nyctaxi/nyc-taxi/2014/04/data.parquet';
```

vendor_id varchar	pickup_at timestamp	dropoff_at timestamp	…	tip_amount float	tolls_amount float	total_amount float
CMT	2014-04-08 08:59:39	2014-04-08 09:28:57	…	3.7	0.0	22.2
CMT	2014-04-08 14:59:22	2014-04-08 15:04:52	…	1.3	0.0	7.8
CMT	2014-04-08 08:45:28	2014-04-08 08:50:41	…	1.2	0.0	7.2
CMT	2014-04-08 08:00:20	2014-04-08 08:11:31	…	1.7	0.0	10.2
CMT	2014-04-08 08:38:36	2014-04-08 08:44:37	…	1.2	0.0	7.2
CMT	2014-04-08 07:52:53	2014-04-08 07:59:12	…	1.3	0.0	7.8
CMT	2014-04-08 16:08:16	2014-04-08 16:12:38	…	1.4	0.0	8.4
CMT	2014-04-08 12:04:09	2014-04-08 12:14:30	…	1.7	0.0	10.2
CMT	2014-04-08 16:18:38	2014-04-08 16:37:04	…	2.5	0.0	17.5
CMT	2014-04-08 15:28:00	2014-04-08 15:34:44	…	1.4	0.0	8.4
·	·	·	·	·	·	·
·	·	·	·	·	·	·
·	·	·	·	·	·	·
CMT	2014-04-25 00:09:34	2014-04-25 00:14:52	…	2.5	0.0	10.0
CMT	2014-04-25 01:59:39	2014-04-25 02:16:07	…	3.5	0.0	21.0
CMT	2014-04-24 23:02:08	2014-04-24 23:47:10	…	8.8	0.0	52.8
CMT	2014-04-25 01:27:11	2014-04-25 01:56:53	…	4.6	0.0	27.6
CMT	2014-04-25 00:15:46	2014-04-25 00:25:37	…	1.0	0.0	11.5
CMT	2014-04-25 00:17:53	2014-04-25 00:22:52	…	1.3	0.0	7.8
CMT	2014-04-25 03:13:19	2014-04-25 03:21:50	…	2.1	0.0	12.6
CMT	2014-04-24 23:53:03	2014-04-25 00:16:01	…	2.85	0.0	31.35
CMT	2014-04-25 00:26:08	2014-04-25 00:31:25	…	1.4	0.0	8.4
CMT	2014-04-24 23:21:39	2014-04-24 23:33:57	…	1.0	0.0	11.5

14618759 rows (20 shown) 18 columns (6 shown)

Context-Aware Auto-Complete. The shell now also ships with context-aware auto-complete. Auto-complete is triggered by pressing the tab character. The shell auto-completes four different groups: (1) keywords, (2) table names + table functions, (3) column names + scalar functions, and (4) file names. The shell looks at the position in the SQL statement to determine which of these auto-completions to trigger. For example:

```
S -> SELECT

SELECT s -> student_id

SELECT student_id F -> FROM


SELECT student_id FROM g -> grades

SELECT student_id FROM 'd -> data/

SELECT student_id FROM 'data/ -> data/grades.csv
```

Progress Bars. DuckDB has supported progress bars in queries for a while now, but they have always been opt-in. In this release we have prettied up the progress bar and enabled it by default in the shell. The progress bar will pop up when a query is run that takes more than 2 seconds, and display an estimated time-to-completion for the query.

```
COPY lineitem TO 'lineitem-big.parquet';
```

In the future we aim to enable the progress bar by default in other clients. For now, this can be done manually by running the following SQL queries:

```
PRAGMA enable_progress_bar;
PRAGMA enable_print_progress_bar;
```

Announcing DuckDB 0.7.0

Publication date: 2023-02-13

Author: Mark Raasveldt

The DuckDB team is happy to announce the latest DuckDB version (0.7.0) has been released. This release of DuckDB is named "Labradorius" after the Labrador Duck (Camptorhynchus labradorius) that was native to North America.

To install the new version, please visit the installation guide. The full release notes can be found on GitHub.

What's in 0.7.0

The new release contains many improvements to the JSON support, new SQL features, improvements to data ingestion and export, and other new features. Below is a summary of the most impactful changes, together with the linked PRs that implement the features.

Data Ingestion/Export Improvements

JSON Ingestion. This version introduces the `read_json` and `read_json_auto` methods. These can be used to ingest JSON files into a tabular format. Similar to `read_csv`, the `read_json` method requires a schema to be specified, while the `read_json_auto` automatically infers the schema of the JSON from the file using sampling. Both new-line delimited JSON and regular JSON are supported.

```
FROM 'data/json/with_list.json';
```

id	name
1	[O, Brother,, Where, Art, Thou?]
2	[Home, for, the, Holidays]
3	[The, Firm]
4	[Broadcast, News]
5	[Raising, Arizona]

Partitioned Parquet/CSV Export. DuckDB has been able to ingest Hive-partitioned Parquet and CSV files for a while. After this release DuckDB will also be able to *write* Hive-partitioned data using the `PARTITION_BY` clause. These files can be exported locally or remotely to S3 compatible storage. Here is a local example:

```
COPY orders TO 'orders' (FORMAT parquet, PARTITION_BY (year, month));
```

This will cause the Parquet files to be written in the following directory structure:

```
orders
├── year=2021
│   ├── month=1
│   │   ├── file1.parquet
│   │   └── file2.parquet
│   └── month=2
│       └── file3.parquet
└── year=2022
    ├── month=11
    │   ├── file4.parquet
    │   └── file5.parquet
    └── month=12
        └── file6.parquet
```

Parallel Parquet/CSV Writing. Parquet and CSV writing are sped up tremendously this release with the parallel Parquet and CSV writer support.

Format	Old	New (8T)
CSV	2.6 s	0.4 s
Parquet	7.5 s	1.3 s

Note that currently the parallel writing is currently limited to non-insertion order preserving – which can be toggled by setting the `preserve_insertion_order` setting to false. In a future release we aim to alleviate this restriction and order parallel insertion order preserving writes as well.

Multi-Database Support

Attach Functionality. This release adds support for attaching multiple databases to the same DuckDB instance. This easily allows data to be transferred between separate DuckDB database files, and also allows data from separate database files to be combined together in individual queries. Remote DuckDB instances (stored on a network accessible location like GitHub, for example) may also be attached.

```
ATTACH 'new_db.db';
CREATE TABLE new_db.tbl (i INTEGER);
INSERT INTO new_db.tbl SELECT * FROM range(1000);
DETACH new_db;
```

See the documentation for more information.

SQLite Storage Back-End. In addition to adding support for attaching DuckDB databases – this release also adds support for *pluggable database engines*. This allows extensions to define their own database and catalog engines that can be attached to the system. Once attached, an engine can support both reads and writes. The SQLite extension makes use of this to add native read/write support for SQLite database files to DuckDB.

```
ATTACH 'sqlite_file.db' AS sqlite (TYPE sqlite);
CREATE TABLE sqlite.tbl (i INTEGER);
INSERT INTO sqlite.tbl VALUES (1), (2), (3);
SELECT * FROM sqlite.tbl;
```

Using this, SQLite database files can be attached, queried and modified as if they are native DuckDB database files. This allows data to be quickly transferred between SQLite and DuckDB – and allows you to use DuckDB's rich SQL dialect to query data stored in SQLite tables.

New SQL Features

Upsert Support. Upsert support is added with this release using the `ON CONFLICT` clause, as well as the `SQLite` compatible `INSERT OR REPLACE`/`INSERT OR IGNORE` syntax.

```
CREATE TABLE movies (id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO movies VALUES (1, 'A New Hope');
FROM movies;
```

id	name
1	A New Hope

```
INSERT OR REPLACE INTO movies VALUES (1, 'The Phantom Menace');
FROM movies;
```

id	name
1	The Phantom Menace

See the documentation for more information.

Lateral Joins. Support for lateral joins is added in this release. Lateral joins are a more flexible variant of correlated subqueries that make working with nested data easier, as they allow easier unnesting of nested data.

Positional Joins. While SQL formally models unordered sets, in practice the order of datasets does frequently have a meaning. DuckDB offers guarantees around maintaining the order of rows when loading data into tables or when exporting data back out to a file – as well as when executing queries such as `LIMIT` without a corresponding `ORDER BY` clause.

To improve support for this use case – this release introduces the `POSITIONAL JOIN`. Rather than joining on the values of rows – this new join type joins rows based on their position in the table.

```
CREATE TABLE t1 AS FROM (VALUES (1), (2), (3)) t(i);
CREATE TABLE t2 AS FROM (VALUES (4), (5), (6)) t(k);
SELECT * FROM t1 POSITIONAL JOIN t2;
```

i	k
1	4
2	5
3	6

Python API Improvements

Query Building. This release introduces easier incremental query building using the Python API by allowing relations to be queried. This allows you to decompose long SQL queries into multiple smaller SQL queries, and allows you to easily inspect query intermediates.

```
>>> import duckdb
>>> lineitem = duckdb.sql('FROM lineitem.parquet')
>>> lineitem.limit(3).show()
```

l_orderkey int32	l_partkey int32	l_suppkey int32	…	l_shipinstruct varchar	l_shipmode varchar	l_comment varchar

1	155190	7706	…	DELIVER IN PERSON	TRUCK	egular courts abov…
1	67310	7311	…	TAKE BACK RETURN	MAIL	ly final dependenc…
1	63700	3701	…	TAKE BACK RETURN	REG AIR	riously. regular, …
3 rows						16 columns (6 shown)

```
>>> lineitem_filtered = duckdb.sql('FROM lineitem WHERE l_orderkey>5000')
>>> lineitem_filtered.limit(3).show()
```

l_orderkey int32	l_partkey int32	l_suppkey int32	…	l_shipinstruct varchar	l_shipmode varchar	l_comment varchar
5024	165411	444	…	NONE	AIR	to the expre
5024	57578	84	…	COLLECT COD	REG AIR	osits hinder caref…
5024	111009	3521	…	NONE	MAIL	zle carefully saut…
3 rows						16 columns (6 shown)

```
>>> duckdb.sql('SELECT min(l_orderkey), max(l_orderkey) FROM lineitem_filtered').show()
```

min(l_orderkey) int32	max(l_orderkey) int32
5024	6000000

Note that everything is lazily evaluated. The Parquet file is not read from disk until the final query is executed – and queries are optimized in their entirety. Executing the decomposed query will be just as fast as executing the long SQL query all at once.

Python Ingestion APIs. This release adds several familiar data ingestion and export APIs that follow standard conventions used by other libraries. These functions emit relations as well – which can be directly queried again.

```
>>> lineitem = duckdb.read_csv('lineitem.csv')
>>> lineitem.limit(3).show()
```

l_orderkey int32	l_partkey int32	l_suppkey int32	…	l_shipinstruct varchar	l_shipmode varchar	l_comment varchar
1	155190	7706	…	DELIVER IN PERSON	TRUCK	egular courts abov…
1	67310	7311	…	TAKE BACK RETURN	MAIL	ly final dependenc…
1	63700	3701	…	TAKE BACK RETURN	REG AIR	riously. regular, …
3 rows						16 columns (6 shown)

```
>>> duckdb.sql('SELECT min(l_orderkey) FROM lineitem').show()
```

min(l_orderkey) int32
1

Polars Integration. This release adds support for tight integration with the Polars DataFrame library, similar to our integration with Pandas DataFrames. Results can be converted to Polars DataFrames using the `.pl()` function.

```
import duckdb
duckdb.sql('SELECT 42').pl()
```

```
shape: (1, 1)
```

42
i32
42

In addition, Polars DataFrames can be directly queried using the SQL interface.

```
import duckdb
import polars as pl
df = pl.DataFrame({'a': 42})
duckdb.sql('SELECT * FROM df').pl()
```

```
shape: (1, 1)
```

a
i64
42

fsspec Filesystem Support. This release adds support for the fsspec filesystem API. fsspec allows users to define their own filesystem that they can pass to DuckDB. DuckDB will then use this file system to read and write data to and from. This enables support for storage back-ends that may not be natively supported by DuckDB yet, such as FTP.

```
import duckdb
from fsspec import filesystem

duckdb.register_filesystem(filesystem('gcs'))

data = duckdb.query("SELECT * FROM read_csv_auto('gcs:///bucket/file.csv')").fetchall()
```

Have a look at the guide for more information

Storage Improvements

Delta Compression. Compression of numeric values in the storage is improved using the new delta and delta-constant compression. This compression method is particularly effective when compressing values that are equally spaced out. For example, sequences of numbers (`1, 2, 3, ...`) or timestamps with a fixed interval between them (`12:00:01, 12:00:02, 12:00:03, ...`).

Final Thoughts

The full release notes can be found on GitHub. We would like to thank all of the contributors for their hard work on improving DuckDB.

JupySQL Plotting with DuckDB

Publication date: 2023-02-24

Author: Eduardo Blancas

TL;DR: JupySQL provides a seamless SQL experience in Jupyter and uses DuckDB to visualize larger than memory datasets in matplotlib.

Introduction

Data visualization is essential for every data practitioner since it allows us to find patterns that otherwise would be hard to see. The typical approach for plotting tabular datasets involves Pandas and Matplotlib. However, this technique quickly falls short as our data grows, given that Pandas introduces a significant memory overhead, making it challenging even to plot a medium-sized dataset.

In this blog post, we'll use JupySQL and DuckDB to efficiently plot *larger-than-memory* datasets in our laptops. JupySQL is a fork of ipython-sql, which adds SQL cells to Jupyter, that is being actively maintained and enhanced by the team at Ploomber.

Combining JupySQL with DuckDB enables a powerful and user friendly local SQL processing experience, especially when combined with JupySQL's new plotting capabilities. There is no need to get beefy (and expensive!) EC2 machines or configure complex distributed frameworks! Get started with JupySQL and DuckDB with our Jupyter Notebook guide, or go directly to an example Colab notebook!

We want JupySQL to offer the best SQL experience in Jupyter, so if you have any feedback, please open an issue on GitHub!

The Problem

One significant limitation when using `pandas` and `matplotlib` for data visualization is that we need to load all our data into memory, making it difficult to plot *larger-than-memory* datasets. Furthermore, given the overhead that `pandas` introduces, we might be unable to visualize some smaller datasets that we might think "fit" into memory.

Let's load a sample `.parquet` dataset using pandas to show the memory overhead:

```
from urllib.request import urlretrieve

_ = urlretrieve("https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2022-01.parquet",
                "yellow_tripdata_2022-01.parquet")
```

The downloaded `.parquet` file takes 36 MB of disk space:

```
ls -lh *.parquet
```

```
-rw-r--r--  1 eduardo  staff    36M Jan 18 14:45 yellow_tripdata_2022-01.parquet
```

Now let's load the `.parquet` as a data frame and see how much memory it takes:

```
import pandas as pd

df = pd.read_parquet("yellow_tripdata_2022-01.parquet")

df_mb = df.memory_usage().sum() / (1024 ** 2)
print(f"Data frame takes {df_mb:.0f} MB")
```

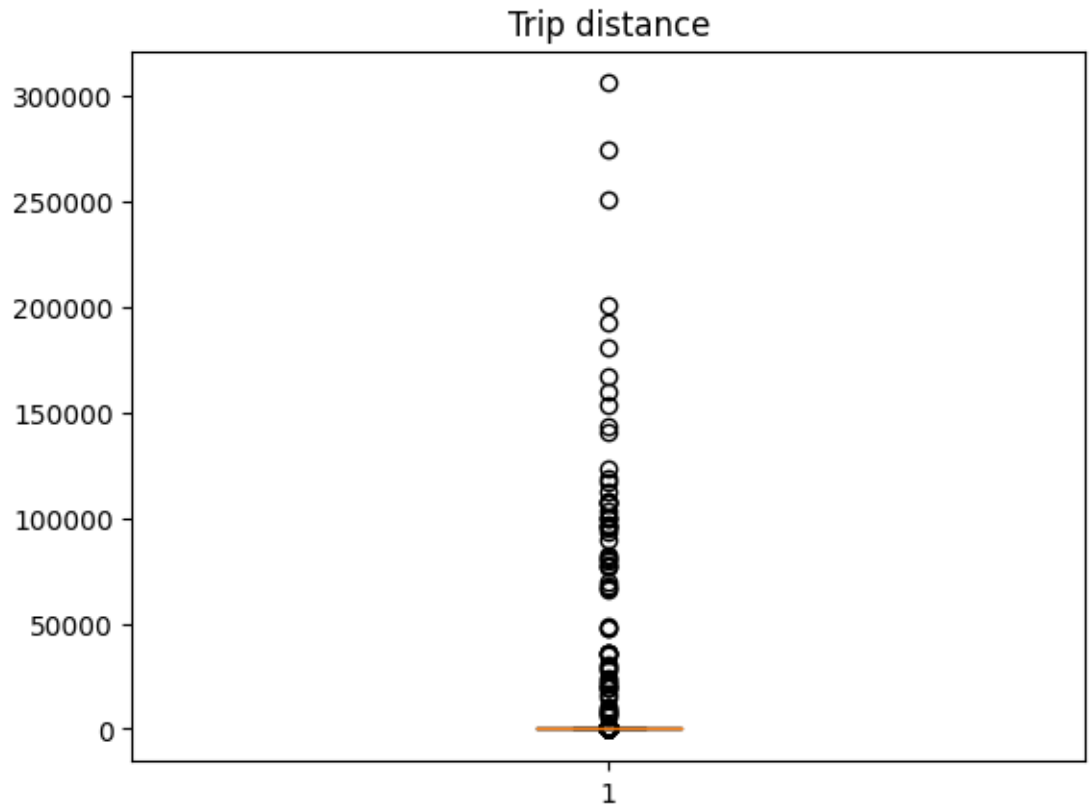
```
Data frame takes 357 MB
```

As you can see, we're using almost 10× as much memory as the file size. Given this overhead, we must be much more conservative about what *larger-than-memory* means, as "medium" files might not fit into memory once loaded. But this is just the beginning of our memory problems.

When plotting data, we often need to preprocess it before it's suitable for visualization. However, if we're not careful, these preprocessing steps will copy our data, dramatically increasing memory. Let's show a practical example.

Our sample dataset contains an observation for each NYC yellow cab trip in January 2022. Let's create a boxplot for the trip's distance:

```
import matplotlib.pyplot as plt

plt.boxplot(df.trip_distance)
_ = plt.title("Trip distance")
```



Wow! It looks like some new yorkers really like taxi rides! Let's put the taxi fans aside to improve the visualization and compute the 99th percentile to use it as the cutoff value:

```
cutoff = df.trip_distance.quantile(q=0.99)
cutoff
```

```
19.7
```

Now, we need to filter out observations larger than the cutoff value; but before we do it, let's create a utility function to capture memory usage:

```
import psutil

def memory_used():
    """Returns memory used in MB"""
    mem = psutil.Process().memory_full_info().uss / (1024 ** 2)
    print(f"Memory used: {mem:.0f} MB")

memory_used()
```

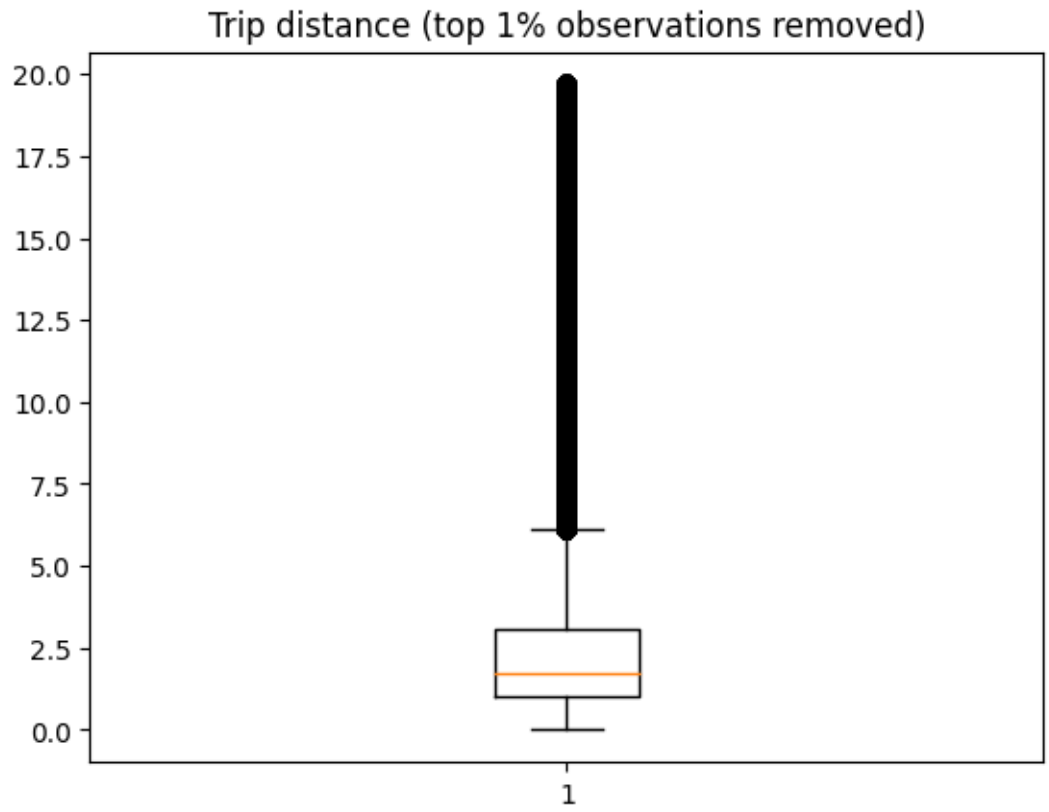
```
Memory used: 941 MB
```

Let's now filter out the observations:

```
df_ = df[df.trip_distance < cutoff]
```

Plot the histogram:

```
plt.boxplot(df_.trip_distance)
_ = plt.title("Trip distance (top 1% observations removed)")
```



We now see more reasonable numbers with the top 1% outliers removed. There are a few trips over 10 miles (perhaps some uptown new yorkers going to Brooklyn for some delicious pizza?)

How much memory are we using now?

```
memory_used()
```

```
Memory used: 1321 MB
```

380 MB more! Loading a 36 MB Parquet file turned into >700 MB in memory after loading and applying one preprocessing step!

So, in reality, when we use `pandas`, what fits in memory is much smaller than we think, and even with a laptop equipped with 16 GB of RAM, we'll be extremely limited in terms of what size of a dataset we process. Of course, we could save a lot of memory by exclusively loading the column we plotted and deleting unneeded data copies; however, let's face it, *this never happens in practice*. When exploring data, we rarely know ahead of time which columns we'll need; furthermore, our time is better spent analyzing and visualizing the data than manually deleting data copies.

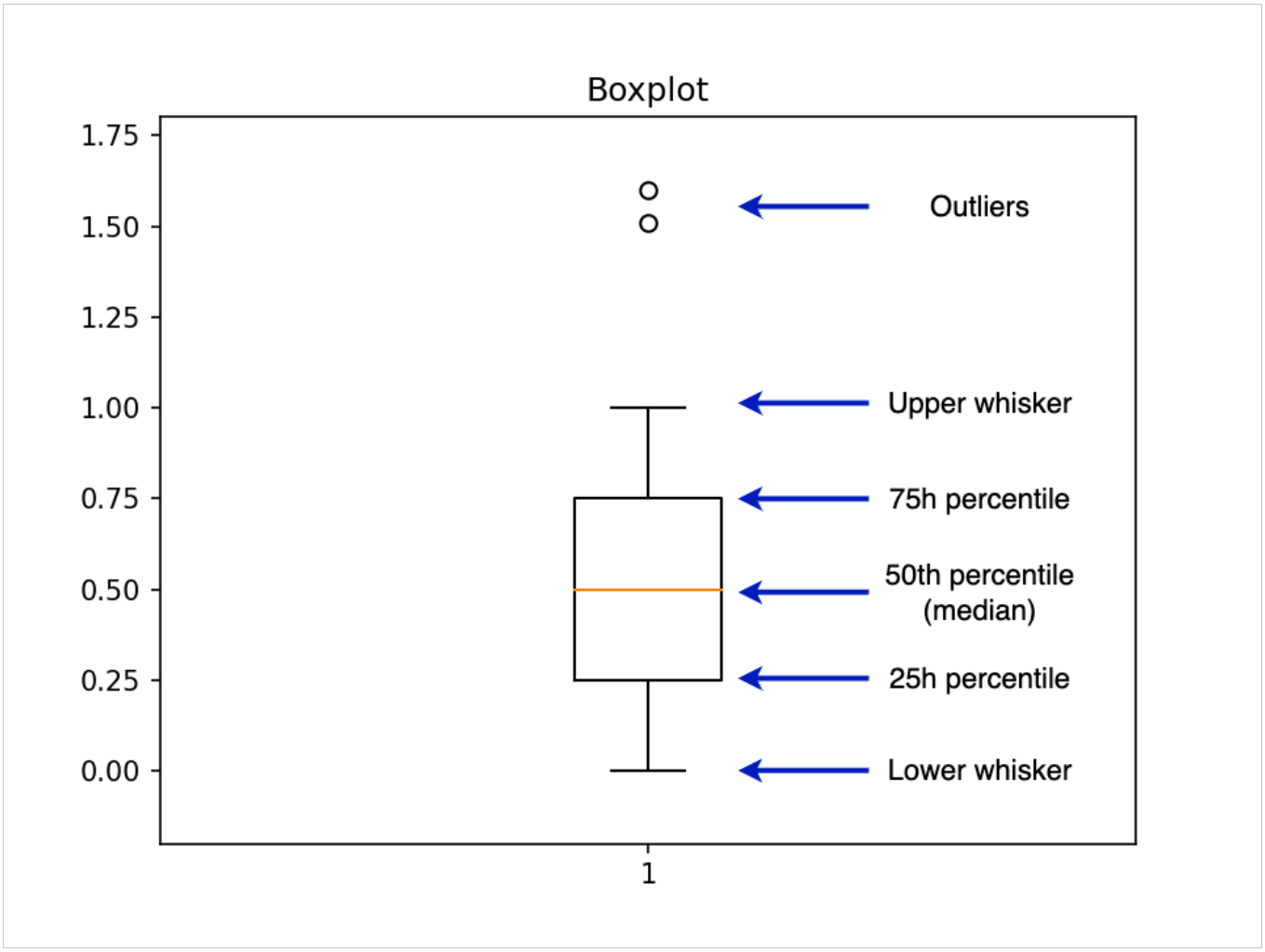
When facing this challenge, we might consider using a distributed framework; however, this adds so much complexity to the process, and it only partially solves the problem since we'd need to write code to compute the statistics in a distributed fashion. Alternatively, we might consider getting a larger machine, a relatively straightforward (but expensive!) approach if we can access cloud resources. However, this still requires us to move our data, set up a new environment, etc. Fortunately for us, there's DuckDB!

DuckDB: A Highly Scalable Backend for Statistical Visualizations

When using functions such as `hist` (histogram) or `boxplot`, `matplotlib` performs two steps:

1. Compute summary statistics
2. Plot data

For example, `boxplot` calls another function called `boxplot_stats` that returns the statistics required to draw the plot. To create a boxplot, we need several summary statistics, such as the 25th percentile, 50th percentile, and 75th percentile. The following diagram shows a boxplot along with the labels for each part:



The bottleneck in the `pandas` + `matplotlib` approach is the `boxplot_stats` function since it requires a `numpy.array` or `pandas.Series` as an input, forcing us to load all our data into memory. However, we can implement a new version of `boxplot_stats` that pushes the data aggregation step to another analytical engine.

We chose DuckDB because it's extremely powerful and easy to use. There is no need to spin up a server or manage complex configurations: install it with `pip install`, point it to your data files, and that's it; you can start aggregating millions and millions of data points in no time!

You can see the full implementation here; essentially, we translated matplotlib's `boxplot_stats` from Python into SQL. For example, the following query will compute the three percentiles we need: 25th, 50th (median), and 75th:

```
%load_ext sql
%sql duckdb://

%%sql
-- We calculate the percentiles all at once and
-- then convert from list format into separate columns
-- (Improving performance by reducing duplicate work)
WITH stats AS (
  SELECT
```

```
    percentile_disc([0.25, 0.50, 0.75]) WITHIN GROUP
      (ORDER BY "trip_distance") AS percentiles
  FROM 'yellow_tripdata_2022-01.parquet'
)
SELECT
  percentiles[1] AS q1,
  percentiles[2] AS median,
  percentiles[3] AS q3
FROM stats;
```

Once we compute all the statistics, we call the `bxp` function, which draws the boxplot from the input statistics.

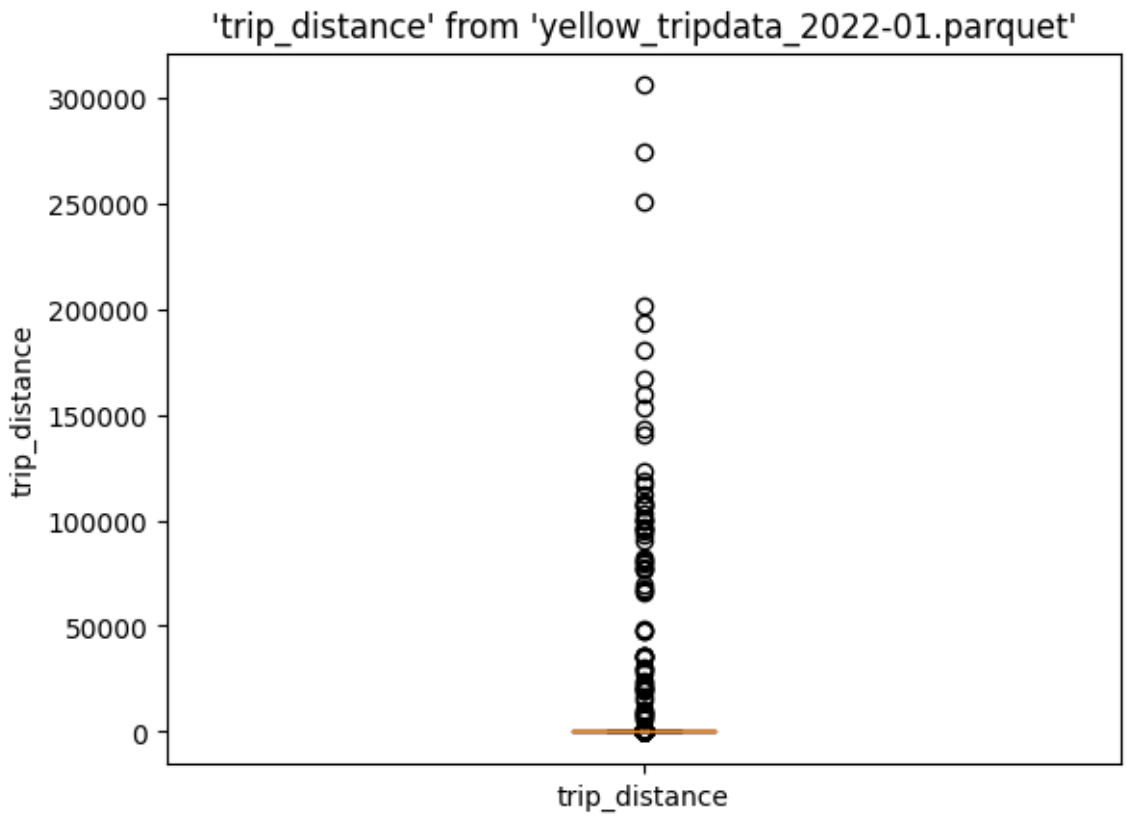
This process is already implemented in JupySQL, and you can create a boxplot with the `%sqlplot boxplot` command. Let's see how. But first, let's check how much memory we're using, so we can compare it to the `pandas` version:

```
memory_used()
```

```
Memory used: 1351 MB
```

Let's create the boxplot:

```
%sqlplot boxplot --table yellow_tripdata_2022-01.parquet --column trip_distance
```



Again, we see all these outliers. Let's compute the cutoff value:

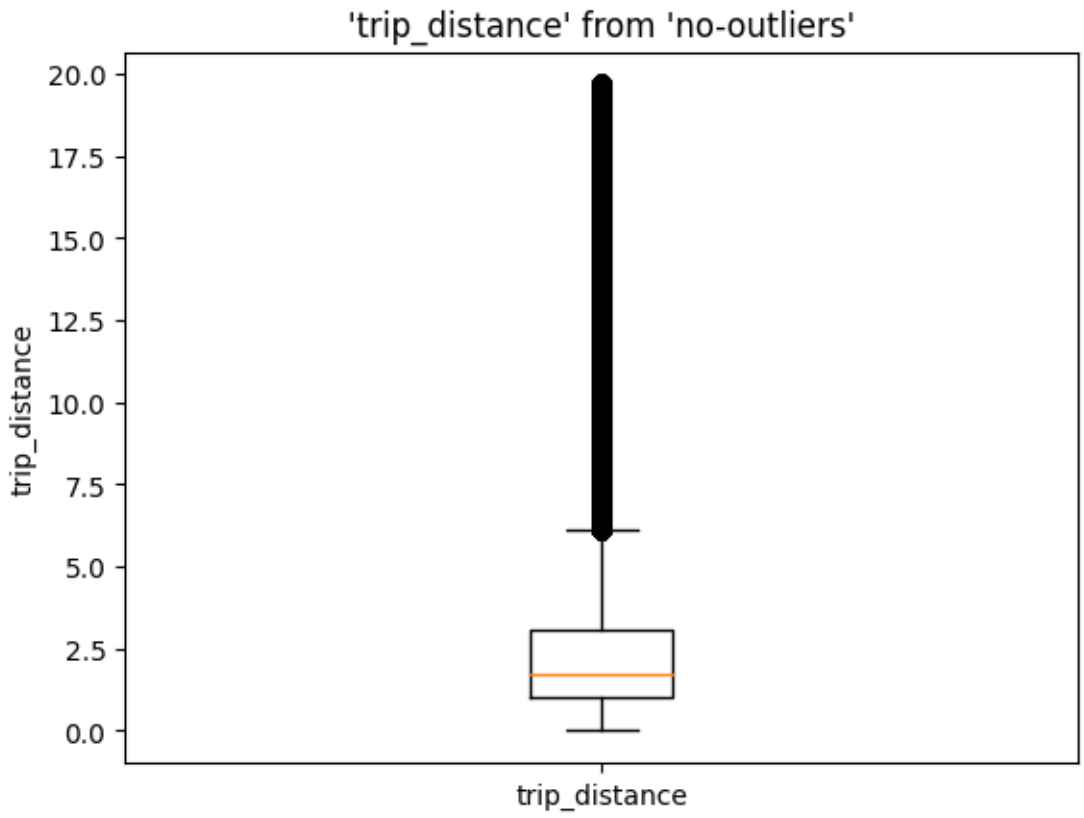
```
%%sql
SELECT percentile_disc(0.99) WITHIN GROUP (ORDER BY trip_distance)
FROM 'yellow_tripdata_2022-01.parquet'
```

Let's define a query that filters out the top 1% of observations. The `--save` option allows us to store this SQL expression and we choose not to execute it.

```
%%sql --save no-outliers --no-execute
SELECT *
FROM 'yellow_tripdata_2022-01.parquet'
WHERE trip_distance < 19.7;
```

We can now use `no-outliers` in our `%sqlplot boxplot` command:

```
%sqlplot boxplot --table no-outliers --column trip_distance --with no-outliers
```




```
memory_used()
```

```
Memory used: 1375 MB
```

Memory usage remained pretty much the same (23 MB difference, mostly due to the newly imported modules). Since we're relying on DuckDB for the data aggregation step, the SQL engine takes care of loading, aggregating, and freeing up memory as soon as we're done; this is much more efficient than loading all our data at the same time and keeping unwanted data copies!

Using DuckDB to Compute Histogram Statistics

We can extend our recipe to other statistical visualizations, such as histograms.

A histogram allows us to visualize the distribution of a dataset, enabling us to find patterns such as modality, outliers, range of values, etc. Like with the boxplot, when using `pandas` + `matplotlib`, creating a histogram involves loading all our data at once into memory; then, `matplotlib` aggregates and plots it.

In our case, we'll push the aggregation to DuckDB, which will compute the bin positions (X-axis) and heights (Y-axis), then we'll pass this to maptlotlib's `bar` function to create the histogram.

The implementation involves two steps.

First, given the number of bins chosen by the user (`N_BINS`), we compute the `BIN_SIZE`:

```
%%sql
SELECT (max(trip_distance) - min(trip_distance)) / N_BINS
FROM 'yellow_tripdata_2022-01.parquet';
```

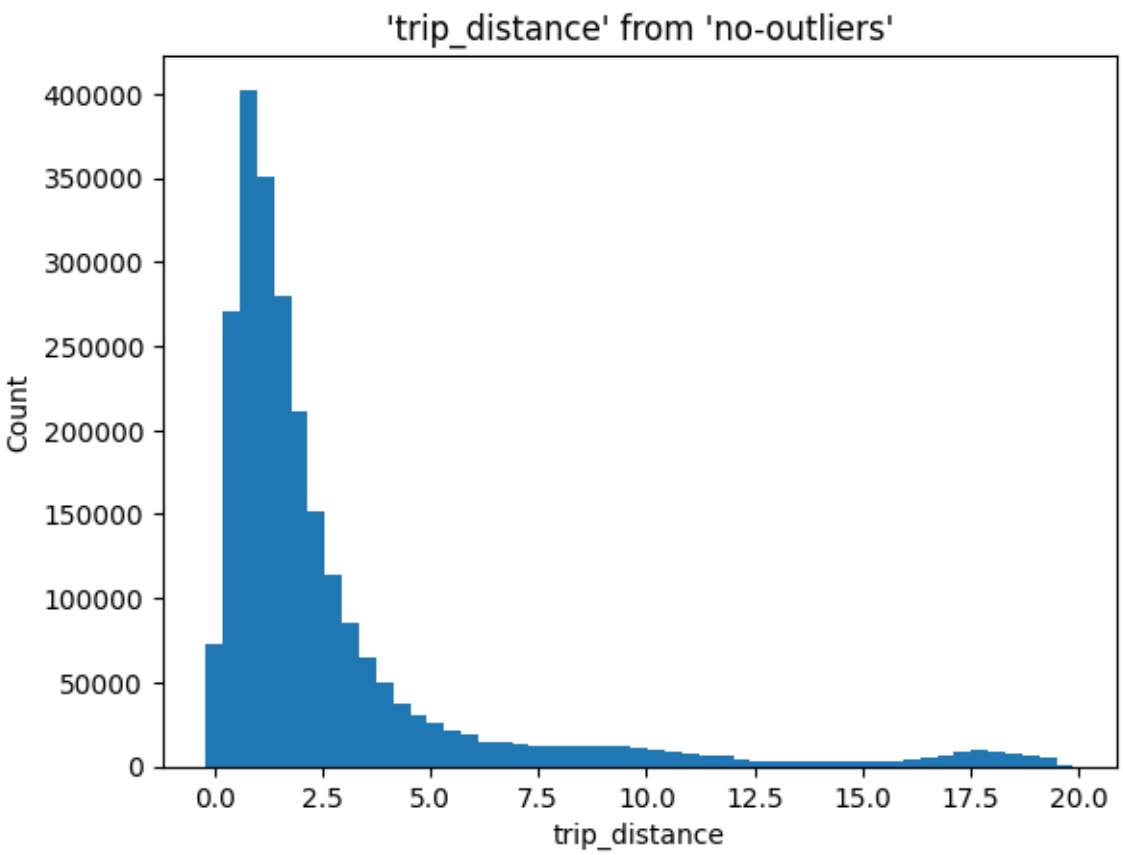
Then, using the `BIN_SIZE`, we find the number of observations that fall into each bin:

```
%%sql
SELECT
    floor("trip_distance" / BIN_SIZE) * BIN_SIZE,
    count(*) AS count
FROM 'yellow_tripdata_2022-01.parquet'
GROUP BY 1
ORDER BY 1;
```

The intuition for the second query is as follows: given that we have `N_BINS`, the `floor("trip_distance" / BIN_SIZE)` portion will assign each observation to their corresponding bin (1, 2, ..., `N_BINS`), then, we multiply by the bin size to get the value in the X axis, while the count represents the value in the Y axis. Once we have that, we call the `bar` plotting function.

All these steps are implemented in the `%sqplot histogram` command:

```
%sqlplot histogram --table no-outliers --column trip_distance --with no-outliers
```



Final Thoughts

This blog post demonstrated a powerful approach for plotting large datasets powered using JupySQL and DuckDB. If you need to visualize large datasets, DuckDB offers unmatched simplicity and flexibility!

At Ploomber, we're working on building a full-fledged SQL client for Jupyter! Exciting features like automated dataset profiling, autocompletion, and more are coming! So keep an eye on updates! If there are features you think we should add to offer the best SQL experience in Jupyter, please open an issue!

JupySQL is an actively maintained fork of `ipython-sql`, and it keeps full compatibility with it. If you want to learn more, check out the GitHub repository and the documentation.

Try It Out

To try it yourself, check out this collab notebook, or here's a snippet you can paste into Jupyter:

```
from urllib.request import urlretrieve

urlretrieve("https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2022-01.parquet",
            "yellow_tripdata_2022-01.parquet")

%pip install jupysql duckdb-engine --quiet
%load_ext sql
%sql duckdb://
%sqlplot boxplot --table yellow_tripdata_2022-01.parquet --column trip_distance
```

Note: the commands that begin with % or %% will only work on Jupyter/IPython. If you want to try this in a regular Python session, check out the Python API.

Shredding Deeply Nested JSON, One Vector at a Time

Publication date: 2023-03-03

Author: Laurens Kuiper

TL;DR: We recently improved DuckDB's JSON extension so JSON files can be directly queried as if they were tables.

> We updated this blog post in December 2024 to reflect the changes in DuckDB's JSON syntax.

DuckDB has a JSON extension that can be installed and loaded through SQL:

```
INSTALL 'json';
LOAD 'json';
```

The JSON extension supports various functions to create, read, and manipulate JSON strings. These functions are similar to the JSON functionality provided by other databases such as PostgreSQL and MySQL. DuckDB uses yyjson internally to parse JSON, a high-performance JSON library written in ANSI C. Many thanks to the yyjson authors and contributors!

Besides these functions, DuckDB is now able to read JSON directly! This is done by automatically detecting the types and column names, then converting the values within the JSON to DuckDB's vectors. The automated schema detection dramatically simplifies working with JSON data and subsequent queries on DuckDB's vectors are significantly faster!

Reading JSON Automatically with DuckDB

Since version 0.7.0, DuckDB supports JSON table functions. To demonstrate these, we will read `todos.json`, a fake TODO list containing 200 fake TODO items (only the first two items are shown):

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
  },
  {
    "userId": 1,
    "id": 2,
    "title": "quis ut nam facilis et officia qui",
    "completed": false
  },
  ...
]
```

Each TODO item is an entry in the JSON array, but in DuckDB, we'd like a table where each entry is a row. This is as easy as:

```
SELECT * FROM 'todos.json' LIMIT 5;
```

userId	id	title	completed
1	1	delectus aut autem	false
1	2	quis ut nam facilis et officia qui	false
1	3	fugiat veniam minus	false
1	4	et porro tempora	true
1	5	laboriosam mollitia et enim quasi adipisci quia provident illum	false

Finding out which user completed the most TODO items is as simple as:

```
SELECT userId, sum(completed::INTEGER) total_completed
FROM 'todos.json'
GROUP BY userId
ORDER BY total_completed DESC
LIMIT 1;
```

userId	total_completed
5	12

Under the hood, DuckDB recognizes the `.json` file extension in `'todos.json'`, and calls `read_json('todos.json')` instead. This function is similar to our `read_csv` function, which automatically infers column names and types for CSV files.

Like our other table functions, `read_json` supports reading multiple files by passing a list, e.g., `read_json(['file1.json', 'file2.json'])`, but also globbing, e.g., `read_json('file*.json')`. DuckDB will read multiple files in parallel.

Newline Delimited JSON

Not all JSON adheres to the format used in `todos.json`, which is an array of 'records'. Newline-delimited JSON, or NDJSON, stores each row on a new line. DuckDB also supports reading (and writing!) this format. First, let's write our TODO list as NDJSON:

```
COPY (SELECT * FROM 'todos.json') to 'todos2.json';
```

Again, DuckDB recognizes the `.json` suffix in the output file and automatically infers that we mean to use `(FORMAT json)`. The created file looks like this (only the first two records are shown):

```
{"userId":1,"id":1,"title":"delectus aut autem","completed":false}
{"userId":1,"id":2,"title":"quis ut nam facilis et officia qui","completed":false}
...
```

DuckDB can read this file in exactly the same way as the original one:

```
SELECT * FROM 'todos2.json';
```

If your JSON file is newline-delimited, DuckDB can parallelize reading. This can be specified by calling `read_ndjson` or passing the `records = true` parameter to `read_json`:

```
SELECT * FROM read_ndjson('todos2.json');
SELECT * FROM read_json('todos2.json', records = true);
```

You can also set `records = auto` to auto-detect whether the JSON file is newline-delimited.

Other JSON Formats

When the `read_json` function is used directly, the format of the JSON can be specified using the `format` parameter. This parameter defaults to `'auto'`, which tells DuckDB to infer what kind of JSON we are dealing with. The first `format` is `'array'`, while the second is `'nd'`. This can be specified like so:

```
SELECT * FROM read_json('todos.json', format = 'array');
SELECT * FROM read_json('todos2.json', format = 'nd');
```

Another supported format is `unstructured`. With this format, records are not required to be a JSON object but can also be a JSON array, string, or anything supported in JSON.

Manual Schemas

What you may also have noticed is the `auto_detect` parameter. This parameter tells DuckDB to infer the schema, i.e., determine the names and types of the returned columns. These can manually be specified like so:

```
SELECT *
FROM read_json(
    'todos.json',
    columns = {userId: 'INTEGER', id: 'INTEGER', title: 'VARCHAR', completed: 'BOOLEAN'},
    format = 'array'
);
```

You don't have to specify all fields, just the ones you're interested in:

```
SELECT *
FROM read_json(
    'todos.json',
    columns = {userId: 'INTEGER', completed: 'BOOLEAN'},
    format = 'array'
);
```

Now that we know how to use the new DuckDB JSON table functions let's dive into some analytics!

GitHub Archive Examples

GH Archive is a project to record the public GitHub timeline, archive it, and make it easily accessible for further analysis. Every hour, a GZIP compressed, newline-delimited JSON file containing all public events on GitHub is uploaded. I downloaded a whole day (2023-02-08) of activity using `wget` and stored the 24 files (starting with `2023-02-08-0.json.gz`) in a directory called `gharchive_gz`. You can get the full day's archive as `gharchive-2023-02-08.zip`:

Keep in mind that the data is compressed:

```
du -sh gharchive_gz
```

```
2.3G  gharchive_gz
```

Decompressed, one day's worth of GitHub activity amounts to more than 18 GB of JSON.

```
gunzip -dc gharchive_gz/* | wc -c
```

```
18396198934
```

To get a feel of what the data looks like, we run the following query:

```
SELECT json_group_structure(json)
FROM (
    SELECT *
    FROM read_ndjson_objects('gharchive_gz/*.json.gz')
    LIMIT 2048
);
```

Here, we use our `read_ndjson_objects` function, which reads the JSON objects in the file as raw JSON, i.e., as strings. The query reads the first 2048 records of JSON from the JSON files `gharchive_gz` directory and describes the structure. You can also directly query the JSON files from GH Archive using DuckDB's `httpfs` extension, but we will be querying the files multiple times, so it is better to download them in this case.

> **Tip.** In the CLI client, you can use `.mode line` to make the output easier to read.

I formatted the result using an online JSON formatter & validator:

```
{
   "id":"VARCHAR",
   "type":"VARCHAR",
   "actor":{
      "id":"UBIGINT",
      "login":"VARCHAR",
      "display_login":"VARCHAR",
      "gravatar_id":"VARCHAR",
      "url":"VARCHAR",
      "avatar_url":"VARCHAR"
   },
   "repo":{
      "id":"UBIGINT",
      "name":"VARCHAR",
      "url":"VARCHAR"
   },
   "payload":{"..."},
   "public":"BOOLEAN",
   "created_at":"VARCHAR",
   "org":{
      "id":"UBIGINT",
      "login":"VARCHAR",
      "gravatar_id":"VARCHAR",
      "url":"VARCHAR",
      "avatar_url":"VARCHAR"
   }
}
```

I left "payload" out because it consists of deeply nested JSON, and its formatted structure takes up more than 1000 lines!

So, how many records are we dealing with exactly? Let's count it using DuckDB:

```
SELECT count(*) AS count
FROM 'gharchive_gz/*.json.gz';
```

count
4434953

That's around 4.4M daily events, which amounts to almost 200K events per hour. This query takes around 7.3 seconds on my laptop, a 2020 MacBook Pro with an M1 chip and 16 GB of memory. This is the time it takes to decompress the GZIP compression and parse every JSON record.

To see how much time is spent decompressing GZIP in the query, I also created a `gharchive` directory containing the same data but uncompressed. Running the same query on the uncompressed data takes around 5.4 seconds, almost 2 seconds faster. We got faster, but we also read more than 18 GB of data from storage, as opposed to 2.3 GB when it was compressed. So, this comparison really depends on the speed of your storage. I prefer to keep the data compressed.

As a side note, the speed of this query really shows how fast yyjson is!

So, what kind of events are in the GitHub data?

```
SELECT type, count(*) AS count
FROM 'gharchive_gz/*.json.gz'
GROUP BY type
ORDER BY count DESC;
```

type	count
PushEvent	2359096
CreateEvent	624062
PullRequestEvent	366090
IssueCommentEvent	238660
WatchEvent	231486
DeleteEvent	154383
PullRequestReviewEvent	131107
IssuesEvent	88917
PullRequestReviewCommentEvent	79540
ForkEvent	64233
CommitCommentEvent	36823
ReleaseEvent	23004
MemberEvent	14872
PublicEvent	14500
GollumEvent	8180

This query takes around 7.4 seconds, not much more than the `count(*)` query. So as we can see, data analysis is very fast once everything has been decompressed and parsed.

The most common event type is the `PushEvent`, taking up more than half of all events, unsurprisingly, which is people pushing their committed code to GitHub. The least common event type is the `GollumEvent`, taking up less than 1% of all events, which is a creation or update of a wiki page.

If we want to analyze the same data multiple times, decompressing and parsing every time is redundant and slows down the analysis. Instead, we can create a DuckDB table like so:

```
CREATE TABLE events AS
    SELECT * EXCLUDE (payload)
    FROM 'gharchive_gz/*.json.gz';
```

This takes around 9 seconds if you're using an in-memory database. If you're using an on-disk database, this takes around 13 seconds and results in a database size of 444 MB. When using an on-disk database, DuckDB ensures the table is persistent and performs all kinds of compression. Note that we have temporarily ignored the `payload` field using the convenient `EXCLUDE` clause.

To get a feel of what we've read, we can ask DuckDB to describe the table:

```
DESCRIBE SELECT * FROM events;
```

This gives us the following:

cid	name	type	notnull	dflt_value	pk
0	id	BIGINT	false		false
1	type	VARCHAR	false		false
2	actor	STRUCT(id UBIGINT, login VARCHAR, display_login VARCHAR, gravatar_id VARCHAR, url VARCHAR, avatar_url VARCHAR)	false		false
3	repo	STRUCT(id UBIGINT, name VARCHAR, url VARCHAR)	false		false
4	public	BOOLEAN	false		false
5	created_at	TIMESTAMP	false		false
6	org	STRUCT(id UBIGINT, login VARCHAR, gravatar_id VARCHAR, url VARCHAR, avatar_url VARCHAR)	false		false

As we can see, the `"actor"`, `"repo"` and `"org"` fields, which are JSON objects, have been converted to DuckDB structs. The `"id"` column was a string in the original JSON but has been converted to a `BIGINT` by DuckDB's automatic type detection. DuckDB can also detect a few different `DATE`/`TIMESTAMP` formats within JSON strings, as well as `TIME` and `UUID`.

Now that we created the table, we can analyze it like any other DuckDB table! Let's see how much activity there was in the `duckdb/duckdb` GitHub repository on this specific day:

```
SELECT type, count(*) AS count
FROM events
WHERE repo.name = 'duckdb/duckdb'
GROUP BY type
ORDER BY count DESC;
```

type	count
PullRequestEvent	35
IssueCommentEvent	30
WatchEvent	29
PushEvent	15

type	count
PullRequestReviewEvent	14
IssuesEvent	9
PullRequestReviewCommentEvent	7
ForkEvent	3

That's a lot of pull request activity! Note that this doesn't mean that 35 pull requests were opened on this day, activity within a pull request is also counted. If we search through the pull requests for that day, we see that there are only 15. This is more activity than normal because most of the DuckDB developers were busy fixing bugs for the 0.7.0 release.

Now, let's see who was the most active:

```
SELECT actor.login, count(*) AS count
FROM events
WHERE repo.name = 'duckdb/duckdb'
  AND type = 'PullRequestEvent'
GROUP BY actor.login
ORDER BY count desc
LIMIT 5;
```

login	count
Mytherin	19
Mause	4
carlopi	3
Tmonster	2
lnkuiper	2

As expected, Mark Raasveldt (Mytherin, co-founder of DuckLabs) was the most active! My activity (lnkuiper, software engineer at DuckLabs) also shows up.

Handling Inconsistent JSON Schemas

So far, we have ignored the "payload" of the events. We ignored it because the contents of this field are different based on the type of event. We can see how they differ with the following query:

```
SELECT json_group_structure(payload) AS structure
FROM (
    SELECT *
    FROM read_json(
        'gharchive_gz/*.json.gz',
        columns = {
            id: 'BIGINT',
            type: 'VARCHAR',
            actor: 'STRUCT(id UBIGINT,
                           login VARCHAR,
                           display_login VARCHAR,
                           gravatar_id VARCHAR,
                           url VARCHAR,
                           avatar_url VARCHAR)',
            repo: 'STRUCT(id UBIGINT, name VARCHAR, url VARCHAR)',
            payload: 'JSON',
            public: 'BOOLEAN',
```

```
            created_at: 'TIMESTAMP',
            org: 'STRUCT(id UBIGINT, login VARCHAR, gravatar_id VARCHAR, url VARCHAR, avatar_url VARCHAR)'
        },
        records = true
    )
    WHERE type = 'WatchEvent'
    LIMIT 2048
);
```

structure
{"action":"VARCHAR"}

The `"payload"` field is simple for events of type `WatchEvent`. However, if we change the type to `PullRequestEvent`, we get a JSON structure of more than 500 lines when formatted with a JSON formatter. We don't want to look through all those fields, so we cannot use our automatic schema detection, which will try to get them all. Instead, we can manually supply the structure of the fields we're interested in. DuckDB will skip reading the other fields. Another approach is to store the `"payload"` field as DuckDB's JSON data type and parse it at query time (see the example later in this post!).

I stripped down the JSON structure for the `"payload"` of events with the type `PullRequestEvent` to the things I'm actually interested in:

```
{
   "action":"VARCHAR",
   "number":"UBIGINT",
   "pull_request":{
      "url":"VARCHAR",
      "id":"UBIGINT",
      "title":"VARCHAR",
      "user":{
         "login":"VARCHAR",
         "id":"UBIGINT",
      },
      "body":"VARCHAR",
      "created_at":"TIMESTAMP",
      "updated_at":"TIMESTAMP",
      "assignee":{
         "login":"VARCHAR",
         "id":"UBIGINT",
      },
      "assignees":[
         {
            "login":"VARCHAR",
            "id":"UBIGINT",
         }
      ],
  }
}
```

This is technically not valid JSON because there are trailing commas. However, we try to allow trailing commas wherever possible in DuckDB, including JSON!

We can now plug this into the `columns` parameter of `read_json`, but we need to convert it to a DuckDB type first. I'm lazy, so I prefer to let DuckDB do this for me:

```
SELECT typeof(
    json_transform('{}', '{
        "action":"VARCHAR",
        "number":"UBIGINT",
        "pull_request":{
          "url":"VARCHAR",
          "id":"UBIGINT",
```

```
          "title":"VARCHAR",
          "user":{
              "login":"VARCHAR",
              "id":"UBIGINT",
          },
          "body":"VARCHAR",
          "created_at":"TIMESTAMP",
          "updated_at":"TIMESTAMP",
          "assignee":{
              "login":"VARCHAR",
              "id":"UBIGINT",
          },
          "assignees":[
              {
                "login":"VARCHAR",
                "id":"UBIGINT",
              }
          ],
        }
    }')
);
```

This gives us back a DuckDB type that we can plug the type into our function! Note that because we are not auto-detecting the schema, we have to supply `timestampformat` to be able to parse the timestamps correctly. The key `"user"` must be surrounded by quotes because it is a reserved keyword in SQL:

```
CREATE OR REPLACE TABLE pr_events AS
    SELECT *
    FROM read_json(
        'gharchive_gz/*.json.gz',
        columns = {
            id: 'BIGINT',
            type: 'VARCHAR',
            actor: 'STRUCT(id UBIGINT,
                           login VARCHAR,
                           display_login VARCHAR,
                           gravatar_id VARCHAR,
                           url VARCHAR,
                           avatar_url VARCHAR)',
            repo: 'STRUCT(id UBIGINT, name VARCHAR, url VARCHAR)',
            payload: 'STRUCT(
                        action VARCHAR,
                        number UBIGINT,
                        pull_request STRUCT(
                          url VARCHAR,
                          id UBIGINT,
                          title VARCHAR,
                          "user" STRUCT(
                            login VARCHAR,
                            id UBIGINT
                          ),
                          body VARCHAR,
                          created_at TIMESTAMP,
                          updated_at TIMESTAMP,
                          assignee STRUCT(login VARCHAR, id UBIGINT),
                          assignees STRUCT(login VARCHAR, id UBIGINT)[]
                        )
                      )',
            public: 'BOOLEAN',
            created_at: 'TIMESTAMP',
            org: 'STRUCT(id UBIGINT, login VARCHAR, gravatar_id VARCHAR, url VARCHAR, avatar_url VARCHAR)'
        },
        format = 'newline_delimited',
        records = true,
        timestampformat = '%Y-%m-%dT%H:%M:%SZ'
```

```
    )
    WHERE type = 'PullRequestEvent';
```

This query completes in around 36 seconds with an on-disk database (resulting size is 478 MB) and 9 seconds with an in-memory database. If you don't care about preserving insertion order, you can speed the query up with this setting:

```
SET preserve_insertion_order = false;
```

With this setting, the query completes in around 27 seconds with an on-disk database and 8.5 seconds with an in-memory database. The difference between the on-disk and in-memory case is quite substantial here because DuckDB has to compress and persist much more data.

Now we can analyze pull request events! Let's see what the maximum number of assignees is:

```
SELECT max(length(payload.pull_request.assignees)) AS max_assignees
FROM pr_events;
```

max_assignees
10

That's a lot of people reviewing a single pull request!

We can check who was assigned the most:

```
WITH assignees AS (
    SELECT payload.pull_request.assignee.login AS assignee
    FROM pr_events
    UNION ALL
    SELECT unnest(payload.pull_request.assignees).login AS assignee
    FROM pr_events
)
SELECT assignee, count(*) AS count
FROM assignees
WHERE assignee NOT NULL
GROUP BY assignee
ORDER BY count DESC
LIMIT 5;
```

assignee	count
poad	494
vinayakkulkarni	268
tmtmtmtm	198
fisker	98
icemac	84

That's a lot of assignments – although I suspect there are duplicates in here.

Storing as JSON to Parse at Query Time

Specifying the JSON schema of the `"payload"` field was helpful because it allowed us to directly analyze what is there, and subsequent queries are much faster. Still, it can also be quite cumbersome if the schema is complex. If you don't want to specify the schema of a field, you can set the type as `'JSON'`:

```
CREATE OR REPLACE TABLE pr_events AS
    SELECT *
    FROM read_json(
        'gharchive_gz/*.json.gz',
        columns = {
            id: 'BIGINT',
            type: 'VARCHAR',
            actor: 'STRUCT(id UBIGINT,
                           login VARCHAR,
                           display_login VARCHAR,
                           gravatar_id VARCHAR,
                           url VARCHAR,
                           avatar_url VARCHAR)',
            repo: 'STRUCT(id UBIGINT, name VARCHAR, url VARCHAR)',
            payload: 'JSON',
            public: 'BOOLEAN',
            created_at: 'TIMESTAMP',
            org: 'STRUCT(id UBIGINT, login VARCHAR, gravatar_id VARCHAR, url VARCHAR, avatar_url VARCHAR)'
        },
        format = 'newline_delimited',
        records = true,
        timestampformat = '%Y-%m-%dT%H:%M:%SZ'
    )
    WHERE type = 'PullRequestEvent';
```

This will load the `"payload"` field as a JSON string, and we can use DuckDB's JSON functions to analyze it when querying. For example:

```
SELECT DISTINCT payload->>'action' AS action, count(*) AS count
FROM pr_events
GROUP BY action
ORDER BY count DESC;
```

The `->>` arrow is short-hand for our `json_extract_string` function. Creating the entire `"payload"` field as a column with type `JSON` is not the most efficient way to get just the `"action"` field, but this example is just to show the flexibility of `read_json`. The query results in the following table:

action	count
opened	189096
closed	174914
reopened	2080

As we can see, only a few pull requests have been reopened.

Conclusion

DuckDB tries to be an easy-to-use tool that can read all kinds of data formats. In the 0.7.0 release, we have added support for reading JSON. JSON comes in many formats and all kinds of schemas. DuckDB's rich support for nested types (`LIST`, `STRUCT`) allows it to fully “shred” the JSON to a columnar format for more efficient analysis.

We are excited to hear what you think about our new JSON functionality. If you have any questions or suggestions, please reach out to us on Discord or GitHub!

The Return of the H2O.ai Database-like Ops Benchmark

Publication date: 2023-04-14

Author: Tom Ebergen

TL;DR: We've resurrected the H2O.ai database-like ops benchmark with up to date libraries and plan to keep re-running it.

Skip directly to the results

> We published a new blog post on the H2O.ai benchmark in November 2023 and improved the benchmark setup for reproducibility. For details, see the new post: "Updates to the H2O.ai db-benchmark!"

The H2O.ai Database-like Ops Benchmark is a well-known benchmark in the data analytics and R community. The benchmark measures the groupby and join performance of various analytical tools like data.table, Polars, dplyr, ClickHouse, DuckDB and more. Since July 2nd 2021, the benchmark has been dormant, with no result updates or maintenance. Many of the analytical systems measured in the benchmark have since undergone substantial improvements, leaving many of the maintainers curious as to where their analytical tool ranks on the benchmark.

DuckDB has decided to give the H2O.ai benchmark new life and maintain it for the foreseeable future. One reason the DuckDB project has decided to maintain the benchmark is because DuckDB has had 10 new minor releases since the most recent published results on July 2nd, 2021. After managing to run parts of the benchmark on a r3-8xlarge AWS box, DuckDB ranked as a top performer on the benchmark. Additionally, the DuckDB project wants to demonstrate it's commitment to performance by consistently comparing DuckDB with other analytical systems. While DuckDB delivers differentiated ease of use, raw performance and scalability are critically important for solving tough problems fast. Plus, just like many of our fellow data folks, we have a need for speed. Therefore, the decision was made to fork the benchmark, modernize underlying dependencies and run the benchmark on the latest versions of the included systems. You can find the repository on GitHub.

The results of the new benchmark are very interesting, but first a quick summary of the benchmark and what updates took place.

The H2O.ai Database-like Ops Benchmark

There are 5 basic grouping tests and 5 advanced grouping tests. The 10 grouping queries all focus on a combination of the following

- Low cardinality (a few big groups)
- High cardinality (lots of very small groups)
- Grouping integer types
- Grouping string types

Each query is run only twice with both results being reported. This way we can see the performance of a cold run and any effects data caching may have. The idea is to avoid reporting any potential "best" results on a hot system. Data analysts only need to run a query once to get their answer. No one drives to the store a second time to get another litre of milk faster.

The time reported is the sum of the time it takes to run all 5 queries twice.

More information about the specific queries can be found below.

The Data and Queries

The queries have not changed since the benchmark went dormant. The data is generated in a rather simple manner. Inspecting the datagen files you can see that the columns are generated with small, medium, and large groups of char and int values. Similar generation logic applies to the join data generation.

Query	SQL	Objective
groupby #1	`SELECT id1, sum(v1) AS v1 FROM tbl GROUP BY id1`	Sum over large cardinality groups, grouped by varchar
groupby #2	`SELECT id1, id2, sum(v1) AS v1 FROM tbl GROUP BY id1, id2`	Sum over medium cardinality groups, grouped by varchars
groupby #3	`SELECT id3, sum(v1) AS v1, mean(v3) AS v3 FROM tbl GROUP BY id3`	Sum and mean over many small cardinality groups, grouped by varchar
groupby #4	`SELECT id4, mean(v1) AS v1, mean(v2) AS v2, mean(v3) AS v3 FROM tbl GROUP BY id4`	Mean over many large cardinality groups, grouped by integer
groupby #5	`SELECT id6, sum(v1) AS v1, sum(v2) AS v2, sum(v3) AS v3 FROM tbl GROUP BY id6`	Sum over many small groups, grouped by integer
advanced groupby #1	`SELECT id4, id5, quantile_cont(v3, 0.5) AS median_v3, stddev(v3) AS sd_v3 FROM tbl GROUP BY id4, id5`	`quantile_cont` over medium cardinality group, grouped by integers
advanced groupby #2	`SELECT id3, max(v1)-min(v2) AS range_v1_v2 FROM tbl GROUP BY id3`	Range selection over small cardinality groups, grouped by integer
advanced groupby #3	`SELECT id6, v3 AS largest2_v3 FROM (SELECT id6, v3, row_number() OVER (PARTITION BY id6 ORDER BY v3 DESC) AS order_v3 FROM x WHERE v3 IS NOT NULL) sub_query WHERE order_v3 <= 2`	Advanced group by query
advanced groupby #4	`SELECT id2, id4, pow(corr(v1, v2), 2) AS r2 FROM tbl GROUP BY id2, id4`	Arithmetic over medium sized groups, grouped by varchar, integer.
advanced groupby #5	`SELECT id1, id2, id3, id4, id5, id6, sum(v3) AS v3, count(*) AS count FROM tbl GROUP BY id1, id2, id3, id4, id5, id6`	Many small groups, the number of groups is the cardinality of the dataset
join #1	`SELECT x.*, small.id4 AS small_id4, v2 FROM x JOIN small USING (id1)`	Joining a large table (x) with a small-sized table on integer type
join #2	`SELECT x.*, medium.id1 AS medium_id1, medium.id4 AS medium_id4, medium.id5 AS medium_id5, v2 FROM x JOIN medium USING (id2)`	Joining a large table (x) with a medium-sized table on integer type
join #3	`SELECT x.*, medium.id1 AS medium_id1, medium.id4 AS medium_id4, medium.id5 AS medium_id5, v2 FROM x LEFT JOIN medium USING (id2)`	Left join a large table (x) with a medium-sized table on integer type
join #4	`SELECT x.*, medium.id1 AS medium_id1, medium.id2 AS medium_id2, medium.id4 AS medium_id4, v2 FROM x JOIN medium USING (id5)`	Join a large table (x) with a medium table on varchar type
join #5	`SELECT x.*, big.id1 AS big_id1, big.id2 AS big_id2, big.id4 AS big_id4, big.id5 AS big_id5, big.id6 AS big_id6, v2 FROM x JOIN big USING (id3)`	Join a large table (x) with a large table on integer type.

You can find more information about the queries in the Efficiency of Data Processing slides.

Modifications to the Benchmark & Hardware

No modifications have been made to the queries or the data generation. Some scripts required minor modifications so that the current version of the library could be run. The hardware used is slightly different as the exact AWS offering the benchmark previously used is no longer available. Base libraries have been updated as well. GPU libraries were not tested.

AWS is an m4.10xlarge instance:

- CPU model: Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40 GHz
- CPU cores: 40
- RAM model: Unknown
- Memory: 160 GB
- NO GPU specifications
- R upgraded, 4.0.0 -> 4.2.2
- Python upgraded 3.[6|7] -> 3.10

Changes Made to Install Scripts of Other Systems

Pandas, Polars, Dask, and ClickHouse required changes to their setup/install scripts. The changes were relatively minor consisting mostly of syntax updates and data ingestion updates. Data ingestion did not affect the reporting timing results.

Results

You can also look at the results. DuckDB's timings have improved significantly since v0.2.7 (released over two years ago). A major contributor to our increased performance is parallel grouped aggregation, merged in March 2022, and parallel result set materialization. In addition, DuckDB now supports enum types, which makes DuckDB `group by` aggregation even faster. Improvements to the out-of-core hash join were merged as well, further improving the performance of our joins.

Questions about Certain Results?

Some solutions may report internal errors for some queries. Feel free to investigate the errors by using the `repro.sh` scripts and file a GitHub issue to resolve any confusion. In addition, there are many areas in the code where certain query results are automatically nullified. If you believe that is the case for a query for your system or if you have any other questions, you can create a GitHub issue to discuss.

Maintenance Plan

DuckDB will continue to maintain this benchmark for the foreseeable future. The process for re-running the benchmarks with updated library versions must still be decided.

Do you have any other questions? Would you like to have your system added to the benchmark? Please feel free to read the README in the repository, and if you still have questions, you can reach out to me at tom@ducklabs.com or on our Discord!

Introducing DuckDB for Swift

Publication date: 2023-04-21

Author: Tristan Celder

TL;DR: DuckDB now has a native Swift API. DuckDB on mobile here we go!

Today we're excited to announce the DuckDB API for Swift. It enables developers on Swift platforms to harness the full power of DuckDB using a native Swift interface with support for great Swift features such as strong typing and concurrency. The API is available not only on Apple platforms, but on Linux too, opening up new opportunities for the growing Swift on Server ecosystem.

What's Included

DuckDB is designed to be fast, reliable and easy to use, and it's this philosophy that also guided the creation of our new Swift API.

This initial release supports many of the great features of DuckDB right out of the box, including:

- Queries via DuckDB's enhanced SQL dialect: In addition to basic SQL, DuckDB supports arbitrary and nested correlated subqueries, window functions, collations, complex types (Swift arrays and structs), and more.
- Import and export of JSON, CSV, and Parquet files: Beyond its built-in and super-efficient native file format, DuckDB supports reading in, and exporting out to, JSON, CSV, and Parquet files.
- Strongly typed result sets: DuckDB's strongly typed result sets are a natural fit for Swift. It's simple to cast DuckDB columns to their native Swift equivalents, ready for presentation using SwiftUI or as part of an existing TabularData workflow.
- Swift concurrency support: by virtue of their `Sendable` conformance, many of DuckDB's core underlying types can be safely passed across concurrency contexts, easing the process of designing parallel processing workflows and ensuring responsive UIs.

Usage

To demonstrate just how well DuckDB works together with Swift, we've created an example project that uses raw data from NASA's Exoplanet Archive loaded directly into DuckDB.

You'll see how to:

- Instantiate a DuckDB in-memory Database and Connection
- Populate a DuckDB table with the contents of a remote CSV
- Query a DuckDB database and prepare the results for presentation

Finally, we'll present our analysis with the help of Apple's TabularData Framework and Swift Charts.

Instantiating DuckDB

DuckDB supports both file-based and in-memory databases. In this example, as we don't intend to persist the results of our Exoplanet analysis to disk, we'll opt for an in-memory Database.

```
let database = try Database(store: .inMemory)
```

However, we can't issue queries just yet. Much like other RDMSs, queries must be issued through a *database connection*. DuckDB supports multiple connections per database. This can be useful to support parallel processing, for example. In our project, we'll need just the one connection that we'll eventually access asynchronously.

```
let connection = try database.connect()
```

Finally, we'll create an app-specific type that we'll use to house our database and connection and through which we'll eventually define our app-specific queries.

```
import DuckDB

final class ExoplanetStore {

    let database: Database
    let connection: Connection

    init(database: Database, connection: Connection) {
        self.database = database
        self.connection = connection
    }
}
```

Populating DuckDB with a Remote CSV File

One problem with our current `ExoplanetStore` type is that it doesn't yet contain any data to query. To fix that, we'll load it with the data of every Exoplanet discovered to date from NASA's Exoplanet Archive.

There are hundreds of configuration options for this incredible resource, but today we want each exoplanet's name and its discovery year packaged as a CSV. Checking the docs gives us the following endpoint:

```
https://exoplanetarchive.ipac.caltech.edu/TAP/sync?query=select+pl_name+,+disc_
year+from+pscomppars&format=csv
```

Once we have our CSV downloaded locally, we can use the following SQL command to load it as a new table within our DuckDB in-memory database. DuckDB's `read_csv_auto` command automatically infers our table schema and the data is immediately available for analysis.

```
CREATE TABLE exoplanets AS
    SELECT * FROM read_csv_auto('downloaded_exoplanets.csv');
```

Let's package this up as a new asynchronous factory method on our `ExoplanetStore` type:

```
import DuckDB
import Foundation

final class ExoplanetStore {

    // Factory method to create and prepare a new ExoplanetStore
    static func create() async throws -> ExoplanetStore {

        // Create our database and connection as described above
        let database = try Database(store: .inMemory)
        let connection = try database.connect()

        // Download the CSV from the exoplanet archive
        let (csvFileURL, _) = try await URLSession.shared.download(
            from: URL(string: "https://exoplanetarchive.ipac.caltech.edu/TAP/sync?query=select+pl_
name+,+disc_year+from+pscomppars&format=csv")!)

        // Issue our first query to DuckDB
        try connection.execute("""
            CREATE TABLE exoplanets AS (
                SELECT * FROM read_csv_auto('\(csvFileURL.path)')
            );
            """)

        // Create our pre-populated ExoplanetStore instance
        return ExoplanetStore(
```

```
            database: database,
            connection: connection
        )
    }

    // Let's make the initializer we defined previously
    // private. This prevents anyone accidentally instantiating
    // the store without having pre-loaded our Exoplanet CSV
    // into the database
    private init(database: Database, connection: Connection) {
        // ...
    }
}
```

Querying the Database

Now that the database is populated with data, it's ready to be analyzed. Let's create a query which we can use to plot a chart of the number of exoplanets discovered by year.

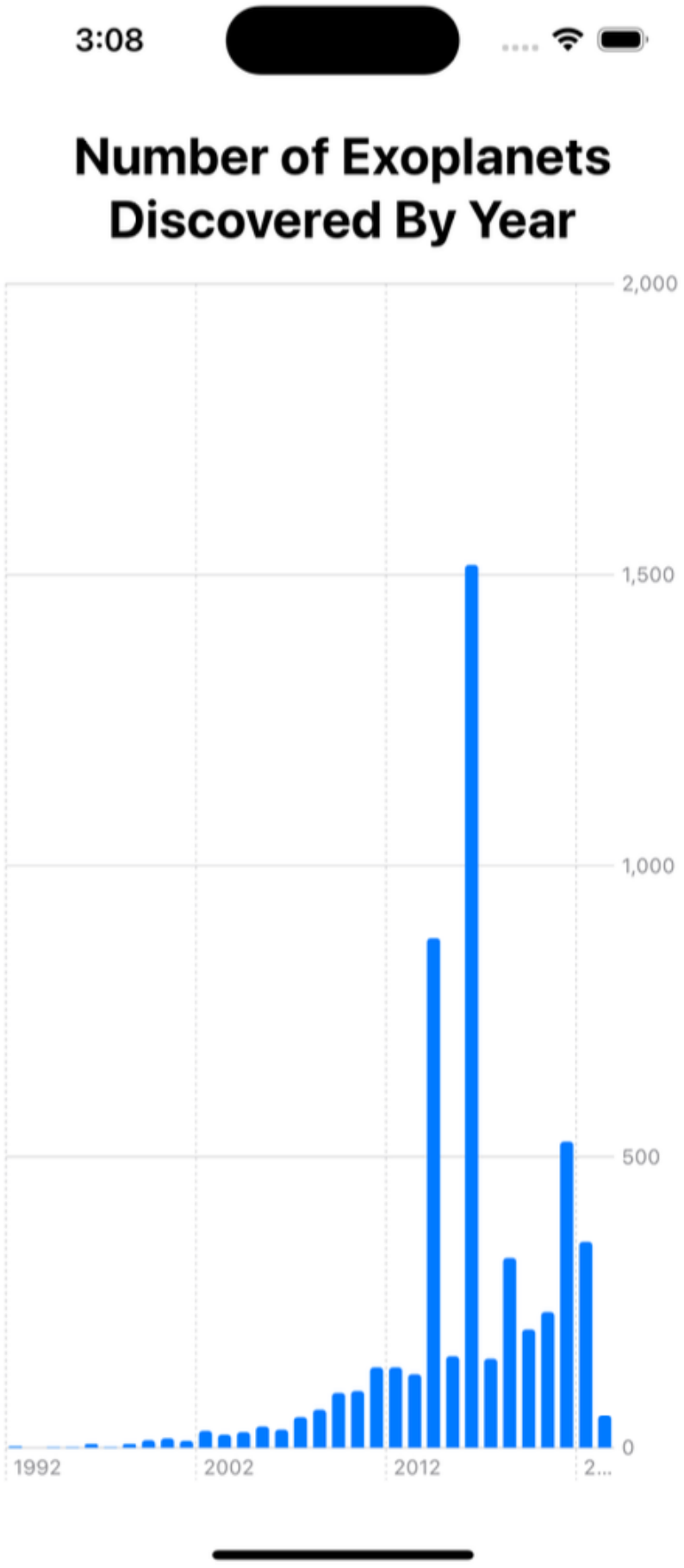
```
SELECT disc_year, count(disc_year) AS Count
FROM exoplanets
GROUP BY disc_year
ORDER BY disc_year;
```

Issuing the query to DuckDB from within Swift is simple. We'll again make use of an async function from which to issue our query. This means the callee won't be blocked while the query is executing. We'll then cast the result columns to Swift native types using DuckDB's `ResultSet` `cast(to:)` family of methods, before finally wrapping them up in a `DataFrame` from the TabularData framework ready for presentation in the UI.

```
...

import TabularData

extension ExoplanetStore {

    // Retrieves the number of exoplanets discovered by year
    func groupedByDiscoveryYear() async throws -> DataFrame {

        // Issue the query we described above
        let result = try connection.query("""
            SELECT disc_year, count(disc_year) AS Count
            FROM exoplanets
            GROUP BY disc_year
            ORDER BY disc_year
            """)

        // Cast our DuckDB columns to their native Swift
        // equivalent types
        let discoveryYearColumn = result[0].cast(to: Int.self)
        let countColumn = result[1].cast(to: Int.self)

        // Use our DuckDB columns to instantiate TabularData
        // columns and populate a TabularData DataFrame
        return DataFrame(columns: [
            TabularData.Column(discoveryYearColumn)
                .eraseToAnyColumn(),
            TabularData.Column(countColumn)
                .eraseToAnyColumn(),
        ])
    }
}
```

Visualizing the Results

In just a few lines of code, our database has been created, populated and analyzed – all that's left to do now is present the results.


3:08
Number of Exoplanets Discovered By Year
2,000
1,500
1,000
500
0
1992
2002
2012
2...

And I have a feeling that we're just getting started…

For the complete example project – including the SwiftUI views and Chart definitions used to create the screenshot above – clone the DuckDB Swift repo and open up the runnable app project located in `Examples/SwiftUI/ExoplanetExplorer.xcodeproj`.

We encourage you to modify the code, explore the Exoplanet Archive and DuckDB, and make some discoveries of your own – interplanetary or otherwise!

Conclusion

In this article we've introduced the brand new Swift API for DuckDB and demonstrated how quickly you can get up and running analyzing data.

With DuckDB's incredible performance and analysis capabilities and Swift's vibrant eco-system and platform support, there's never been a better time to begin exploring analytical datasets in Swift.

We can't wait to see what you do with it. Feel free to reach out on our Discord if you have any questions!

The Swift API for DuckDB is packaged using Swift Package Manager and lives in a new top-level repository available at https://github.com/duckdb/duckdb-swift.

PostGEESE? Introducing The DuckDB Spatial Extension

Publication date: 2023-04-28

Author: Max Gabrielsson

TL;DR: DuckDB now has an official Spatial extension to enable geospatial processing.

Geospatial data has become increasingly important and prevalent in modern-day applications and data engineering workflows, with use-cases ranging from location-based services to environmental monitoring.

While there are many great and specialized tools for working with geospatial data, integrating geospatial capabilities directly into DuckDB has multiple advantages. For one, you get to operate, transform and join your geospatial data alongside your regular, unstructured or time-series data using DuckDBs rich type system and extensions like `JSON` and `ICU`. Secondly, spatial queries involving geometric predicates and relations translate surprisingly well to SQL, which is all about expressing relations after all! Not to mention all the other benefits provided by DuckDB such as transactional semantics, high performance multi-threaded vectorized execution and larger-than-memory data processing.

Therefore, we're very excited to announce that DuckDB now has a Spatial extension packed with features easily installable from the DuckDB CLI and other DuckDB clients. Simply execute:

```
INSTALL spatial;
LOAD spatial;
```

And you're good to go!

No, we're not calling it GeoDuck either, that's just gross.

What's in It?

The core of the extension is a `GEOMETRY` type based on the "Simple Features" geometry model and accompanying functions such as `ST_Area`, `ST_Intersects`. It also provides methods for reading and writing geospatial data formats and converting between coordinate reference systems (details later in the post!). While we're not ready to commit to full compliance with the OGC Simple Feature Access and SQL/MM Standards yet, if you've worked with geospatial functionality in other database systems such as PostGIS or SpatiaLite, you should feel right at home.

Most of the implemented functions are based on the trifecta of foundational geospatial libraries, GEOS, GDAL and PROJ, which provide algorithms, format conversions and coordinate reference system transformations respectively. In particular, we leverage GDAL to provide a set of table and copy functions that enable import and export of tables from and to 50+ different geospatial data formats (so far!), including the most common ones such as Shapefiles, GeoJSON, GeoPackage, KML, GML, WKT, WKB, etc.

Check for yourself by running:

Initially we have prioritized providing a breadth of capabilities by wrapping existing libraries. We're planning to implement more of the core functions and algorithms natively in the future to enable faster performance and more efficient memory management.

As an initial step in this direction, we provide a set of non-standard specialized columnar DuckDB native geometry types such as `POINT_2D`, `BOX_2D`, etc. that should provide better compression and faster execution in exchange for some flexibility, but work around these are still very much experimental.

Example Usage

The following demonstrates how you can use the spatial extension to read and export multiple geospatial data formats, transform geometries between different coordinate reference systems and work with spatial property and predicate functions. While this example may be slightly contrived, we want to showcase the power of the currently available features. You can find the datasets used in this example in the spatial extension repository.

Let's import the NYC taxi ride data provided in Parquet format as well as the accompanying taxi zone data from a shapefile, using the `ST_Read` table function provided by the spatial extension. These taxi zones break NYC into polygons that represent regions, for example the Newark Airport. We then create a table for the rides and a table for the zones. Note that `ST_Read` produces a table with a `wkb_geometry` column that contains the geometry data encoded as a WKB (Well-Known Binary) blob, which we then convert to the `GEOMETRY` type using the `ST_GeomFromWKB` function.

> This may all seem a bit much if you are not familiar with the geospatial ecosystem, but rest assured this is all you really need to get started. In short: – Shapefile (.shp, .shx, .dbf) is a common format for storing geometry vector data and auxiliary metadata such as indexes and attributes. – WKB (Well Known Binary), while not really a file format in itself, is a common binary encoding of vector geometry data, used in e.g., GeoParquet. Comes in multiple flavors, but we're only concerned with "standard" WKB for now. – GEOMETRY is a DuckDB type that represents a Simple Features geometry object, which is based on a set of standards modeling vector geometry data as points, linestrings, polygons or collections of such. This is the core data type used by the spatial extension, and what most of the provided functions take and return.

```
INSTALL spatial;
LOAD spatial;

CREATE TABLE rides AS
    SELECT *
    FROM 'yellow_tripdata_2010-01-limit1mil.parquet';

-- Load the NYC taxi zone data from a shapefile using the gdal-based ST_Read function
CREATE TABLE zones AS
    SELECT zone, LocationId, borough, geom
    FROM ST_Read('taxi_zones/taxi_zones.shx');
```

Let's compare the trip distance to the linear distance between the pickup and dropoff points to figure out how efficient the taxi drivers are (or how dirty the data is, since some diffs seem to be negative). We transform the coordinates from "WGS84" (given by the identifier EPSG:4326), also commonly known as simply latitude/longitude to the "NAD83 / New York Long Island ftUS" (identified as ESRI:102718) coordinate reference system which is a projection with minimal distortion around New York. We then calculate the distance using the `ST_Distance` function. In This case we get the distance in feet since we've converted the coordinates to NAD83 but we can easily convert it into to miles (5280 ft/mile) which is the unit used in the rides dataset so we can compare them correctly.

Trips with a distance shorter than the aerial distance are likely to be erroneous, so we use this query to filter out some bad data. The query below takes advantage of DuckDB's ability to refer to column aliases defined within the same select statement. This is a small example of how DuckDB's rich SQL dialect can simplify geospatial analysis.

```
CREATE TABLE cleaned_rides AS
    SELECT
        ST_Point(pickup_latitude, pickup_longitude) AS pickup_point,
        ST_Point(dropoff_latitude, dropoff_longitude) AS dropoff_point,
        dropoff_datetime::TIMESTAMP - pickup_datetime::TIMESTAMP AS time,
        trip_distance,
        ST_Distance(
            ST_Transform(pickup_point, 'EPSG:4326', 'ESRI:102718'),
            ST_Transform(dropoff_point, 'EPSG:4326', 'ESRI:102718')) / 5280
            AS aerial_distance,
        trip_distance - aerial_distance AS diff
    FROM rides
    WHERE diff > 0
    ORDER BY diff DESC;
```

> It should be noted that this is not entirely accurate since the `ST_Distance` function we use does not take into account the curvature of the earth. However, we'll accept it as a good enough approximation for our purposes. Spherical and geodesic distance calculations are on the roadmap!

Now let's join the taxi rides with the taxi zones to get the start and end zone for each ride. We use the `ST_Within` function as our join condition to check if a pickup or dropoff point is within a taxi zone polygon. Again we need to transform the coordinates from WGS84 to the NAD83 since the taxi zone data also use that projection. Spatial joins like these are the bread and butter of geospatial data processing, but we don't currently have any optimizations in place (such as spatial indexes) to speed up these queries, which is why we only use a subset of the data for the following step.

```
-- Since we don't have spatial indexes yet, use a smaller dataset for the join.
DELETE FROM cleaned_rides WHERE rowid > 5000;

CREATE TABLE joined AS
    SELECT
        pickup_point,
        dropoff_point,
        start_zone.zone AS start_zone,
        end_zone.zone AS end_zone,
        trip_distance,
        time,
    FROM cleaned_rides
    JOIN zones AS start_zone
      ON ST_Within(ST_Transform(pickup_point, 'EPSG:4326', 'ESRI:102718'), start_zone.geom)
    JOIN zones AS end_zone
      ON ST_Within(ST_Transform(dropoff_point, 'EPSG:4326', 'ESRI:102718'), end_zone.geom);
```

We can export the joined table to a `GeoJSONSeq` file using the GDAL copy function, passing in a GDAL layer creation option. Since GeoJSON only supports a single `GEOMETRY` per record, we use the `ST_MakeLine` function to combine the pickup and dropoff points into a single line geometry. The default coordinate reference system for `GeoJSON` is WGS84, but the coordinate pairs are expected to be in longitude/latitude, so we need to flip the geometry using the `ST_FlipCoordinates` function.

```
COPY (
    SELECT
        ST_MakeLine(pickup_point, dropoff_point)
            .ST_FlipCoordinates()
            .ST_AsWKB()
            AS wkb_geometry,
        start_zone,
        end_zone,
        time::VARCHAR AS trip_time
    FROM joined)
TO 'joined.geojsonseq'
WITH (
    FORMAT gdal,
    DRIVER 'GeoJSONSeq',
    LAYER_CREATION_OPTIONS 'WRITE_BBOX=YES'
);
```

And there we have it! We pulled tabular data from Parquet, combined it with geospatial data in a shapefile, cleaned and analyzed that combined data, and output it to a human readable geospatial format. The full set of currently supported functions and their implementation status can be found over at the docs in this table.

What's Next?

While it's probably going to take a while for us to catch up to the full set of functions provided by e.g., PostGIS, we believe that DuckDB's vectorized execution model and columnar storage format will enable a whole new class of optimizations for geospatial processing that we've just begun exploring. Improving the performance of spatial joins and predicates is therefore high on our list of priorities.

There are also some limitations with our `GEOMETRY` type that we would eventually like to tackle, such as the fact that we don't support additional Z and M dimensions, or don't support the full range of geometry sub-types that are mandated by the OGC standard, like curves or polyhedral surfaces.

We're also interested in supporting spherical and ellipsoidal calculations in the near future, perhaps in the form of a dedicated `GEOGRAPHY` type.

Wasm builds are also just around the corner!

Please take a look at the GitHub repository for the full roadmap and to see what we're currently working on. If you would like to help build this capability, please reach out on GitHub!

Conclusion

The DuckDB Spatial extension is another step towards making DuckDB a swiss army knife for data engineering and analytics. This extension provides a flexible and familiar `GEOMETRY` type, reprojectable between thousands of coordinate reference systems, coupled with the capability to export and import geospatial data between more than 50 different data sources. All embedded into a single extension with minimal runtime dependencies. This enables DuckDB to fit seamlessly into your existing GIS workflows regardless of which geospatial data formats or projections you're working with.

We are excited to hear what you make of the DuckDB spatial extension. It's still early days but we hope to have a lot more to share in the future as we continue making progress! If you have any questions, suggestions, ideas or issues, please don't hesitate to reach out to us on Discord or GitHub!

10 000 Stars on GitHub

Publication date: 2023-05-12

Authors: Mark Raasveldt, Hannes Mühleisen

Today, DuckDB reached 10 000 stars on GitHub. We would like to pause for a second to express our gratitude to everyone who contributed to DuckDB and of course all its users. When we started working on DuckDB back in 2018, we would have never dreamt of getting this kind of adoption in such a short time.

From those brave souls who were early adopters of DuckDB back in 2019 to the many today, we are happy you're part of our community. Thank you for your feedback, feature requests and for your enthusiasm in adopting new features and integrations. Thank you for helping each other on our Discord server or in GitHub Discussions. Thank you for spreading the word, too.

We also would like to extend special thanks to the DuckDB foundation supporters, who through their generous donations keep DuckDB independent.

For us, the maintainers of DuckDB, the past few years have also been quite eventful: We spun off from the research group where DuckDB originated to a successful company with close to 20 employees and many excellent partnerships.

We are very much looking forward to what the future will hold for DuckDB. Things are looking bright!

Announcing DuckDB 0.8.0

Publication date: 2023-05-17

Authors: Mark Raasveldt, Hannes Mühleisen

The DuckDB team is happy to announce the latest DuckDB release (0.8.0). This release is named "Fulvigula" after the Mottled Duck (Anas Fulvigula) native to the Gulf of Mexico.

To install the new version, please visit the installation guide. The full release notes can be found on GitHub.

What's New in 0.8.0

There have been too many changes to discuss them each in detail, but we would like to highlight several particularly exciting features!

- New pivot and unpivot statements
- Improvements to parallel data import/export
- Time series joins
- Recursive globbing
- Lazy-loading of storage metadata for faster startup times
- User-defined functions for Python
- Arrow Database Connectivity (ADBC) support
- New Swift integration

Below is a summary of those new features with examples, starting with two breaking changes in our SQL dialect that are designed to produce more intuitive results by default.

Breaking SQL Changes

This release includes two breaking changes to the SQL dialect: The division operator uses floating point division by default, and the default null sort order is changed from `NULLS FIRST` to `NULLS LAST`. While DuckDB is still in Beta, we recognize that many DuckDB queries are already used in production. So, the old behavior can be restored using the following settings:

```
SET integer_division = true;
SET default_null_order = 'NULLS_FIRST';
```

Division Operator. The division operator / will now always perform a floating point division even with integer parameters. The new operator // retains the old semantics and can be used to perform integer division. This makes DuckDB's division operator less error prone for beginners, and consistent with the division operator in Python 3 and other systems in the OLAP space like Spark, Snowflake and BigQuery.

```
SELECT 42 / 5, 42 // 5;
```

(42 / 5)	(42 // 5)
8.4	8

Default Null Sort Order. The default null sort order is changed from `NULLS FIRST` to `NULLS LAST`. The reason for this change is that `NULLS LAST` sort-order is more intuitive when combined with `LIMIT`. With `NULLS FIRST`, Top-N queries always return the `NULL` values first. With `NULLS LAST`, the actual Top-N values are returned instead.

```
CREATE TABLE bigdata (col INTEGER);
INSERT INTO bigdata VALUES (NULL), (42), (NULL), (43);
FROM bigdata ORDER BY col DESC LIMIT 3;
```

v0.7.1	v0.8.0
NULL	43
NULL	42
43	NULL

New SQL Features

Pivot and Unpivot. There are many shapes and sizes of data, and we do not always have control over the process in which data is generated. While SQL is well-suited for reshaping datasets, turning columns into rows or rows into columns is tedious in vanilla SQL. With this release, DuckDB introduces the `PIVOT` and `UNPIVOT` statements that allow reshaping data sets so that rows are turned into columns or vice versa. A key advantage of DuckDB's syntax is that the column names to pivot or unpivot can be automatically deduced. Here is a short example:

```
CREATE TABLE sales (year INTEGER, amount INTEGER);
INSERT INTO sales VALUES (2021, 42), (2022, 100), (2021, 42);
PIVOT sales ON year USING sum(amount);
```

2021	2022
84	100

The documentation contains more examples.

ASOF Joins for Time Series. When joining time series data with background fact tables, the timestamps often do not exactly match. In this case it is often desirable to join rows so that the timestamp is joined with the *nearest timestamp*. The ASOF join can be used for this purpose – it performs a fuzzy join to find the closest join partner for each row instead of requiring an exact match.

```
CREATE TABLE a (ts TIMESTAMP);
CREATE TABLE b (ts TIMESTAMP);
INSERT INTO a VALUES (TIMESTAMP '2023-05-15 10:31:00'), (TIMESTAMP '2023-05-15 11:31:00');
INSERT INTO b VALUES (TIMESTAMP '2023-05-15 10:30:00'), (TIMESTAMP '2023-05-15 11:30:00');

FROM a ASOF JOIN b ON a.ts >= b.ts;
```

a.ts	b.ts
2023-05-15 10:31:00	2023-05-15 10:30:00
2023-05-15 11:31:00	2023-05-15 11:30:00

Please refer to the documentation for a more in-depth explanation.

Data Integration Improvements

Default Parallel CSV Reader. In this release, the parallel CSV reader has been vastly improved and is now the default CSV reader. We would like to thank everyone that has tried out the experimental reader for their valuable feedback and reports. The `experimental_parallel_csv` flag has been deprecated and is no longer required. The parallel CSV reader enables much more efficient reading of large CSV files.

```
CREATE TABLE lineitem AS FROM lineitem.csv;
```

v0.7.1	v0.8.0
4.1 s	1.2 s

Parallel Parquet, CSV and JSON Writing. This release includes support for parallel *order-preserving* writing of Parquet, CSV and JSON files. As a result, writing to these file formats is parallel by default, also without disabling insertion order preservation, and writing to these formats is greatly sped up.

```
COPY lineitem TO 'lineitem.csv';
COPY lineitem TO 'lineitem.parquet';
COPY lineitem TO 'lineitem.json';
```

Format	v0.7.1	v0.8.0
CSV	3.9 s	0.6 s
Parquet	8.1 s	1.2 s
JSON	4.4 s	1.1 s

Recursive File Globbing using `**`. This release adds support for recursive globbing where an arbitrary number of subdirectories can be matched using the `**` operator (double-star).

```
FROM 'data/glob/crawl/stackoverflow/**/*.csv';
```

The documentation has been updated with various examples of this syntax.

Storage Improvements

Lazy-Loading Table Metadata. DuckDB's internal storage format stores metadata for every row group in a table, such as min-max indexes and where in the file every row group is stored. In the past, DuckDB would load this metadata immediately once the database was opened. However, once the data gets very big, the metadata can also get quite large, leading to a noticeable delay on database startup. In this release, we have optimized the metadata handling of DuckDB to only read table metadata as its being accessed. As a result, startup is near-instantaneous even for large databases, and metadata is only loaded for columns that are actually used in queries. The benchmarks below are for a database file containing a single large TPC-H `lineitem` table (120× SF1) with ~770 million rows and 16 columns:

Query	v0.6.1	v0.7.1	v0.8.0	Parquet
`SELECT 42`	1.60 s	0.31 s	0.02 s	-
`FROM lineitem LIMIT 1`	1.62 s	0.32 s	0.03 s	0.27 s

Clients

User-Defined Scalar Functions for Python. Arbitrary Python functions can now be registered as scalar functions within SQL queries. This will only work when using DuckDB from Python, because it uses the actual Python runtime that DuckDB is running within. While plain Python values can be passed to the function, there is also a vectorized variant that uses PyArrow under the hood for higher efficiency and better parallelism.

```
import duckdb

from duckdb.typing import *
from faker import Faker

def random_date():
     fake = Faker()
     return fake.date_between()

duckdb.create_function('random_date', random_date, [], DATE)
res = duckdb.sql('SELECT random_date()').fetchall()
print(res)
# [(datetime.date(2019, 5, 15),)]
```

See the documentation for more information.

Arrow Database Connectivity Support (ADBC). ADBC is a database API standard for database access libraries that uses Apache Arrow to transfer query result sets and to ingest data. Using Arrow for this is particularly beneficial for columnar data management systems which traditionally suffered a performance hit by emulating row-based APIs such as JDBC/ODBC. From this release, DuckDB natively supports ADBC. We're happy to be one of the first systems to offer native support, and DuckDB's in-process design fits nicely with ADBC.

Swift Integration. DuckDB has gained another official language integration: Swift. Swift is a language developed by Apple that most notably is used to create Apps for Apple devices, but also increasingly used for server-side development. The DuckDB Swift API allows developers on all swift platforms to harness DuckDB using a native Swift interface with support for Swift features like strong typing and concurrency.

Final Thoughts

The full release notes can be found on GitHub. We would like to thank all of the contributors for their hard work on improving DuckDB.

Correlated Subqueries in SQL

Publication date: 2023-05-26

Author: Mark Raasveldt

Subqueries in SQL are a powerful abstraction that allow simple queries to be used as composable building blocks. They allow you to break down complex problems into smaller parts, and subsequently make it easier to write, understand and maintain large and complex queries.

DuckDB uses a state-of-the-art subquery decorrelation optimizer that allows subqueries to be executed very efficiently. As a result, users can freely use subqueries to create expressive queries without having to worry about manually rewriting subqueries into joins. For more information, skip to the Performance section.

Types of Subqueries

SQL subqueries exist in two main forms: subqueries as *expressions* and subqueries as *tables*. Subqueries that are used as expressions can be used in the `SELECT` or `WHERE` clauses. Subqueries that are used as tables can be used in the `FROM` clause. In this blog post we will focus on subqueries used as *expressions*. A future blog post will discuss subqueries as *tables*.

Subqueries as expressions exist in three forms.

- Scalar subqueries
- `EXISTS`
- `IN`/`ANY`/`ALL`

All of the subqueries can be either *correlated* or *uncorrelated*. An uncorrelated subquery is a query that is independent from the outer query. A correlated subquery is a subquery that contains expressions from the outer query. Correlated subqueries can be seen as *parameterized subqueries*.

Uncorrelated Scalar Subqueries

Uncorrelated scalar subqueries can only return *a single value*. That constant value is then substituted and used in the query. As an example of why this is useful – imagine that we want to select all of the shortest flights in our dataset. We could run the following query to obtain the shortest flight distance:

```
SELECT min(distance)
FROM ontime;
```

min(distance)
31.0

We could manually take this distance and use it in the `WHERE` clause to obtain all flights on this route.

```
SELECT uniquecarrier, origincityname, destcityname, flightdate
FROM ontime
WHERE distance = 31.0;
```

uniquecarrier	origincityname	destcityname	flightdate
AS	Petersburg, AK	Wrangell, AK	2017-01-15
AS	Wrangell, AK	Petersburg, AK	2017-01-15
AS	Petersburg, AK	Wrangell, AK	2017-01-16

However – this requires us to hardcode the constant inside the query. By using the first query as a *subquery* we can compute the minimum distance as part of the query.

```
SELECT uniquecarrier, origincityname, destcityname, flightdate
FROM ontime
WHERE distance = (
     SELECT min(distance)
     FROM ontime
);
```

Correlated Scalar Subqueries

While uncorrelated subqueries are powerful, they come with a hard restriction: only a *single value* can be returned. Often, what we want to do is *parameterize* the query, so that we can return different values per row.

For example, suppose that we want to find all of the shortest flights *for each carrier*. We can find the shortest flight for a *specific carrier* using the following parameterized query:

```
PREPARE min_distance_per_carrier AS
SELECT min(distance)
FROM ontime
WHERE uniquecarrier = ?;
```

We can execute this prepared statement to obtain the minimum distance for a specific carrier.

```
EXECUTE min_distance_per_carrier('UA');
```

min(distance)
67.0

If we want to use this parameterized query as a subquery, we need to use a *correlated subquery*. Correlated subqueries allow us to use parameterized queries as scalar subqueries by referencing columns from *the outer query*. We can obtain the set of shortest flights per carrier using the following query:

```
SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime AS ontime_outer
WHERE distance = (
     SELECT min(distance)
     FROM ontime
     WHERE uniquecarrier = ontime_outer.uniquecarrier
);
```

uniquecarrier	origincityname	destcityname	flightdate	distance
AS	Wrangell, AK	Petersburg, AK	2017-01-01	31.0
NK	Fort Lauderdale, FL	Orlando, FL	2017-01-01	177.0
VX	Las Vegas, NV	Los Angeles, CA	2017-01-01	236.0

Notice how the column from the *outer* relation (`ontime_outer`) is used *inside* the query. This is what turns the subquery into a *correlated subquery*. The column from the outer relation (`ontime_outer.uniquecarrier`) is a *parameter* for the subquery. Logically the subquery is executed once for every row that is present in `ontime`, where the value for the column at that row is substituted as a parameter.

In order to make it more clear that the correlated subquery is in essence a *parameterized query*, we can create a scalar macro that contains the query using DuckDB's macros.

```
CREATE MACRO min_distance_per_carrier(param) AS (
     SELECT min(distance)
     FROM ontime
     WHERE uniquecarrier = param
);
```

We can then use the macro in our original query as if it is a function.

```
SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime AS ontime_outer
WHERE distance = min_distance_per_carrier(ontime_outer.uniquecarrier);
```

This gives us the same result as placing the correlated subquery inside of the query, but is cleaner as we can decompose the query into multiple segments more effectively.

EXISTS

`EXISTS` can be used to check if a given subquery has any results. This is powerful when used as a correlated subquery. For example, we can use `EXISTS` if we want to obtain the *last flight that has been flown on each route*.

We can obtain a list of all flights on a given route past a certain date using the following query:

```
PREPARE flights_after_date AS
SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime
WHERE origin = ? AND dest = ? AND flightdate > ?;

EXECUTE flights_after_date('LAX', 'JFK', DATE '2017-05-01');
```

uniquecarrier	origincityname	destcityname	flightdate	distance
AA	Los Angeles, CA	New York, NY	2017-08-01	2475.0
AA	Los Angeles, CA	New York, NY	2017-08-02	2475.0
AA	Los Angeles, CA	New York, NY	2017-08-03	2475.0

Now in order to obtain the *last flight on a route*, we need to find flights *for which no later flight exists*.

```
SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime AS ontime_outer
WHERE NOT EXISTS (
     SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
     FROM ontime
     WHERE origin = ontime_outer.origin
       AND dest = ontime_outer.dest
       AND flightdate > ontime_outer.flightdate
);
```

uniquecarrier	origincityname	destcityname	flightdate	distance
AA	Daytona Beach, FL	Charlotte, NC	2017-02-27	416.0
EV	Abilene, TX	Dallas/Fort Worth, TX	2017-02-15	158.0

uniquecarrier	origincityname	destcityname	flightdate	distance
EV	Dallas/Fort Worth, TX	Durango, CO	2017-02-13	674.0

IN / ANY / ALL

`IN` can be used to check if a *given value* exists within the result returned by the subquery. For example, we can obtain a list of all carriers that have performed more than `250 000` flights in the dataset using the following query:

```
SELECT uniquecarrier
FROM ontime
GROUP BY uniquecarrier
HAVING count(*) > 250000;
```

We can then use an `IN` clause to obtain all flights performed by those carriers.

```
SELECT *
FROM ontime
WHERE uniquecarrier IN (
     SELECT uniquecarrier
     FROM ontime
     GROUP BY uniquecarrier
     HAVING count(*) > 250000
);
```

A correlated subquery can be useful here if we want to not count the total amount of flights performed by each carrier, but count the total amount of flights *for the given route*. We can select all flights performed by carriers that have performed *at least 1000 flights on a given route* using the following query.

```
SELECT *
FROM ontime AS ontime_outer
WHERE uniquecarrier IN (
     SELECT uniquecarrier
     FROM ontime
     WHERE ontime.origin = ontime_outer.origin
       AND ontime.dest = ontime_outer.dest
     GROUP BY uniquecarrier
     HAVING count(*) > 1000
);
```

`ANY` and `ALL` are generalizations of `IN`. `IN` checks if the value is present in the set returned by the subquery. This is equivalent to `= ANY(...)`. The `ANY` and `ALL` operators can be used to perform other comparison operators (such as `>`, `<`, `<>`). The above query can be rewritten to `ANY` in the following form.

```
SELECT *
FROM ontime AS ontime_outer
WHERE uniquecarrier = ANY (
     SELECT uniquecarrier
     FROM ontime
     WHERE ontime.origin = ontime_outer.origin
       AND ontime.dest = ontime_outer.dest
     GROUP BY uniquecarrier
     HAVING count(*) > 1000
);
```

Performance

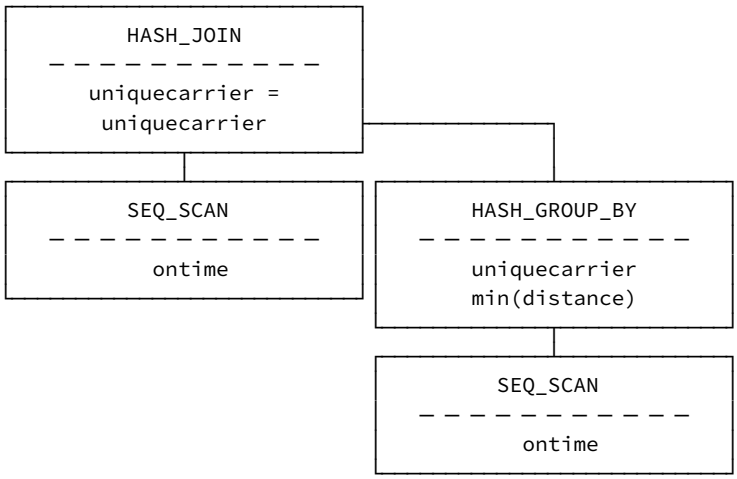
Whereas scalar subqueries are logically executed *once*, correlated subqueries are logically executed *once per row*. As such, it is natural to think that correlated subqueries are very expensive and should be avoided for performance reasons.

While that is true in many SQL systems, it is not the case in DuckDB. In DuckDB, subqueries are **always** *decorrelated*. DuckDB uses a state-of-the-art subquery decorrelation algorithm as described in the Unnesting Arbitrary Queries paper. This allows all subqueries to be decorrelated and executed as a single, much more efficient, query.

In DuckDB, correlation does not imply performance degradation.

If we look at the query plan for the correlated scalar subquery using `EXPLAIN`, we can see that the query has been transformed into a hash aggregate followed by a hash join. This allows the query to be executed very efficiently.

```
EXPLAIN SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime AS ontime_outer
WHERE distance = (
     SELECT min(distance)
     FROM ontime
     WHERE uniquecarrier = ontime_outer.uniquecarrier
);
```



We can see the drastic performance difference that subquery decorrelation has when we compare the run-time of this query in DuckDB with the run-time in Postgres and SQLite. When running the above query on the `ontime` dataset for `2017` with roughly `~4 million` rows, we get the following performance results:

DuckDB	Postgres	SQLite
0.06 s	>48 hours	>48 hours

As Postgres and SQLite do not de-correlate the subquery, the query is not just *logically*, but *actually* executed once for every row. As a result, the subquery is executed *4 million times* in those systems, which takes an immense amount of time.

In this case, it is possible to manually decorrelate the query and generate the following SQL:

```
SELECT ontime.uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime
JOIN (
     SELECT uniquecarrier, min(distance) AS min_distance
     FROM ontime
     GROUP BY uniquecarrier
  ) AS subquery
  ON ontime.uniquecarrier = subquery.uniquecarrier
 AND distance = min_distance;
```

By performing the de-correlation manually, the performance of SQLite and Postgres improves significantly. However, both systems remain over 30× slower than DuckDB.

DuckDB	Postgres	SQLite
0.06 s	1.98 s	2.81 s

Note that while it is possible to manually decorrelate certain subqueries by rewriting the SQL, it is not always possible to do so. As described in the Unnesting Arbitrary Queries paper, special join types that are not present in SQL are necessary to decorrelate arbitrary queries.

In DuckDB, these special join types will be automatically generated by the system to decorrelate all subqueries. In fact, DuckDB does not have support for executing subqueries that are not decorrelated. All subqueries will be decorrelated before DuckDB executes them.

Conclusion

Subqueries are a very powerful tool that allow you to take arbitrary queries and convert them into ad-hoc functions. When used in combination with DuckDB's powerful subquery decorrelation, they can be executed extremely efficiently, making previously intractable queries not only possible, but fast.

From Waddle to Flying: Quickly Expanding DuckDB's Functionality with Scalar Python UDFs

Publication date: 2023-07-07

Authors: Pedro Holanda, Thijs Bruineman, Phillip Cloud

TL;DR: DuckDB now supports vectorized Scalar Python User Defined Functions (UDFs). By implementing Python UDFs, users can easily expand the functionality of DuckDB while taking advantage of DuckDB's fast execution model, SQL and data safety.

User Defined Functions (UDFs) enable users to extend the functionality of a Database Management System (DBMS) to perform domain-specific tasks that are not implemented as built-in functions. For instance, users who frequently need to export private data can benefit from an anonymization function that masks the local part of an email while preserving the domain. Ideally, this function would be executed directly in the DBMS. This approach offers several advantages:

1) **Performance.** The function could be executed using the same execution model (e.g., streaming results, beyond-memory/out-of-core execution) of the DBMS, and without any unnecessary transformations.

2) **Easy Use.** UDFs can be seamlessly integrated into SQL queries, allowing users to leverage the power of SQL to call the functions. This eliminates the need for passing data through a separate database connector and executing external code. The functions can be utilized in various SQL contexts (e.g., subqueries, join conditions).

3) **Safety.** The sensitive data never leaves the DBMS process.

There are two main reasons users often refrain from implementing UDFs. 1) There are security concerns associated with UDFs. Since UDFs are custom code created by users and executed within the DBMS process, there is a potential risk of crashing the server. However, when it comes to DuckDB, an embedded database, this concern is mitigated as each analyst runs their own DuckDB process separately. Therefore, the impact on server stability is not a significant worry. 2) The difficulty of implementation is a common deterrent for users. High-Performance UDFs are typically only supported in low-level languages. UDFs in higher-level languages like Python incur significant performance costs. Consequently many users cannot quickly implement their UDFs without investing a significant amount of time in learning a low-level language and understanding the internal details of the DBMS.

DuckDB followed a similar approach. As a DBMS tailored for analytical tasks, performance is a key consideration, leading to the implementation of its core in C++. Consequently, the initial focus of extensibility efforts was centered around C++. However, this duck is not limited

to just waddling; it can also fly. So we are delighted to announce the recent addition of Scalar Python UDFs to DuckDB.

DuckDB provides support for two distinct types of Python UDFs, differing in the Python object used for communication between DuckDB's native data types and the Python process. These communication layers include support for Python built-in types and PyArrow Tables.

The two approaches exhibit two key differences:

1) **Zero-Copy.** PyArrow Tables leverage our zero-copy integration with Arrow, enabling efficient translation of data types to Python-Land with zero-copy cost.

2) **Vectorization.** PyArrow Table functions operate on a chunk level, processing chunks of data containing up to 2048 rows. This approach maximizes cache locality and leverages vectorization. On the other hand, the built-in types UDF implementation operates on a per-row basis.

This blog post aims to demonstrate how you can extend DuckDB using Python UDFs, with a particular emphasis on PyArrow-powered UDFs. In our quick-tour section, we will provide examples using the PyArrow UDF types. For those interested in benchmarks, you can jump ahead to the benchmark section below. If you want to see a detailed description of the Python UDF API, please refer to our documentation.

Python UDFs

This section depicts several practical examples of using Python UDFs. Each example uses a different type of Python UDF.

Quick-Tour

To demonstrate the usage of Python UDFs in DuckDB, let's consider the following example. We have a dictionary called `world_cup_titles` that maps countries to the number of World Cups they have won. We want to create a Python UDF that takes a country name as input, searches for the corresponding value in the dictionary, and returns the number of World Cups won by that country. If the country is not found in the dictionary, the UDF will return `NULL`.

Here's an example implementation:

```
import duckdb
from duckdb.typing import *

con = duckdb.connect()

# Dictionary that maps countries and world cups they won
world_cup_titles = {
    "Brazil": 5,
    "Germany": 4,
    "Italy": 4,
    "Argentina": 2,
    "Uruguay": 2,
    "France": 2,
    "England": 1,
    "Spain": 1
}

# Function that will be registered as an UDF, simply does a lookup in the python dictionary
def world_cups(x):
     return world_cup_titles.get(x)

# We register the function
con.create_function("wc_titles", world_cups, [VARCHAR], INTEGER)
```

That's it, the function is then registered and ready to be called through SQL.

```
# Let's create an example countries table with the countries we are interested in using
con.execute("CREATE TABLE countries (country VARCHAR)")
con.execute("INSERT INTO countries VALUES ('Brazil'), ('Germany'), ('Italy'), ('Argentina'), ('Uruguay'),
('France'), ('England'), ('Spain'), ('Netherlands')")
```

```
# We can simply call the function through SQL, and even use the function return to eliminate the countries
that never won a world cup
con.sql("SELECT country, wc_titles(country) AS world_cups FROM countries").fetchall()
# [('Brazil', 5), ('Germany', 4), ('Italy', 4), ('Argentina', 2), ('Uruguay', 2), ('France', 2),
('England', 1), ('Spain', 1), ('Netherlands', None)]
```

Generating Fake Data with Faker (Built-In Type UDF)

Here is an example that demonstrates the usage of the Faker library to generate a scalar function in DuckDB, which returns randomly generated dates. The function, named `random_date`, does not require any inputs and outputs a `DATE` column. Since Faker utilizes built-in Python types, the function directly returns them. One important thing to notice is that a function that is not deterministic based on its input must be marked as having `side_effects`.

```
import duckdb

# By importing duckdb.typing we can specify DuckDB Types directly without using strings
from duckdb.typing import *

from faker import Faker

# Our Python UDF generates a random date every time it's called
def random_date():
     fake = Faker()
     return fake.date_between()
```

We then have to register the Python function in DuckDB using `create_function`. Since our function doesn't require any inputs, we can pass an empty list as the `argument_type_list`. As the function returns a date, we specify `DATE` from `duckdb.typing` as the `return_type`. Note that since our `random_date()` function returns a built-in Python type (`datetime.date`), we don't need to specify the UDF type.

```
# To exemplify the effect of side-effect, let's first run the function without marking it.
duckdb.create_function('random_date', random_date, [], DATE)

# After registration, we can use the function directly via SQL
# Notice that without side_effect=True, it's not guaranteed that the function will be re-evaluated.
res = duckdb.sql('SELECT random_date() FROM range (3)').fetchall()
# [(datetime.date(2003, 8, 3),), (datetime.date(2003, 8, 3),), (datetime.date(2003, 8, 3),)]

# Now let's re-add the function with side-effects marked as true.
duckdb.remove_function('random_date')
duckdb.create_function('random_date', random_date, [], DATE, side_effects=True)
res = duckdb.sql('SELECT random_date() FROM range (3)').fetchall()
# [(datetime.date(2020, 11, 29),), (datetime.date(2009, 5, 18),), (datetime.date(2018, 5, 24),)]
```

Swap String Case (PyArrow Type UDF)

One issue with using built-in types is that you don't benefit from zero-copy, vectorization and cache locality. Using PyArrow as a UDF type should be favored to leverage these optimizations.

To demonstrate a PyArrow function, let's consider a simple example where we want to transform lowercase characters to uppercase and uppercase characters to lowercase. Fortunately, PyArrow already has a function for this in the compute engine, and it's as simple as calling `pc.utf8_swapcase(x)`.

```
import duckdb

# By importing duckdb.typing we can specify DuckDB Types directly without using strings
from duckdb.typing import *

import pyarrow as pa
import pyarrow.compute as pc
```

```
def swap_case(x):
     # Swap the case of the 'column' using utf8_swapcase and return the result
     return pc.utf8_swapcase(x)

con = duckdb.connect()
# To register the function, we must define it's type to be 'arrow'
con.create_function('swap_case', swap_case, [VARCHAR], VARCHAR, type='arrow')

res = con.sql("SELECT swap_case('PEDRO HOLANDA')").fetchall()
# [('pedro holanda',)]
```

Predicting Taxi Fare Costs (Ibis + PyArrow UDF)

Python UDFs offer significant power as they enable users to leverage the extensive Python ecosystem and tools, including libraries like PyTorch and Tensorflow that efficiently implement machine learning operations.

Additionally the Ibis project offers a DataFrame API with great DuckDB integration and supports both of DuckDB's native Python and PyArrow UDFs.

In this example, we demonstrate the usage of a pre-built PyTorch model to estimate taxi fare costs based on the traveled distance. You can find a complete example in this blog post by the Ibis team.

```
import torch
import pyarrow as pa
import ibis
import ibis.expr.datatypes as dt

from ibis.expr.operations import udf


# The code to generate the model is not specified in this snippet, please refer to the provided link for
more information
model = ...

# Function that uses the model and a traveled distance input tensor to predict values, please refer to the
provided link for more information
def predict_linear_regression(model, tensor: torch.Tensor) -> torch.Tensor:
    ...


# Indicate to ibis that this is a scalar user-defined function whose input format is pyarrow
@udf.scalar.pyarrow
def predict_fare(x: dt.float64) -> dt.float32:
    # `x` is a pyarrow.ChunkedArray; the `dt.float64` annotation indicate the element type of the
    ChunkedArray.

    # Transform the data from PyArrow to the required torch tensor format and dimension.
    tensor = torch.from_numpy(x.to_numpy()[:, None]).float()

    # Call the actual prediction function, which also returns a torch tensor.
    predicted = predict_linear_regression(model, tensor).ravel()
    return pa.array(predicted.numpy())


# Execute a query on the NYC Taxi Parquet file to showcase our model's predictions, the actual fare amount,
and the distance.
expr = (
    ibis.read_parquet('yellow_tripdata_2016-02.parquet')
    .mutate(
        "fare_amount",
        "trip_distance",
        predicted_fare=lambda t: predict_fare(t.trip_distance),
    )
)
```

```
df = expr.execute()
```

By utilizing Python UDFs in DuckDB with Ibis, you can seamlessly incorporate machine learning models and perform predictions directly within your Ibis code and SQL queries. The example demonstrates how to predict taxi fare costs based on distance using a PyTorch model, showcasing the integration of machine learning capabilities within DuckDB's SQL environment driven by Ibis.

Benchmarks

In this section, we will perform simple benchmark comparisons to demonstrate the performance differences between two different types of Python UDFs. The benchmark will measure the execution time, and peak memory consumption. The benchmarks are executed 5 times, and the median value is considered. The benchmark is conducted on a Mac Apple M1 with 16 GB of RAM.

Built-In Python vs. PyArrow

To benchmark these UDF types, we create UDFs that take an integral column as input, add one to each value, and return the result. The code used for this benchmark section can be found in a GitHub Gist.

```
import pyarrow.compute as pc
import duckdb
import pyarrow as pa

# Built-In UDF
def add_built_in_type(x):
    return x + 1

# Arrow UDF
def add_arrow_type(x):
    return pc.add(x, 1)

con = duckdb.connect()

# Registration
con.create_function('built_in_types', add_built_in_type, ['BIGINT'], 'BIGINT', type='native')
con.create_function('add_arrow_type', add_arrow_type, ['BIGINT'], 'BIGINT', type='arrow')

# Integer View with 10,000,000 elements.
con.sql("""
     SELECT i
     FROM range(10000000) tbl(i);
""").to_view("numbers")

# Calls for both UDFs
native_res = con.sql("SELECT sum(add_built_in_type(i)) FROM numbers").fetchall()
arrow_res = con.sql("SELECT sum(add_arrow_type(i)) FROM numbers").fetchall()
```

Name	Time (s)
Built-In	5.37
PyArrow	0.35

We can observe a performance difference of more than one order of magnitude between the two UDFs. The difference in performance is primarily due to three factors:

1) In Python, object construction and general use is rather slow. This is due to several reasons, including automatic memory management, interpretation, and dynamic typing. 2) The PyArrow UDF does not require any data copying. 3) The PyArrow UDF is executed in a vectorized fashion, processing chunks of data instead of individual rows.

Python UDFs vs. External Functions

Here we compare the usage of a Python UDF with an external function. In this case, we have a function that calculates the sum of the lengths of all strings in a column. You can find the code used for this benchmark section in a GitHub Gist.

```
import duckdb
import pyarrow as pa

# Function used in UDF
def string_length_arrow(x):
     tuples = len(x)
     values = [len(i.as_py()) if i.as_py() != None else 0 for i in x]
     array = pa.array(values, type=pa.int32(), size=tuples)
     return array


# Same Function but external to the database
def exec_external(con):
     arrow_table = con.sql("SELECT i FROM strings tbl(i)").arrow()
     arrow_column = arrow_table['i']
     tuples = len(arrow_column)
     values = [len(i.as_py()) if i.as_py() != None else 0 for i in arrow_column]
     array = pa.array(values, type=pa.int32(), size=tuples)
     arrow_tbl = pa.Table.from_arrays([array], names=['i'])
     return con.sql("SELECT sum(i) FROM arrow_tbl").fetchall()


con = duckdb.connect()
con.create_function('strlen_arrow', string_length_arrow, ['VARCHAR'], int, type='arrow')

con.sql("""
     SELECT
          CASE WHEN i != 0 AND i % 42 = 0
          THEN
               NULL
          ELSE
               repeat(chr((65 + (i % 26))::INTEGER), (4 + (i % 12))) END
          FROM range(10000000) tbl(i);
""").to_view("strings")

con.sql("SELECT sum(strlen_arrow(i)) FROM strings tbl(i)").fetchall()

exec_external(con)
```

Name	Time (s)	Peak memory consumption (MB)
External	5.65	584.032
UDF	5.63	112.848

Here we can see that there is no significant regression in performance when utilizing UDFs. However, you still have the benefits of safer execution and the utilization of SQL. In our example, we can also notice that the external function materializes the entire query, resulting in a 5× higher peak memory consumption compared to the UDF approach.

Conclusions and Further Development

Scalar Python UDFs are now supported in DuckDB, marking a significant milestone in extending the functionality of the database. This enhancement empowers users to perform complex computations using a high-level language. Additionally, Python UDFs can leverage DuckDB's zero-copy integration with Arrow, eliminating data transfer costs and ensuring efficient query execution.

While the introduction of Python UDFs is a major step forward, our work in this area is ongoing. Our roadmap includes the following focus areas:

1. **Aggregate/Table-Producing UDFs**: Currently, users can create Scalar UDFs, but we are actively working on supporting Aggregation Functions (which perform calculations on a set of values and return a single result) and Table-Producing Functions (which return tables without limitations on the number of columns and rows).
2. **Types**: Scalar Python UDFs currently support most DuckDB types, with the exception of ENUM types and BIT types. We are working towards expanding the type support to ensure comprehensive functionality.

If you encounter any problems using our Python UDFs, please open an issue in DuckDB's issue tracker.

DuckDB ADBC – Zero-Copy Data Transfer via Arrow Database Connectivity

Publication date: 2023-08-04

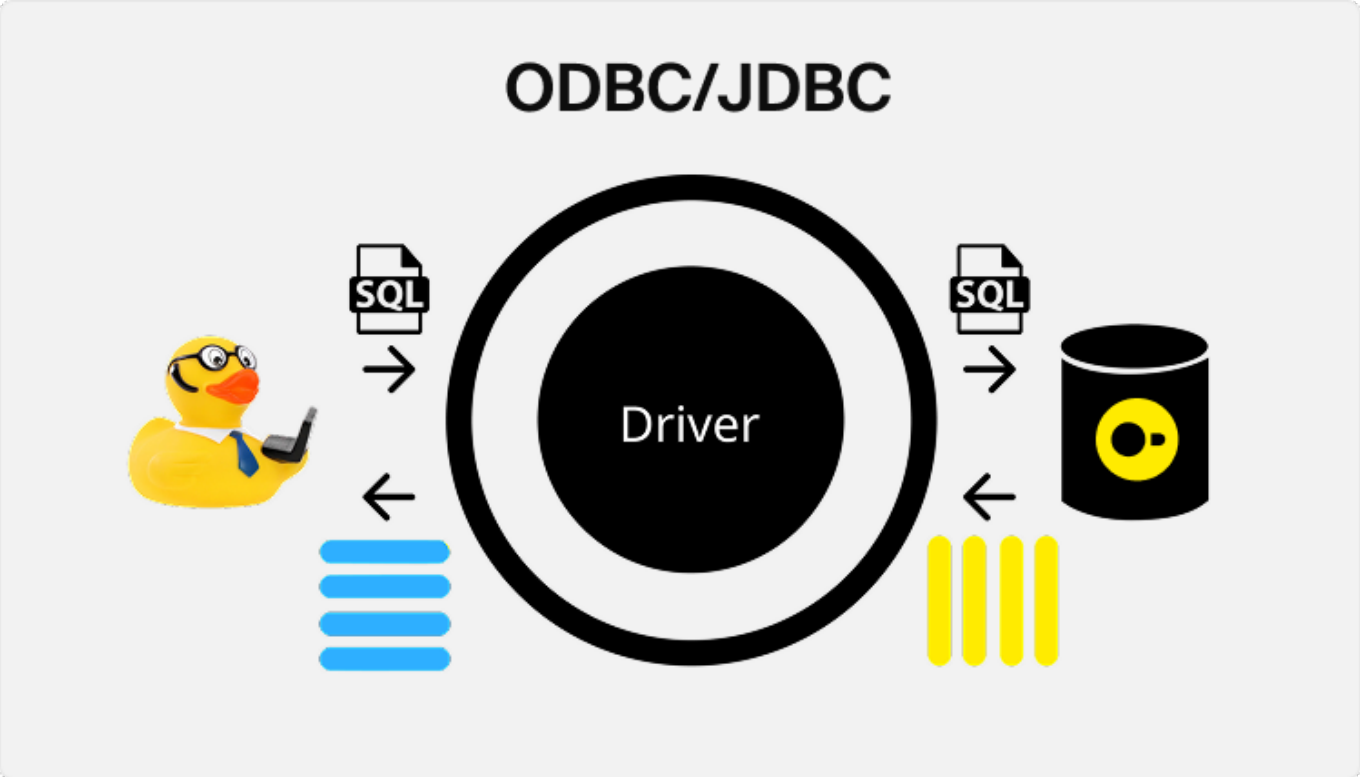
Author: Pedro Holanda

TL;DR: DuckDB has added support for Arrow Database Connectivity (ADBC), an API standard that enables efficient data ingestion and retrieval from database systems, similar to Open Database Connectivity (ODBC) interface. However, unlike ODBC, ADBC specifically caters to the columnar storage model, facilitating fast data transfers between a columnar database and an external application.

Database interface standards allow developers to write application code that is independent of the underlying database management system (DBMS) being used. DuckDB has supported two standards that have gained popularity in the past few decades: the core interface of ODBC and Java Database Connectivity (JDBC). Both interfaces are designed to fully support database connectivity and management, with JDBC being catered for the Java environment. With these APIs, developers can query DBMS agnostically, retrieve query results, run prepared statements, and manage connections.

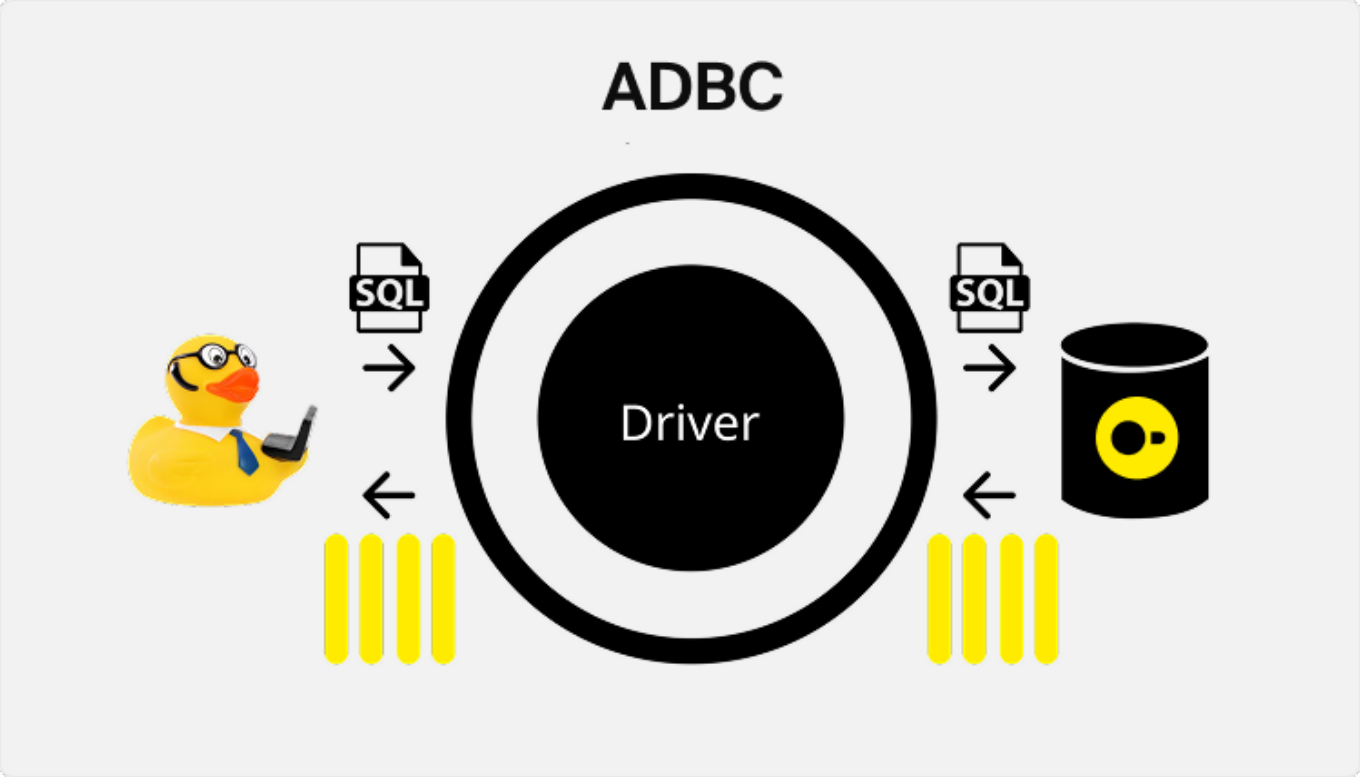
These interfaces were designed in the early 90s when row-wise database systems reigned supreme. As a result, they were primarily intended for transferring data in a row-wise format. However, in the mid-2000s, columnar-wise database systems started gaining a lot of traction due to their drastic performance advantages for data analysis (you can find myself giving a brief exemplification of this difference at EuroPython). This means that these APIs offer no support for transferring data in a columnar-wise format (or, in the case of ODBC, some support with a lot of added complexity). In practice, when analytical, column-wise systems like DuckDB make use of these APIs, converting the data between these representation formats becomes a major bottleneck.

The figure below depicts how a developer can use these APIs to query a DuckDB database. For example, developers can submit SQL queries via the API, which then uses a DuckDB driver to internally call the proper functions. A query result is then produced in DuckDB's internal columnar representation, and the driver takes care of transforming it to the JDBC or ODBC row-wise result format. This transformation has significant costs for rearranging and copying the data, quickly becoming a major bottleneck.



To overcome this transformation cost, ADBC has been proposed, with a generic API to support database operations while using the Apache Arrow memory format to send data in and out of the DBMS. DuckDB now supports the ADBC specification. Due to DuckDB's zero-copy integration with the Arrow format, using ADBC as an interface is rather efficient, since there is only a small *constant* cost to transform DuckDB query results to the Arrow format.

The figure below depicts the query execution flow when using ADBC. Note that the main difference between ODBC/JDBC is that the result does not need to be transformed to a row-wise format.



Quick Tour

For our quick tour, we will illustrate an example of round-tripping data using DuckDB-ADBC via Python. Please note that DuckDB-ADBC can also be utilized with other programming languages. Specifically, you can find C++ DuckDB-ADBC examples and tests in the DuckDB GitHub repository along with usage examples available in C++. For convenience, you can also find a ready-to-run version of this tour in a Colab notebook. If you would like to see a more detailed explanation of the DuckDB-ADBC API or view a C++ example, please refer to our documentation page.

Setup

For this example, you must have a dynamic library from the latest bleeding-edge version of DuckDB, pyarrow, and the adbc-driver-manager. The ADBC driver manager is a Python package developed by Voltron Data. The driver manager is compliant with DB-API 2.0. It wraps ADBC, making its usage more straightforward. For details on ADBC drivers, see the find the documentation of the ADBC Driver Manager.

> While DuckDB is already DB-API compliant in Python, what sets ADBC apart is that you do not need a DuckDB module installed and loaded. Additionally, unlike the DB-API, it does not utilize row-wise as its data transfer format of choice.

```
pip install pyarrow
pip install adbc-driver-manager
```

Insert Data

First, we need to include the necessary libraries that will be used in this tour. Mainly, PyArrow and the DBAPI from the ADBC Driver Manager.

```
import pyarrow
from adbc_driver_manager import dbapi
```

Next, we can create a connection via ADBC with DuckDB. This connection simply requires the path to DuckDB's driver and the entrypoint function name. DuckDB's entrypoint is `duckdb_adbc_init`. By default, connections are established with an in-memory database. However, if desired, you have the option to specify the `path` variable and connect to a local DuckDB instance, allowing you to store the data on disk. Note that these are the only variables in ADBC that are not DBMS agnostic; instead, they are set by the user, often through a configuration file.

```
con = dbapi.connect(driver="path/to/duckdb.lib", entrypoint="duckdb_adbc_init", db_kwargs={"path":
"test.db"})
```

To insert the data, we can simply call the `adbc_ingest` function with a cursor from our connection. It requires the name of the table we want to perform the ingestion to and the Arrow Python object we want to ingest. This function also has two modes: `append`, where data is appended to an existing table, and `create`, where the table does not exist yet and will be created with the input data. By default, it's set to create, so we don't need to define it here.

```
table = pyarrow.table(
     [
          ["Tenacious D", "Backstreet Boys", "Wu Tang Clan"],
          [4, 10, 7]

     ],
     names=["Name", "Albums"],
)

with con.cursor() as cursor:
     cursor.adbc_ingest("Bands", table)
```

After calling `adbc_ingest`, the table is created in the DuckDB connection and the data is fully inserted.

Read Data

To read data from DuckDB, one simply needs to use the `execute` function with a SQL query and then return the cursor's result to the desired Arrow format, such as a PyArrow Table in this example.

```
with con.cursor() as cursor:
     cursor.execute("SELECT * FROM Bands")
     cursor.fetch_arrow_table()
```

Benchmark ADBC vs ODBC

In our benchmark section, we aim to evaluate the differences in data reading from DuckDB via ADBC and ODBC. This benchmark was executed on an Apple M1 Max with 32 GB of RAM and involves outputting and inserting the `lineitem` table of TPC-H SF1. You can find the repository with the code used to run this benchmark on GitHub.

Name	Time (s)
ODBC	28.149
ADBC	0.724

The time difference between ODBC and ADBC is 38×. This significant contrast results from the extra allocations and copies that exist in ODBC.

Conclusions

DuckDB now supports the ADBC standard for database connection. ADBC is particularly efficient when combined with DuckDB, thanks to its use of the Arrow zero-copy integration.

ADBC is particularly interesting because it can drastically decrease interactions between analytic systems compared to ODBC. For example, if software that already support ODBC, e.g., if MS-Excel was to implement ADBC, integrations with columnar systems like DuckDB could benefit from this significant difference in performance.

DuckDB-ADBC is currently supported via the C Interface and through the Python ADBC Driver Manager. We will add more extensive tutorials for other languages to our documentation webpage. Please feel free to let us know your preferred language for interacting with DuckDB via ADBC!

If you encounter any problems using ADBC, please open an issue in DuckDB's issue tracker.

Even Friendlier SQL with DuckDB

Publication date: 2023-08-23

Author: Alex Monahan

TL;DR: DuckDB continues to push the boundaries of SQL syntax to both simplify queries and make more advanced analyses possible. Highlights include dynamic column selection, queries that start with the FROM clause, function chaining, and list comprehensions. We boldly go where no SQL engine has gone before! For more details, see the documentation for friendly SQL features.

Who says that SQL should stay frozen in time, chained to a 1999 version of the specification? As a comparison, do folks remember what JavaScript felt like before Promises? Those didn't launch until 2012! It's clear that innovation at the programming syntax layer can have a profoundly positive impact on an entire language ecosystem.

We believe there are many valid reasons for innovation in the SQL language, among them opportunities to simplify basic queries and also

to make more dynamic analyses possible. Many of these features arose from community suggestions! Please let us know your SQL pain points on Discord or GitHub and join us as we change what it feels like to write SQL!

If you have not had a chance to read the first installment in this series, please take a quick look to the prior blog post, "Friendlier SQL with DuckDB".

The Future Is Now

The first few enhancements in this list were included in the "Ideas for the Future" section of the prior post.

Reusable Column Aliases

When working with incremental calculated expressions in a select statement, traditional SQL dialects force you to either write out the full expression for each column or create a common table expression (CTE) around each step of the calculation. Now, any column alias can be reused by subsequent columns within the same select statement. Not only that, but these aliases can be used in the where and order by clauses as well.

Old Way 1: Repeat Yourself

```
SELECT
    'These are the voyages of the starship Enterprise...' AS intro,
    instr('These are the voyages of the starship Enterprise...', 'starship')
        AS starship_loc,
    substr('These are the voyages of the starship Enterprise...',
    instr('These are the voyages of the starship Enterprise...', 'starship')
        + len('starship') + 1) AS trimmed_intro;
```

Old Way 2: All the CTEs

```
WITH intro_cte AS (
    SELECT
        'These are the voyages of the starship Enterprise...' AS intro
), starship_loc_cte AS (
    SELECT
        intro,
        instr(intro, 'starship') AS starship_loc
    FROM intro_cte
)
SELECT
    intro,
    starship_loc,
    substr(intro, starship_loc + len('starship') + 1) AS trimmed_intro
FROM starship_loc_cte;
```

New Way

```
SELECT
     'These are the voyages of the starship Enterprise...' AS intro,
     instr(intro, 'starship') AS starship_loc,
     substr(intro, starship_loc + len('starship') + 1) AS trimmed_intro;
```

intro	starship_loc	trimmed_intro
These are the voyages of the starship Enterprise...	30	Enterprise...

Dynamic Column Selection

Databases typically prefer strictness in column definitions and flexibility in the number of rows. This can help by enforcing data types and recording column level metadata. However, in data science workflows and elsewhere, it is very common to dynamically generate columns (for example during feature engineering).

No longer do you need to know all of your column names up front! DuckDB can select and even modify columns based on regular expression pattern matching, EXCLUDE or REPLACE modifiers, and even lambda functions (see the section on lambda functions below for details!).

Let's take a look at some facts gathered about the first season of Star Trek. Using DuckDB's `httpfs` extension, we can query a CSV dataset directly from GitHub. It has several columns so let's DESCRIBE it.

```
INSTALL httpfs;
LOAD httpfs;

CREATE TABLE trek_facts AS
    SELECT *
    FROM 'https://blobs.duckdb.org/data/Star_Trek-Season_1.csv';

DESCRIBE trek_facts;
```

column_name	column_type	null	key	default	extra
season_num	BIGINT	YES	NULL	NULL	NULL
episode_num	BIGINT	YES	NULL	NULL	NULL
aired_date	DATE	YES	NULL	NULL	NULL
cnt_kirk_hookups	BIGINT	YES	NULL	NULL	NULL
cnt_downed_redshirts	BIGINT	YES	NULL	NULL	NULL
bool_aliens_almost_took_over_planet	BIGINT	YES	NULL	NULL	NULL
bool_aliens_almost_took_over_enterprise	BIGINT	YES	NULL	NULL	NULL
cnt_vulcan_nerve_pinch	BIGINT	YES	NULL	NULL	NULL
cnt_warp_speed_orders	BIGINT	YES	NULL	NULL	NULL
highest_warp_speed_issued	BIGINT	YES	NULL	NULL	NULL
bool_hand_phasers_fired	BIGINT	YES	NULL	NULL	NULL
bool_ship_phasers_fired	BIGINT	YES	NULL	NULL	NULL
bool_ship_photon_torpedoes_fired	BIGINT	YES	NULL	NULL	NULL
cnt_transporter_pax	BIGINT	YES	NULL	NULL	NULL
cnt_damn_it_jim_quote	BIGINT	YES	NULL	NULL	NULL
cnt_im_givin_her_all_shes_got_quote	BIGINT	YES	NULL	NULL	NULL
cnt_highly_illogical_quote	BIGINT	YES	NULL	NULL	NULL
bool_enterprise_saved_the_day	BIGINT	YES	NULL	NULL	NULL

`COLUMNS()` with Regular Expressions The COLUMNS expression can accept a string parameter that is a regular expression and will return all column names that match the pattern. How did warp change over the first season? Let's examine any column name that contains the word `warp`.

```
SELECT
    episode_num,
    COLUMNS('.*warp.*')
FROM trek_facts;
```

episode_num	cnt_warp_speed_orders	highest_warp_speed_issued
0	1	1
1	0	0
2	1	1
3	1	0
…	…	…
27	1	1
28	0	0
29	2	8

The COLUMNS expression can also be wrapped by other functions to apply those functions to each selected column. Let's simplify the above query to look at the maximum values across all episodes:

```
SELECT
    max(COLUMNS('.*warp.*'))
FROM trek_facts;
```

max(trek_facts.cnt_warp_speed_orders)	max(trek_facts.highest_warp_speed_issued)
5	8

We can also create a WHERE clause that applies across multiple columns. All columns must match the filter criteria, which is equivalent to combining them with AND. Which episodes had at least 2 warp speed orders and at least a warp speed level of 2?

```
SELECT
    episode_num,
    COLUMNS('.*warp.*')
FROM trek_facts
WHERE
    COLUMNS('.*warp.*') >= 2;
    -- cnt_warp_speed_orders >= 2
    -- AND
    -- highest_warp_speed_issued >= 2
```

episode_num	cnt_warp_speed_orders	highest_warp_speed_issued
14	3	7
17	2	7
18	2	8
29	2	8

COLUMNS() with EXCLUDE and REPLACE

Individual columns can also be either excluded or replaced prior to applying calculations on them. For example, since our dataset only includes season 1, we do not need to find the max of that column. It would be highly illogical.

```
SELECT
    max(COLUMNS(* EXCLUDE season_num))
FROM trek_facts;
```

max(trek_facts.episode_num)	max(trek_facts.aired_date)	max(trek_facts.cnt_kirk_hookups)	...	max(trek_facts.bool_enterprise_saved_the_day)
29	1967-04-13	2	...	1

The REPLACE syntax is also useful when applied to a dynamic set of columns. In this example, we want to convert the dates into timestamps prior to finding the maximum value in each column. Previously this would have required an entire subquery or CTE to pre-process just that single column!

```
SELECT
    max(COLUMNS(* REPLACE aired_date::timestamp AS aired_date))
FROM trek_facts;
```

max(trek_facts.season_num)	max(trek_facts.episode_num)	max(aired_date := CAST(aired_date AS TIMESTAMP))	...	max(trek_facts.bool_enterprise_saved_the_day)
1	29	1967-04-13 00:00:00	...	1

`COLUMNS()` with Lambda Functions

The most flexible way to query a dynamic set of columns is through a lambda function. This allows for any matching criteria to be applied to the names of the columns, not just regular expressions. See more details about lambda functions below.

For example, if using the `LIKE` syntax is more comfortable, we can select columns matching a `LIKE` pattern rather than with a regular expression.

```
SELECT
    episode_num,
    COLUMNS(lambda col: col LIKE '%warp%')
FROM trek_facts
WHERE
    COLUMNS(lambda col: col LIKE '%warp%') >= 2;
```

episode_num	cnt_warp_speed_orders	highest_warp_speed_issued
14	3	7
17	2	7
18	2	8
29	2	8

Automatic JSON to Nested Types Conversion

The first installment in the series mentioned JSON dot notation references as future work. However, the team has gone even further! Instead of referring to JSON-typed columns using dot notation, JSON can now be automatically parsed into DuckDB's native types for significantly faster performance, compression, as well as that friendly dot notation!

First, install and load the `httpfs` and `json` extensions if they don't come bundled with the client you are using. Then query a remote JSON file directly as if it were a table!

```
INSTALL httpfs;
LOAD httpfs;
INSTALL json;
LOAD json;
```

```
SELECT
     starfleet[10].model AS starship
FROM 'https://raw.githubusercontent.com/vlad-saling/star-trek-ipsum/master/src/content/content.json';
```

starship
USS Farragut - NCC-1647 - Ship on which James Kirk served as a phaser station operator. Attacked by the Dikironium Cloud Creature, killing half the crew. ad.

Now for some new SQL capabilities beyond the ideas from the prior post!

FROM First in SELECT Statements

When building a query, the first thing you need to know is where your data is coming FROM. Well then why is that the second clause in a SELECT statement?? No longer! DuckDB is building SQL as it should have always been – putting the FROM clause first! This addresses one of the longest standing complaints about SQL, and the DuckDB team implemented it in 2 days.

```
FROM my_table SELECT my_column;
```

Not only that, the SELECT statement can be completely removed and DuckDB will assume all columns should be SELECTed. Taking a look at a table is now as simple as:

```
FROM my_table;
-- SELECT * FROM my_table
```

Other statements like COPY are simplified as well.

```
COPY (FROM trek_facts) TO 'phaser_filled_facts.parquet';
```

This has an additional benefit beyond saving keystrokes and staying in a development flow state: autocomplete will have much more context when you begin to choose columns to query. Give the AI a helping hand!

Note that this syntax is completely optional, so your SELECT * FROM keyboard shortcuts are safe, even if they are obsolete... 🙅

Function Chaining

Many SQL blogs advise the use of CTEs instead of subqueries. Among other benefits, they are much more readable. Operations are compartmentalized into discrete chunks and they can be read in order top to bottom instead of forcing the reader to work their way inside out.

DuckDB enables the same interpretability improvement for every scalar function! Use the dot operator to chain functions together, just like in Python. The prior expression in the chain is used as the first argument to the subsequent function.

```
SELECT
     ('Make it so')
          .upper()
          .string_split(' ')
          .list_aggr('string_agg','.')
          .concat('.') AS im_not_messing_around_number_one;
```

im_not_messing_around_number_one
MAKE.IT.SO.

Now compare that with the old way...

```
SELECT
     concat(
          list_aggr(
               string_split(
                    upper('Make it stop'),
               ' '),
          'string_agg','.'),
     '.') AS oof;
```

oof
MAKE.IT.STOP.

Union by Name

DuckDB aims to blend the best of databases and dataframes. This new syntax is inspired by the concat function in Pandas. Rather than vertically stacking tables based on column position, columns are matched by name and stacked accordingly. Simply replace `UNION` with `UNION BY NAME` or `UNION ALL` with `UNION ALL BY NAME`.

For example, we had to add some new alien species proverbs in The Next Generation:

```
CREATE TABLE proverbs AS
     SELECT
          'Revenge is a dish best served cold' AS klingon_proverb
     UNION ALL BY NAME
     SELECT
          'You will be assimilated' AS borg_proverb,
          'If winning is not important, why keep score?' AS klingon_proverb;

FROM proverbs;
```

klingon_proverb	borg_proverb
Revenge is a dish best served cold	NULL
If winning is not important, why keep score?	You will be assimilated

This approach has additional benefits. As seen above, not only can tables with different column orders be combined, but so can tables with different numbers of columns entirely. This is helpful as schemas migrate, and is particularly useful for DuckDB's multi-file reading capabilities.

Insert by Name

Another common situation where column order is strict in SQL is when inserting data into a table. Either the columns must match the order exactly, or all of the column names must be repeated in two locations within the query.

Instead, add the keywords BY NAME after the table name when inserting. Any subset of the columns in the table in any order can be inserted.

```
INSERT INTO proverbs BY NAME
     SELECT 'Resistance is futile' AS borg_proverb;

SELECT * FROM proverbs;
```

klingon_proverb	borg_proverb
Revenge is a dish best served cold	NULL
If winning is not important, why keep score?	You will be assimilated
NULL	Resistance is futile

Dynamic `PIVOT` and `UNPIVOT`

Historically, databases are not well-suited for pivoting operations. However, DuckDB's `PIVOT` and `UNPIVOT` clauses can create or stack dynamic column names for a truly flexible pivoting capability! In addition to that flexibility, DuckDB also provides both the SQL standard syntax and a friendlier shorthand.

For example, let's take a look at some procurement forecast data just as the Earth-Romulan war was beginning:

```
CREATE TABLE purchases (item VARCHAR, year INTEGER, count INTEGER);

INSERT INTO purchases
    VALUES ('phasers', 2155, 1035),
           ('phasers', 2156, 25039),
           ('phasers', 2157, 95000),
           ('photon torpedoes', 2155, 255),
           ('photon torpedoes', 2156, 17899),
           ('photon torpedoes', 2157, 87492);

FROM purchases;
```

item	year	count
phasers	2155	1035
phasers	2156	25039
phasers	2157	95000
photon torpedoes	2155	255
photon torpedoes	2156	17899
photon torpedoes	2157	87492

It is easier to compare our phaser needs to our photon torpedo needs if each year's data is visually close together. Let's pivot this into a friendlier format! Each year should receive its own column (but each year shouldn't need to be specified in the query!), we want to sum up the total `count`, and we still want to keep a separate group (row) for each `item`.

```
CREATE TABLE pivoted_purchases AS
     PIVOT purchases
          ON year
          USING sum(count)
          GROUP BY item;

FROM pivoted_purchases;
```

item	2155	2156	2157
phasers	1035	25039	95000
photon torpedoes	255	17899	87492

Looks like photon torpedoes went on sale...

Now imagine the reverse situation. Scotty in engineering has been visually analyzing and manually constructing his purchases forecast. He prefers things pivoted so it's easier to read. Now you need to fit it back into the database! This war may go on for a bit, so you may need to do this again next year. Let's write an UNPIVOT query to return to the original format that can handle any year.

The COLUMNS expression will use all columns except `item`. After stacking, the column containing the column names from `pivoted_purchases` should be renamed to `year`, and the values within those columns represent the `count`. The result is the same dataset as the original.

```
UNPIVOT pivoted_purchases
     ON COLUMNS(* EXCLUDE item)
     INTO
          NAME year
          VALUE count;
```

item	year	count
phasers	2155	1035
phasers	2156	25039
phasers	2157	95000
photon torpedoes	2155	255
photon torpedoes	2156	17899
photon torpedoes	2157	87492

More examples are included as a part of our DuckDB 0.8.0 announcement post, and the `PIVOT` and `UNPIVOT` documentation pages highlight more complex queries.

Stay tuned for a future post to cover what is happening behind the scenes!

List Lambda Functions

List lambdas allow for operations to be applied to each item in a list. These do not need to be pre-defined – they are created on the fly within the query.

In this example, a lambda function is used in combination with the `list_transform` function to shorten each official ship name.

```
SELECT
     (['Enterprise NCC-1701', 'Voyager NCC-74656', 'Discovery NCC-1031'])
          .list_transform(lambda x: x.string_split(' ')[1]) AS short_name;
```

ship_name
[Enterprise, Voyager, Discovery]

Lambdas can also be used to filter down the items in a list. The lambda returns a list of booleans, which is used by the `list_filter` function to select specific items. The `contains` function is using the function chaining described earlier.

```
SELECT
     (['Enterprise NCC-1701', 'Voyager NCC-74656', 'Discovery NCC-1031'])
          .list_filter(lambda x: x.contains('1701')) AS the_original;
```

the_original
[Enterprise NCC-1701]

List Comprehensions

What if there was a simple syntax to both modify and filter a list? DuckDB takes inspiration from Python's approach to list comprehensions to dramatically simplify the above examples. List comprehensions are syntactic sugar – these queries are rewritten into lambda expressions behind the scenes!

Within brackets, first specify the transformation that is desired, then indicate which list should be iterated over, and finally include the filter criteria.

```
SELECT
     [x.string_split(' ')[1]
     FOR x IN ['Enterprise NCC-1701', 'Voyager NCC-74656', 'Discovery NCC-1031']
     IF x.contains('1701')] AS ready_to_boldly_go;
```

ready_to_boldly_go
[Enterprise]

Exploding Struct.*

A struct in DuckDB is a set of key/value pairs. Behind the scenes, a struct is stored with a separate column for each key. As a result, it is computationally easy to explode a struct into separate columns, and now it is also syntactically simple as well! This is another example of allowing SQL to handle dynamic column names.

```
WITH damage_report AS (
     SELECT {'gold_casualties':5, 'blue_casualties':15, 'red_casualties': 10000} AS casualties
)
FROM damage_report
SELECT
     casualties.*;
```

gold_casualties	blue_casualties	red_casualties
5	15	10000

Automatic Struct Creation

DuckDB exposes an easy way to convert any table into a single-column struct. Instead of SELECTing column names, SELECT the table name itself.

```
WITH officers AS (
     SELECT 'Captain' AS rank, 'Jean-Luc Picard' AS name
     UNION ALL
     SELECT 'Lieutenant Commander', 'Data'
)
FROM officers
SELECT officers;
```

officers
{'rank': Captain, 'name': Jean-Luc Picard}
{'rank': Lieutenant Commander, 'name': Data}

Union Data Type

DuckDB utilizes strong typing to provide high performance and enforce data quality. However, DuckDB is also as forgiving as possible using approaches like implicit casting to avoid always having to cast between data types.

Another way DuckDB enables flexibility is the new `UNION` data type. A `UNION` data type allows for a single column to contain multiple types of values. This can be thought of as an "opt-in" to SQLite's flexible data typing rules (the opposite direction of SQLite's recently announced strict tables).

By default DuckDB will seek the common denominator of data types when combining tables together. The below query results in a `VARCHAR` column:

```
SELECT 'The Motion Picture' AS movie UNION ALL
SELECT 2 UNION ALL
SELECT 3 UNION ALL
SELECT 4 UNION ALL
SELECT 5 UNION ALL
SELECT 6 UNION ALL
SELECT 'First Contact';
```

movie
The Motion Picture
First Contact
6
5
4
3
2

However, if a `UNION` type is used, each individual row retains its original data type. A `UNION` is defined using key-value pairs with the key as a name and the value as the data type. This also allows the specific data types to be pulled out as individual columns:

```
CREATE TABLE movies (
     movie UNION(num INTEGER, name VARCHAR)
);
INSERT INTO movies VALUES
     ('The Motion Picture'), (2), (3), (4), (5), (6), ('First Contact');

FROM movies
SELECT
     movie,
     union_tag(movie) AS type,
     movie.name,
     movie.num;
```

movie	type	name	num
The Motion Picture	name	The Motion Picture	
2	num		2
3	num		3
4	num		4
5	num		5
6	num		6
First Contact	name	First Contact	

Additional Friendly Features

Several other friendly features are worth mentioning and some are powerful enough to warrant their own blog posts.

DuckDB takes a nod from the `describe` function in Pandas and implements a `SUMMARIZE` keyword that will calculate a variety of statistics about each column in a dataset for a quick, high-level overview. Simply prepend `SUMMARIZE` to any table or `SELECT` statement.

Have a look at the correlated subqueries post to see how to use subqueries that refer to each others' columns. DuckDB's advanced optimizer improves correlated subquery performance by orders of magnitude, allowing for queries to be expressed as naturally as possible. What was once an anti-pattern for performance reasons can now be used freely!

DuckDB has added more ways to `JOIN` tables together that make expressing common calculations much easier. Some like `LATERAL`, `ASOF`, `SEMI`, and `ANTI` joins are present in other systems, but have high-performance implementations in DuckDB. DuckDB also adds a new `POSITIONAL` join that combines by the row numbers in each table to match the commonly used Pandas capability of joining on row number indexes. See the `JOIN` documentation for details, and look out for a blog post describing DuckDB's state of the art `ASOF` joins!

Summary and Future Work

DuckDB aims to be the easiest database to use. Fundamental architectural decisions to be in-process, have zero dependencies, and have strong typing contribute to this goal, but the friendliness of its SQL dialect has a strong impact as well. By extending the industry-standard PostgreSQL dialect, DuckDB aims to provide the simplest way to express the data transformations you need. These changes range from altering the ancient clause order of the `SELECT` statement to begin with `FROM`, allowing a fundamentally new way to use functions with chaining, to advanced nested data type calculations like list comprehensions. Each of these features are available in the 0.8.1 release.

Future work for friendlier SQL includes:

- Lambda functions with more than 1 argument, like `list_zip`
- Underscores as digit separators (Ex: `1_000_000` instead of `1000000`)
- Extension user experience, including autoloading
- Improvements to file globbing
- Your suggestions!

Please let us know what areas of SQL can be improved! We welcome your feedback on Discord or GitHub.

Live long and prosper! 🖖

DuckDB's AsOf Joins: Fuzzy Temporal Lookups

Publication date: 2023-09-15

Author: Richard Wesley

TL;DR: DuckDB supports AsOf Joins – a way to match nearby values. They are especially useful for searching event tables for temporal analytics.

Do you have time series data that you want to join, but the timestamps don't quite match? Or do you want to look up a value that changes over time using the times in another table? And did you end up writing convoluted (and slow) inequality joins to get your results? Then this post is for you!

What Is an AsOf Join?

Time series data is not always perfectly aligned. Clocks may be slightly off, or there may be a delay between cause and effect. This can make connecting two sets of ordered data challenging. AsOf Joins are a tool for solving this and other similar problems.

One of the problems that AsOf Joins are used to solve is finding the value of a varying property at a specific point in time. This use case is so common that it is where the name came from: *Give me the value of the property **as of this time.***

More generally, however, AsOf joins embody some common temporal analytic semantics, which can be cumbersome and slow to implement in standard SQL.

Portfolio Example

Let's start with a concrete example. Suppose we have a table of stock `prices` with timestamps:

ticker	when	price
APPL	2001-01-01 00:00:00	1
APPL	2001-01-01 00:01:00	2
APPL	2001-01-01 00:02:00	3
MSFT	2001-01-01 00:00:00	1
MSFT	2001-01-01 00:01:00	2
MSFT	2001-01-01 00:02:00	3
GOOG	2001-01-01 00:00:00	1
GOOG	2001-01-01 00:01:00	2
GOOG	2001-01-01 00:02:00	3

We have another table containing portfolio `holdings` at various points in time:

ticker	when	shares
APPL	2000-12-31 23:59:30	5.16

ticker	when	shares
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	24.13
GOOG	2000-12-31 23:59:30	9.33
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	10.58
DATA	2000-12-31 23:59:30	6.65
DATA	2001-01-01 00:00:30	17.95
DATA	2001-01-01 00:01:30	18.37

We can compute the value of each holding at that point in time by finding the most recent price before the holding's timestamp by using an AsOf Join:

```
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h ASOF JOIN prices p
  ON h.ticker = p.ticker
 AND h.when >= p.when;
```

This attaches the value of the holding at that time to each row:

ticker	when	value
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	48.26
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	21.16

It essentially executes a function defined by looking up nearby values in the `prices` table. Note also that missing `ticker` values do not have a match and don't appear in the output.

Outer AsOf Joins

Because AsOf produces at most one match from the right hand side, the left side table will not grow as a result of the join, but it could shrink if there are missing times on the right. To handle this situation, you can use an *outer* AsOf Join:

```
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h ASOF LEFT JOIN prices p
  ON h.ticker = p.ticker
 AND h.when >= p.when
ORDER BY ALL;
```

As you might expect, this will produce NULL prices and values instead of dropping left side rows when there is no ticker or the time is before the prices begin.

ticker	when	value
APPL	2000-12-31 23:59:30	
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	48.26

ticker	when	value
GOOG	2000-12-31 23:59:30	
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	21.16
DATA	2000-12-31 23:59:30	
DATA	2001-01-01 00:00:30	
DATA	2001-01-01 00:01:30	

Windowing Alternative

Standard SQL can implement this kind of join, but you need to use a window function and an inequality join. These can both be fairly expensive operations, but the query would look like this:

```
WITH state AS (
    SELECT
        ticker,
        price,
        "when",
        lead("when", 1, 'infinity')
            OVER (PARTITION BY ticker ORDER BY "when") AS end
    FROM prices
)
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h
INNER JOIN state s
        ON h.ticker = s.ticker
      AND h.when >= s.when
      AND h.when < s.end;
```

The default value of `infinity` is used to make sure there is an end value for the last row that can be compared. Here is what the `state` CTE looks like for our example:

ticker	price	when	end
APPL	1	2001-01-01 00:00:00	2001-01-01 00:01:00
APPL	2	2001-01-01 00:01:00	2001-01-01 00:02:00
APPL	3	2001-01-01 00:02:00	infinity
GOOG	1	2001-01-01 00:00:00	2001-01-01 00:01:00
GOOG	2	2001-01-01 00:01:00	2001-01-01 00:02:00
GOOG	3	2001-01-01 00:02:00	infinity
MSFT	1	2001-01-01 00:00:00	2001-01-01 00:01:00
MSFT	2	2001-01-01 00:01:00	2001-01-01 00:02:00
MSFT	3	2001-01-01 00:02:00	infinity

In the case where there is no equality condition, the planner would have to use an inequality join, which can be very expensive. And even in the equality condition case, the resulting hash join may end up with long chains of identical `ticker` keys that will all match and need pruning.

Why AsOf?

If SQL can compute AsOf joins already, why do we need a new join type? There are two big reasons: expressibility and performance. The windowing alternative is more verbose and harder to understand than the AsOf syntax, so making it easier to say what you are doing helps others (or even you!) understand what is happening.

The syntax also makes it easier for DuckDB to understand what you want and produce your results faster. The window and inequality join version loses the valuable information that the intervals do not overlap. It also prevents the query optimizer from moving the join because SQL insists that windowing happens *after* joins. By treating the operation *as a join* with *known data constraints*, DuckDB can move the join for performance and use a tailored join algorithm. The algorithm we use is to sort the right side table and then do a kind of merge join with the left side values. But unlike a standard merge join, AsOf can stop searching when it finds the first match because there is at most one match.

State Tables

You may be wondering why the Common Table Expression in the `WITH` clause was called *state*. This is because the `prices` table is really an example of what in temporal analytics is called an *event table* . The rows of an event table contain timestamps and what happened at that time (i.e., events). The events in the `prices` table are changes to the price of a stock. Another common example of an event table is a structured log file: Each row of the log records when something "happened" – usually a change to a part of the system.

Event tables are difficult to work with because each fact only has the start time. In order to know whether the fact is still true (or true at a specific time) you need the end time as well. A table with both the start and end time is called a *state table*. Converting event tables to state tables is a common temporal data preparation task, and the windowing CTE above shows how to do it in general using SQL.

Sentinel Values

One limitation of the windowing approach is that the ordering type needs to have sentinel value that can be used if it does not support `infinity`, either an unused value or `NULL`.

Both of these choices are potentially problematic. In the first case, it may not be easy to determine an upper sentinel value (suppose the ordering was a string column?) In the second case, you would need to write the condition as `h.when < s.end OR s.end IS NULL` and using an `OR` like this in a join condition makes comparisons slow and hard to optimize. Moreover, if the ordering column is already using `NULL` to indicate missing values, this option is not available.

For most state tables, there are suitable choices (e.g., large dates) but one of the advantages of AsOf is that it can avoid having to design a state table if it is not needed for the analytic task.

Event Table Variants

So far we have been using a standard type of event table where the timestamps are assumed to be the start of the state transitions. But AsOf can now use any inequality, which allows it to handle other types of event tables.

To explore this, let's use two very simple tables with no equality conditions. The build side will just have four integer "timestamps" with alphabetic values:

Time	Value
1	a
2	b
3	c
4	d

The probe table will just be the time values plus the midpoints, and we can make a table showing what value each probe time matches for greater than or equal to:

Probe	>=
0.5	
1.0	a
1.5	a
2.0	b
2.5	b
3.0	c
3.5	c
4.0	d
4.5	d

This shows us that the interval a probe value matches is in the half-open interval `[Tn, Tn+1)`.

Now let's see what happens if use strictly greater than as the inequality:

Probe	>
0.5	
1.0	
1.5	a
2.0	a
2.5	b
3.0	b
3.5	c
4.0	c
4.5	d

Now we can see that the interval a probe value matches is in the half-open interval `(Tn, Tn+1]`. The only difference is that the interval is closed at the end instead of the beginning. This means that for this inequality type, the time is not part of the interval.

What if the inequality goes in the other direction, say less than or equal to?

Probe	<=
0.5	a
1.0	a
1.5	b
2.0	b
2.5	c
3.0	c
3.5	d

Probe	<=
4.0	d
4.5	

Again, we have half-open intervals, but this time we are matching the *previous* interval `(Tn-1, Tn]`. One way to interpret this is that the times in the build table are the *end* of the interval, instead of the beginning. Also, unlike greater than or equal to, the interval is closed at the end instead of the beginning. Adding this to what we found for strictly greater than, we can interpret this as meaning that the lookup times are part of the interval when non-strict inequalities are used.

We can check this by looking at the last inequality: strictly less than:

Probe	<
0.5	a
1.0	b
1.5	b
2.0	c
2.5	c
3.0	d
3.5	d
4.0	
4.5	

In this case the matching intervals are `[Tn-1, Tn)`. This is a strict inequality, so the table time is not in the interval, and it is a less than, so the time is the end of the interval.

To sum up, here is the full list:

Inequality	Interval
>	(Tn, Tn+1]
>=	[Tn, Tn+1)
<=	(Tn-1, Tn]
<	[Tn-1, Tn)

We now have two natural interpretations of what the inequalities mean:

- The greater (resp. less) than inequalities mean the time is the beginning (resp. end) of the interval.
- The strict (resp. non-strict) inequalities mean the time is excluded from (resp. included in) the interval.

So if we know whether the time marks the start or the end of the event, and whether the time is include or excluded, we can choose the appropriate AsOf inequality.

Usage

So far we have been explicit about specifying the conditions for AsOf, but SQL also has a simplified join condition syntax for the common case where the column names are the same in both tables. This syntax uses the `USING` keyword to list the fields that should be compared for equality. AsOf also supports this syntax, but with two restrictions:

- The last field is the inequality
- The inequality is >= (the most common case)

Our first query can then be written as:

```
SELECT ticker, h.when, price * shares AS value
FROM holdings h
ASOF JOIN prices p
    USING(ticker, "when");
```

Be aware that if you don't explicitly list the columns in the `SELECT`, the ordering field value will be the probe value, not the build value. For a natural join, this is not an issue because all the conditions are equalities, but for AsOf, one side has to be chosen. Since AsOf can be viewed as a lookup function, it is more natural to return the "function arguments" than the function internals.

Under the Hood

What an AsOf Join is really doing is allowing you to treat an event table as a state table for join operations. By knowing the semantics of the join, it can avoid creating a full state table and be more efficient than a general inequality join.

Let's start by looking at how the windowing version works. Remember that we used this query to convert the event table to a state table:

```
WITH state AS (
    SELECT
        ticker,
        price,
        "when",
        lead("when", 1, 'infinity')
            OVER (PARTITION BY ticker ORDER BY "when") AS end
    FROM prices
);
```

The state table CTE is created by hash partitioning the table on `ticker`, sorting on when and then computing another column that is just when shifted down by one. The join is then implemented with a hash join on `ticker` and two comparisons on when.

If there was no `ticker` column (e.g., the prices were for a single item) then the join would be implemented using our inequality join operator, which would materialise and sort both sides because it doesn't know that the ranges are disjoint.

The AsOf operator uses all three operator pipeline APIs to consolidate and collect rows. During the `sink` phase, AsOf hash partitions and sorts the right hand side to make a temporary state table. (In fact it uses the same code as Window, but without unnecessarily materialising the end column.) During the `operator` phase, it filters out (or returns) rows that cannot match because of `NULL` values in the predicate expressions, and then hash partitions and sorts the remaining rows into a cache. Finally, during the `source` phase, it matches hash partitions and then merge joins the sorted values within each hash partition.

Benchmarks

Because AsOf joins can be implemented in various ways using standard SQL queries, benchmarking is really about comparing the various alternatives.

One alternative is a debugging PRAGMA for AsOf called `debug_asof_iejoin`, which implements the join using Window and IEJoin. This allows us to easily toggle between the implementations and compare runtimes.

Other alternatives combine equi-joins and window functions. The equi-join is used to implement the equality matching conditions, and the window is used to select the closest inequality. We will now look at two different windowing techniques and compare their performance. If you wish to skip this section, the bottom line is that while they are sometimes a bit faster, the AsOf join has the most consistent behavior of all the algorithms.

Window as State Table

The first benchmark compares a hash join with a state table. It probes a 5M row table of values built from 100K timestamps and 50 partitioning keys using a self-join where only 50% of the keys are present and the timestamps have been shifted to be halfway between the originals:

```
CREATE OR REPLACE TABLE build AS (
    SELECT k, '2001-01-01 00:00:00'::TIMESTAMP + INTERVAL (v) MINUTE AS t, v
    FROM range(0, 100_000) vals(v), range(0, 50) keys(k)
);

CREATE OR REPLACE TABLE probe AS (
    SELECT k * 2 AS k, t - INTERVAL (30) SECOND AS t
    FROM build
);
```

The `build` table looks like this:

k	t	v
0	2001-01-01 00:00:00	0
0	2001-01-01 00:01:00	1
0	2001-01-01 00:02:00	2
0	2001-01-01 00:03:00	3
…	…	…

and the probe table looks like this (with only even values for k):

k	t
0	2000-12-31 23:59:30
0	2001-01-01 00:00:30
0	2001-01-01 00:01:30
0	2001-01-01 00:02:30
0	2001-01-01 00:03:30
…	…

The benchmark just does the join and sums up the v column:

```
SELECT sum(v)
FROM probe
ASOF JOIN build USING(k, t);
```

The debugging PRAGMA does not allow us to use a hash join, but we can create the state table in a CTE again and use an inner join:

```
-- Hash Join implementation
WITH state AS (
    SELECT k,
      t AS begin,
      v,
      lead(t, 1, 'infinity'::TIMESTAMP) OVER (PARTITION BY k ORDER BY t) AS end
    FROM build
)
SELECT sum(v)
FROM probe p
```

```
INNER JOIN state s
        ON p.t >= s.begin
       AND p.t < s.end
       AND p.k = s.k;
```

This works because the planner assumes that equality conditions are more selective than inequalities and generates a hash join with a filter.

Running the benchmark, we get results like this:

Algorithm	Median of 5
AsOf	0.425 s
IEJoin	3.522 s
State Join	192.460 s

The runtime improvement of AsOf over IEJoin here is about 9×. The horrible performance of the Hash Join is caused by the long (100K) bucket chains in the hash table.

The second benchmark tests the case where the probe side is about 10× smaller than the build side:

```
CREATE OR REPLACE TABLE probe AS
    SELECT k,
      '2021-01-01T00:00:00'::TIMESTAMP +
          INTERVAL (random() * 60 * 60 * 24 * 365) SECOND AS t,
    FROM range(0, 100_000) tbl(k);

CREATE OR REPLACE TABLE build AS
    SELECT r % 100_000 AS k,
      '2021-01-01T00:00:00'::TIMESTAMP +
          INTERVAL (random() * 60 * 60 * 24 * 365) SECOND AS t,
      (random() * 100_000)::INTEGER AS v
    FROM range(0, 1_000_000) tbl(r);

SELECT sum(v)
FROM probe p
ASOF JOIN build b
  ON p.k = b.k
 AND p.t >= b.t

-- Hash Join Version
WITH state AS (
  SELECT k,
    t AS begin,
    v,
    lead(t, 1, 'infinity'::TIMESTAMP)
        OVER (PARTITION BY k ORDER BY t) AS end
  FROM build
)
SELECT sum(v)
FROM probe p
INNER JOIN state s
        ON p.t >= s.begin
       AND p.t < s.end
       AND p.k = s.k;
```

Algorithm	Median of 5 runs
State Join	0.065 s
AsOf	0.077 s

Algorithm	Median of 5 runs
IEJoin	49.508 s

Now the runtime improvement of AsOf over IEJoin is huge (~500×) because it can leverage the partitioning to eliminate almost all of the equality mismatches.

The Hash Join implementation does much better here because the optimizer notices that the probe side is smaller and builds the hash table on the "probe" table. Also, the probe values here are unique, so the hash table chains are minimal.

Window with Ranking

Another way to use the window operator is to:

- Join the tables on the equality predicates
- Filter to pairs where the build time is before the probe time
- Partition the result on both the equality keys *and* the probe timestamp
- Sort the partitions on the build timestamp *descending*
- Filter out all value except rank 1 (i.e., the largest build time <= the probe time)

The query looks like:

```
WITH win AS (
    SELECT p.k, p.t, v,
        rank() OVER (PARTITION BY p.k, p.t ORDER BY b.t DESC) AS r
    FROM probe p INNER JOIN build b
      ON p.k = b.k
    AND p.t >= b.t
    QUALIFY r = 1
)
SELECT k, t, v
FROM win;
```

The advantage of this windowing query is that it does not require sentinel values, so it can work with any data type. The disadvantage is that it creates many more partitions because it includes both timestamps, which requires more complex sorting. Moreover, because it applies the window *after* the join, it can produce huge intermediates that can result in external sorting and expensive out-of-memory operations.

For this benchmark, we will be using three build tables, and two probe tables, all containing 10K integer equality keys. The probe tables have either 1 or 15 timestamps per key:

```
CREATE OR REPLACE TABLE probe15 AS
    SELECT k, t
    FROM range(10_000) cs(k),
         range('2022-01-01'::TIMESTAMP, '2023-01-01'::TIMESTAMP, INTERVAL 26 DAY) ts(t);

CREATE OR REPLACE TABLE probe1 AS
    SELECT k, '2022-01-01'::TIMESTAMP t
    FROM range(10_000) cs(k);
```

The build tables are much larger and have approximately 10/100/1000× the number of entries as the 15 element tables:

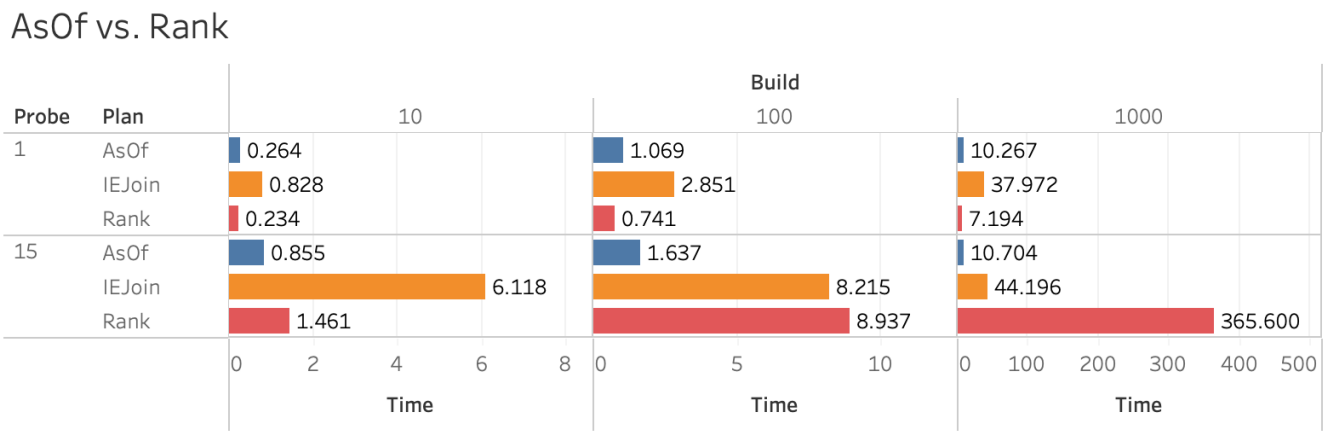
```
-- 10:1
CREATE OR REPLACE TABLE build10 AS
    SELECT k, t, (random() * 1000)::DECIMAL(7, 2) AS v
    FROM range(10_000) ks(k),
         range('2022-01-01'::TIMESTAMP, '2023-01-01'::TIMESTAMP, INTERVAL 59 HOUR) ts(t);

-- 100:1
CREATE OR REPLACE TABLE build100 AS
    SELECT k, t, (random() * 1000)::DECIMAL(7, 2) AS v
```

```
    FROM range(10_000) ks(k),
         range('2022-01-01'::TIMESTAMP, '2023-01-01'::TIMESTAMP, INTERVAL 350 MINUTE) ts(t);

-- 1000:1
CREATE OR REPLACE TABLE build1000 AS
    SELECT k, t, (random() * 1000)::DECIMAL(7, 2) AS v
    FROM range(10_000) ks(k),
         range('2022-01-01'::TIMESTAMP, '2023-01-01'::TIMESTAMP, INTERVAL 35 MINUTE) ts(t);
```

The AsOf join queries are:

```
-- AsOf/IEJoin
SELECT p.k, p.t, v
FROM probe p ASOF JOIN build b
  ON p.k = b.k
 AND p.t >= b.t
ORDER BY 1, 2;

-- Rank
WITH win AS (
    SELECT p.k, p.t, v,
        rank() OVER (PARTITION BY p.k, p.t ORDER BY b.t DESC)  AS r
    FROM probe p INNER JOIN build b
      ON p.k = b.k
    AND p.t >= b.t
    QUALIFY r = 1
)
SELECT k, t, v
FROM win
ORDER BY 1, 2;
```

The results are shown here:



(Median of 5 except for Rank/15/1000).

- For all ratios with 15 probes, AsOf is the most performant.
- For small ratios with 15 probes, Rank beats IEJoin (both with windowing), but by 100:1 it is starting to explode.
- For single element probes, Rank is most effective, but even there, its edge over AsOf is only about 50% at scale.

This shows that AsOf could be possibly be improved upon, but predicting where that happens would be tricky, and getting it wrong would have enormous costs.

Future Work

DuckDB can now execute AsOf joins for all inequality types with reasonable performance. In some cases, the performance gain is several orders of magnitude over the standard SQL versions – even with our fast inequality join operator.

While the current AsOf operator is completely general, there are a couple of planning optimisations that could be applied here.

- When there are selective equality conditions, it is likely that a hash join with filtering against a materialised state table would be significantly faster. If we can detect this and suitable sentinel values are available, the planner could choose to use a hash join instead of the default AsOf implementation.
- There are also use cases where the probe table is much smaller than the build table, along with equality conditions, and performing a hash join against the *probe* table could yield significant performance improvements.

Nevertheless, remember that one of the advantages of SQL is that it is a declarative language:
You specify *what* you want and leave it up to the database to figure out *how*. Now that we have defined the semantics of the AsOf join, you the user can write queries saying this is *what* you want – and we are free to keep improving the *how*!

Happy Joining!

One of the most interesting parts of working on DuckDB is that it stretches the traditional SQL model of unordered data. DuckDB makes it easy to query *ordered* data sets such as data frames and Parquet files, and when you have data like that, you expect to be able to do ordered analysis! Implementing Fast Sorting, Fast Windowing and Fast AsOf joins is how we are making this expectation a reality.

Announcing DuckDB 0.9.0

Publication date: 2023-09-26

Authors: Mark Raasveldt, Hannes Mühleisen

The DuckDB team is happy to announce the latest DuckDB release (0.9.0). This release is named Undulata after the Yellow-billed duck native to Africa.

To install the new version, please visit the installation guide. The full release notes can be found on GitHub.

What's New in 0.9.0

There have been too many changes to discuss them each in detail, but we would like to highlight several particularly exciting features!

- Out-of-Core Hash Aggregate
- Storage Improvements
- Index Improvements

- DuckDB-Wasm Extensions
- Extension Auto-Loading
- Improved AWS Support
- Iceberg Support
- Azure Support
- PySpark-Compatible API

Below is a summary of those new features with examples, starting with a change in our SQL dialect that is designed to produce more intuitive results by default.

Breaking SQL Changes

Struct Auto-Casting. Previously the names of struct entries were ignored when determining auto-casting rules. As a result, struct field names could be silently renamed. Starting with this release, this will result in an error instead.

```
CREATE TABLE structs (s STRUCT(i INTEGER));
INSERT INTO structs VALUES ({'k': 42});
```

```
Mismatch Type Error: Type STRUCT(k INTEGER) does not match with STRUCT(i INTEGER). Cannot cast STRUCTs with different names
```

Unnamed structs constructed using the `ROW` function can still be inserted into struct fields.

```
INSERT INTO structs VALUES (ROW(42));
```

Core System Improvements

Out-of-Core Hash Aggregates and **Hash Aggregate Performance Improvements.** When working with large data sets, memory management is always a potential pain point. By using a streaming execution engine and buffer manager, DuckDB supports many operations on larger than memory data sets. DuckDB also aims to support queries where *intermediate* results do not fit into memory by using disk-spilling techniques.

In this release, support for disk-spilling techniques is further extended through the support for out-of-core hash aggregates. Now, hash tables constructed during `GROUP BY` queries or `DISTINCT` operations that do not fit in memory due to a large number of unique groups will spill data to disk instead of throwing an out-of-memory exception. Due to the clever use of radix partitioning, performance degradation is gradual, and performance cliffs are avoided. Only the subset of the table that does not fit into memory will be spilled to disk.

The performance of our hash aggregate has also improved in general, especially when there are many groups. For example, we compute the number of unique rows in a data set with 30 million rows and 15 columns by using the following query:

```
SELECT count(*) FROM (SELECT DISTINCT * FROM tbl);
```

If we keep all the data in memory, the query should use around 6 GB. However, we can still complete the query if less memory is available. In the table below, we can see how the runtime is affected by lowering the memory limit:

memory limit	v0.8.1	v0.9.0
10 GB	8.52 s	2.91 s
9 GB	8.52 s	3.45 s
8 GB	8.52 s	3.45 s
7 GB	8.52 s	3.47 s
6 GB	OOM	3.41 s
5 GB	OOM	3.67 s
4 GB	OOM	3.87 s

memory limit	v0.8.1	v0.9.0
3 GB	OOM	4.20 s
2 GB	OOM	4.39 s
1 GB	OOM	4.91 s

Compressed Materialization. DuckDB's streaming execution engine has a low memory footprint, but more memory is required for operations such as grouped aggregation. The memory footprint of these operations can be reduced by compression. DuckDB already uses many compression techniques in its storage format, but many of these techniques are too costly to use during query execution. However, certain lightweight compression techniques are so cheap that the benefit of the reducing memory footprint outweight the cost of (de)compression.

In this release, we add support for compression of strings and integer types right before data goes into the grouped aggregation and sorting operators. By using statistics, both types are compressed to the smallest possible integer type. For example, if we have the following table:

id int32	name varchar
300	alice
301	bob
302	eve
303	mallory
304	trent

The `id` column uses a 32-bit integer. From our statistics we know that the minimum value is 300, and the maximum value is 304. We can subtract 300 and cast to an 8-bit integer instead, reducing the width from 4 bytes down to 1.

The `name` column uses our internal string type, which is 16 bytes wide. However, our statistics tell us that the longest string here is only 7 bytes. We can fit this into a 64-bit integer like so:

```
alice   -> alice005
bob     -> bob00003
eve     -> eve00003
mallory -> mallory7
trent   -> trent005
```

This reduces the width from 16 bytes down to 8. To support sorting of compressed strings, we flip the bytes on big-endian machines so that our comparison operators are still correct:

```
alice005 -> 500ecila
bob00003 -> 30000bob
eve00003 -> 30000eve
mallory7 -> 7yrollam
trent005 -> 500tnert
```

By reducing the size of query intermediates, we can prevent/reduce spilling data to disk, reducing the need for costly I/O operations, thereby improving query performance.

Window Function Performance Improvements (#7831, #7996, #8050, #8491). This release features many improvements to the performance of Window functions due to improved vectorization of the code, more re-use of partial aggregates and improved parallelism through work stealing of tasks. As a result, performance of Window functions has improved significantly, particularly in scenarios where there are no or few partitions.

```
SELECT
    sum(driver_pay) OVER (
        ORDER BY dropoff_datetime ASC
        RANGE BETWEEN
```

```
        INTERVAL 3 DAYS PRECEDING AND
        INTERVAL 0 DAYS FOLLOWING
    )
FROM tripdata;
```

Version	Run time
v0.8.0	33.8 s
v0.9.0	3.8 s

Storage Improvements

Vacuuming of Deleted Row Groups. Starting with this release, when deleting data using `DELETE` statements, entire row groups that are deleted will be automatically cleaned up. Support is also added to truncate the database file on checkpoint which allows the database file to be reduced in size after data is deleted. Note that this only occurs if the deleted row groups are located at the end of the file. The system does not yet move around data in order to reduce the size of the file on disk. Instead, free blocks earlier on in the file are re-used to store later data.

Index Storage Improvements (#7930, #8112, #8437, #8703). Many improvements have been made to both the in-memory footprint, and the on-disk footprint of ART indexes. In particular for indexes created to maintain `PRIMARY KEY`, `UNIQUE` or `FOREIGN KEY` constraints the storage and in-memory footprint is drastically reduced.

```
CREATE TABLE integers (i INTEGER PRIMARY KEY);
INSERT INTO integers FROM range(10000000);
```

Version	Size
v0.8.0	278 MB
v0.9.0	78 MB

In addition, due to improvements in the manner in which indexes are stored on disk they can now be written to disk incrementally instead of always requiring a full rewrite. This allows for much quicker checkpointing for tables that have indexes.

Extensions

Extension Auto-Loading. Starting from this release, DuckDB supports automatically installing and loading of trusted extensions. As many workflows rely on core extensions that are not bundled, such as `httpfs`, many users found themselves having to remember to load the required extensions up front. With this change, the extensions will instead be automatically loaded (and optionally installed) when used in a query.

For example, in Python the following code snippet now works without needing to explicitly load the `httpfs` or `json` extensions.

```
import duckdb

duckdb.sql("FROM 'https://raw.githubusercontent.com/duckdb/duckdb/main/data/json/example_n.ndjson'")
```

The set of autoloadable extensions is limited to official extensions distributed by DuckLabs, and can be found here. The behavior can also be disabled using the `autoinstall_known_extensions` and `autoload_known_extensions` settings, or through the more general `enable_external_access` setting. See the configuration options.

DuckDB-Wasm Extensions. This release adds support for loadable extensions to DuckDB-Wasm. Previously, any extensions that you wanted to use with the Wasm client had to be baked in. With this release, extensions can be loaded dynamically instead. When an extension is loaded, the Wasm bundle is downloaded and the functionality of the extension is enabled. Give it a try in our Wasm shell.

```
LOAD inet;
SELECT '127.0.0.1'::INET;
```

AWS Extension. This release marks the launch of the DuckDB AWS extension. This extension contains AWS related features that rely on the AWS SDK. Currently, the extension contains one function, `LOAD_AWS_CREDENTIALS`, which uses the AWS Credential Provider Chain to automatically fetch and set credentials:

```
CALL load_aws_credentials();
SELECT * FROM 's3://some-bucket/that/requires/authentication.parquet';
```

See the documentation for more information.

Experimental Iceberg Extension. This release marks the launch of the DuckDB Iceberg extension. This extension adds support for reading tables stored in the Iceberg format.

```
SELECT count(*)
FROM iceberg_scan('data/iceberg/lineitem_iceberg', allow_moved_paths = true);
```

See the documentation for more information.

Experimental Azure Extension. This release marks the launch of the DuckDB Azure extension. This extension allows for DuckDB to natively read data stored on Azure, in a similar manner to how it can read data stored on S3.

```
SET azure_storage_connection_string = '<your_connection_string>';
SELECT * FROM 'azure://<my_container>/*.csv';
```

See the documentation for more information.

Clients

Experimental PySpark API. This release features the addition of an experimental Spark API to the Python client. The API aims to be fully compatible with the PySpark API, allowing you to use the Spark API as you are familiar with but while utilizing the power of DuckDB. All statements are translated to DuckDB's internal plans using our relational API and executed using DuckDB's query engine.

```
from duckdb.experimental.spark.sql import SparkSession as session
from duckdb.experimental.spark.sql.functions import lit, col
import pandas as pd

spark = session.builder.getOrCreate()

pandas_df = pd.DataFrame({
    'age': [34, 45, 23, 56],
    'name': ['Joan', 'Peter', 'John', 'Bob']
})

df = spark.createDataFrame(pandas_df)
df = df.withColumn(
    'location', lit('Seattle')
)
res = df.select(
    col('age'),
    col('location')
).collect()

print(res)
#[
#    Row(age=34, location='Seattle'),
#    Row(age=45, location='Seattle'),
#    Row(age=23, location='Seattle'),
#    Row(age=56, location='Seattle')
#]
```

Note that the API is currently experimental and features are still missing. We are very interested in feedback. Please report any functionality that you are missing, either through Discord or on GitHub.

Final Thoughts

The full release notes can be found on GitHub. We would like to thank all of the contributors for their hard work on improving DuckDB.

DuckDB's CSV Sniffer: Automatic Detection of Types and Dialects

Publication date: 2023-10-27

Author: Pedro Holanda

TL;DR: DuckDB is primarily focused on performance, leveraging the capabilities of modern file formats. At the same time, we also pay attention to flexible, non-performance-driven formats like CSV files. To create a nice and pleasant experience when reading from CSV files, DuckDB implements a CSV sniffer that automatically detects CSV dialect options, column types, and even skips dirty data. The sniffing process allows users to efficiently explore CSV files without needing to provide any input about the file format.

There are many different file formats that users can choose from when storing their data. For example, there are performance-oriented binary formats like Parquet, where data is stored in a columnar format, partitioned into row groups, and heavily compressed. However, Parquet is known for its rigidity, requiring specialized systems to read and write these files.

On the other side of the spectrum, there are files with the CSV (comma-separated values) format, which I like to refer to as the 'Woodstock of data'. CSV files offer the advantage of flexibility; they are structured as text files, allowing users to manipulate them with any text editor, and nearly any data system can read and execute queries on them.

However, this flexibility comes at a cost. Reading a CSV file is not a trivial task, as users need a significant amount of prior knowledge about the file. For instance, DuckDB's CSV reader offers more than 25 configuration options. I've found that people tend to think I'm not working hard enough if I don't introduce at least three new options with each release. *Just kidding.* These options include specifying the delimiter, quote and escape characters, determining the number of columns in the CSV file, and identifying whether a header is present while also defining column types. This can slow down an interactive data exploration process, and make analyzing new datasets a cumbersome and less enjoyable task.

One of the raison d'être of DuckDB is to be pleasant and easy to use, so we don't want our users to have to fiddle with CSV files and input options manually. Manual input should be reserved only for files with rather unusual choices for their CSV dialect (where a dialect comprises the combination of the delimiter, quote, escape, and newline values used to create that file) or for specifying column types.

Automatically detecting CSV options can be a daunting process. Not only are there many options to investigate, but their combinations can easily lead to a search space explosion. This is especially the case for CSV files that are not well-structured. Some might argue that CSV files have a specification, but the truth of the matter is that the "specification" changes as soon as a single system is capable of reading a flawed file. And, oh boy, I've encountered my fair share of semi-broken CSV files that people wanted DuckDB to read in the past few months.

DuckDB implements a multi-hypothesis CSV sniffer that automatically detects dialects, headers, date/time formats, column types, and identifies dirty rows to be skipped. Our ultimate goal is to automatically read anything resembling a CSV file, to never give up and never let you down! All of this is achieved without incurring a substantial initial cost when reading CSV files. In the bleeding edge version, the sniffer runs when reading a CSV file by default. Note that the sniffer will always prioritize any options set by the user (e.g., if the user sets `,` as the delimiter, the sniffer won't try any other options and will assume that the user input is correct).

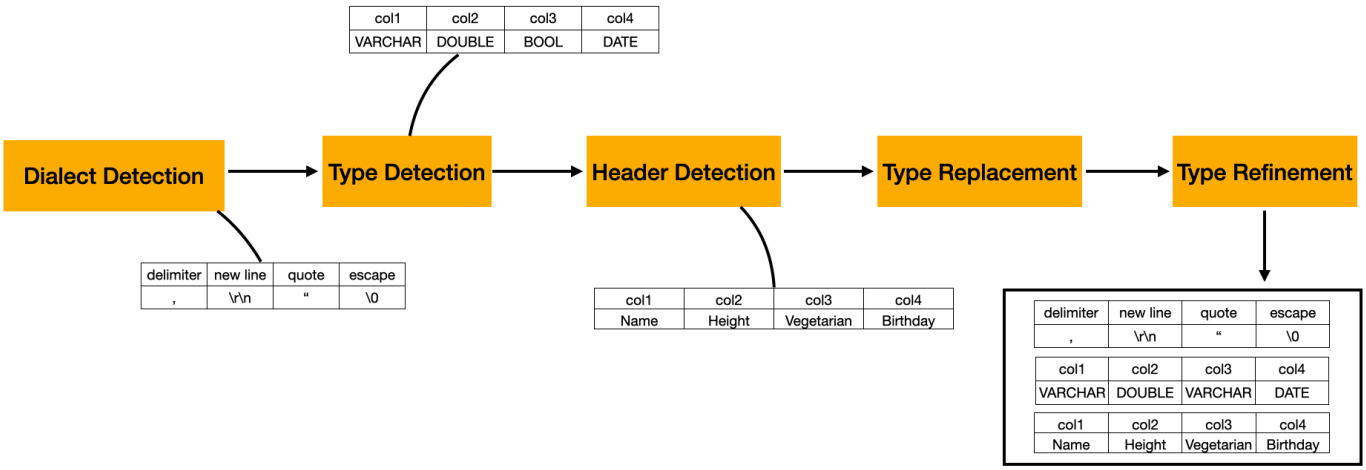
In this blog post, I will explain how the current implementation works, discuss its performance, and provide insights into what comes next!

DuckDB's Automatic Detection

The process of parsing CSV files is depicted in the figure below. It currently consists of five different phases, which will be detailed in the next sections.

The CSV file used in the overview example is as follows:

```
Name, Height, Vegetarian, Birthday
"Pedro", 1.73, False, 30-07-92
... imagine 2048 consistent rows ...
"Mark", 1.72, N/A, 20-09-92
```



In the first phase, we perform *Dialect Detection*, where we select the dialect candidates that generate the most per-row columns in the CSV file while maintaining consistency (i.e., not exhibiting significant variations in the number of columns throughout the file). In our example, we can observe that, after this phase, the sniffer successfully detects the necessary options for the delimiter, quotes, escapes, and new line delimiters.

The second phase, referred to as *Type Detection*, involves identifying the data types for each column in our CSV file. In our example, our sniffer recognizes four column types: `VARCHAR`, `DOUBLE`, `BOOL`, and `DATE`.

The third step, known as *Header Detection*, is employed to ascertain whether our file includes a header. If a header is present, we use it to set the column names; otherwise, we generate them automatically. In our example, there is a header, and each column gets its name defined in there.

Now that our columns have names, we move on to the fourth, optional phase: *Type Replacement*. DuckDB's CSV reader provides users with the option to specify column types by name. If these types are specified, we replace the detected types with the user's specifications.

Finally, we progress to our last phase, *Type Refinement*. In this phase, we analyze additional sections of the file to validate the accuracy of the types determined during the initial type detection phase. If necessary, we refine them. In our example, we can see that the `Vegetarian` column was initially categorized as `BOOL`. However, upon further examination, it was found to contain the string `N/A`, leading to an upgrade of the column type to VARCHAR to accommodate all possible values.

The automatic detection is only executed on a sequential sample of the CSV file. By default, the size of the sample is 20,480 tuples (i.e., 10 DuckDB execution chunks). This can be configured via the `sample_size` option, and can be set to `-1` in case the user wants to sniff the complete file. Since the same data is repeatedly read with various options, and users can scan the entire file, all CSV buffers generated during sniffing are cached and efficiently managed to ensure high performance.

Of course, running the CSV Sniffer on very large files will have a drastic impact on the overall performance (see our benchmark section below). In these cases, the sample size should be kept at a reasonable level.

In the next subsections, I will describe each phase in detail.

Dialect Detection

In the *Dialect Detection*, we identify the delimiter, quotes, escapes, and new line delimiters of a CSV file.

Our delimiter search space consists of the following delimiters: `,`, `|`, `;`, `\t`. If the file has a delimiter outside the search space, it must be provided by the user (e.g., `delim='?'`). Our quote search space is `"`, `'` and `\0`, where `\0` is a string terminator indicating no quote is present; again, users can provide custom characters outside the search space (e.g., `quote='?'`). The search space of escape values depends on the value of the quote option, but in summary, they are the same as quotes with the addition of `\`, and again, they can also be provided by the user (`escape='?'`). Finally, the last detected option is the new line delimiters; they can be `\r`, `\n`, `\r\n`, and a mix of everything (trust me, I've seen a real-world CSV file that used a mix).

By default, the dialect detection runs on 24 different combinations of dialect configurations. To determine the most promising configuration, we calculate the number of columns each CSV tuple would produce under each of these configurations. The one that results in the most columns with the most consistent rows will be chosen.

The calculation of consistent rows depends on additional user-defined options. For example, the `null_padding` option will pad missing columns with NULL values. Therefore, rows with missing columns will have the missing columns padded with `NULL`.

If `null_padding` is set to true, CSV files with inconsistent rows will still be considered, but a preference will be given to configurations that minimize the occurrence of padded rows. If `null_padding` is set to false, the dialect detector will skip inconsistent rows at the beginning of the CSV file. As an example, consider the following CSV file.

```
I like my csv files to have notes to make dialect detection harder
I also like commas like this one : ,
A,B,C
1,2,3
4,5,6
```

Here the sniffer would detect that with the delimiter set to `,` the first row has one column, the second has two, but the remaining rows have 3 columns. Hence, if `null_padding` is set to false, it would still select `,` as a delimiter candidate, by assuming the top rows are dirty notes. (Believe me, CSV notes are a thing!). Resulting in the following table:

```
A,B,C
1, 2, 3
4, 5, 6
```

If `null_padding` is set to true, all lines would be accepted, resulting in the following table:

```
'I like my csv files to have notes to make dialect detection harder', None, None
'I also like commas like this one : ', None, None
'A', 'B', 'C'
'1', '2', '3'
'4', '5', '6'
```

If the `ignore_errors` option is set, then the configuration that yields the most columns with the least inconsistent rows will be picked.

Type Detection

After deciding the dialect that will be used, we detect the types of each column. Our *Type Detection* considers the following types: `SQLNULL`, `BOOLEAN`, `BIGINT`, `DOUBLE`, `TIME`, `DATE`, `TIMESTAMP`, `VARCHAR`. These types are ordered in specificity, which means we first check if a column is a `SQLNULL`; if not, if it's a `BOOLEAN`, and so on, until it can only be a `VARCHAR`. DuckDB has more types than the ones used by default. Users can also define which types the sniffer should consider via the `auto_type_candidates` option.

At this phase, the type detection algorithm goes over the first chunk of data (i.e., 2048 tuples). This process starts on the second valid row (i.e., not a note) of the file. The first row is stored separately and not used for type detection. It will be later detected if the first row is a header or not. The type detection runs a per-column, per-value casting trial process to determine the column types. It starts off with a unique, per-column array with all types to be checked. It tries to cast the value of the column to that type; if it fails, it removes the type from the array, attempts to cast with the new type, and continues that process until the whole chunk is finished.

At this phase, we also determine the format of `DATE` and `TIMESTAMP` columns. The following formats are considered for `DATE` columns:

- `%m-%d-%Y`
- `%m-%d-%y`
- `%d-%m-Y`
- `%d-%m-%y`
- `%Y-%m-%d`
- `%y-%m-%d`

The following are considered for `TIMESTAMP` columns:

- `%Y-%m-%dT%H:%M:%S.%f`
- `%Y-%m-%d %H:%M:%S.%f`
- `%m-%d-%Y %I:%M:%S %p`
- `%m-%d-%y %I:%M:%S %p`
- `%d-%m-%Y %H:%M:%S`
- `%d-%m-%y %H:%M:%S`
- `%Y-%m-%d %H:%M:%S`
- `%y-%m-%d %H:%M:%S`

For columns that use formats outside this search space, they must be defined with the `dateformat` and `timestampformat` options.

As an example, let's consider the following CSV file.

```
Name, Age
,
Jack Black, 54
Kyle Gass, 63.2
```

The first row [`Name`, `Age`] will be stored separately for the header detection phase. The second row [`NULL`, `NULL`] will allow us to cast the first and second columns to `SQLNULL`. Therefore, their type candidate arrays will be the same: [`SQLNULL`, `BOOLEAN`, `BIGINT`, `DOUBLE`, `TIME`, `DATE`, `TIMESTAMP`, `VARCHAR`].

In the third row [`Jack Black`, `54`], things become more interesting. With 'Jack Black,' the type candidate array for column 0 will exclude all values with higher specificity, as 'Jack Black' can only be converted to a `VARCHAR`. The second column cannot be converted to either `SQLNULL` or `BOOLEAN`, but it will succeed as a `BIGINT`. Hence, the type candidate for the second column will be [`BIGINT`, `DOUBLE`, `TIME`, `DATE`, `TIMESTAMP`, `VARCHAR`].

In the fourth row, we have [`Kyle Gass`, `63.2`]. For the first column, there's no problem since it's also a valid `VARCHAR`. However, for the second column, a cast to `BIGINT` will fail, but a cast to `DOUBLE` will succeed. Hence, the new array of candidate types for the second column will be [`DOUBLE`, `TIME`, `DATE`, `TIMESTAMP`, `VARCHAR`].

Header Detection

The *Header Detection* phase simply obtains the first valid line of the CSV file and attempts to cast it to the candidate types in our columns. If there is a cast mismatch, we consider that row as the header; if not, we treat the first row as actual data and automatically generate a

header.

In our previous example, the first row was [Name, Age], and the column candidate type arrays were [VARCHAR] and [DOUBLE, TIME, DATE, TIMESTAMP, VARCHAR]. Name is a string and can be converted to VARCHAR. Age is also a string, and attempting to cast it to DOUBLE will fail. Since the casting fails, the auto-detection algorithm considers the first row as a header, resulting in the first column being named Name and the second as Age.

If a header is not detected, column names will be automatically generated with the pattern `column${x}`, where x represents the column's position (0-based index) in the CSV file.

Type Replacement

Now that the auto-detection algorithm has discovered the header names, if the user specifies column types, the types detected by the sniffer will be replaced with them in the *Type Replacement* phase. For example, we can replace the Age type with FLOAT by using:

```
SELECT *
FROM read_csv('greatest_band_in_the_world.csv', types = {'Age': 'FLOAT'});
```

This phase is optional and will only be triggered if there are manually defined types.

Type Refinement

The *Type Refinement* phase performs the same tasks as type detection; the only difference is the granularity of the data on which the casting operator works, which is adjusted for performance reasons. During type detection, we conduct cast checks on a per-column, per-value basis.

In this phase, we transition to a more efficient vectorized casting algorithm. The validation process remains the same as in type detection, with types from type candidate arrays being eliminated if a cast fails.

How Fast Is the Sniffing?

To analyze the impact of running DuckDB's automatic detection, we execute the sniffer on the NYC taxi dataset. The file consists of 19 columns, 10,906,858 tuples and is 1.72 GB in size.

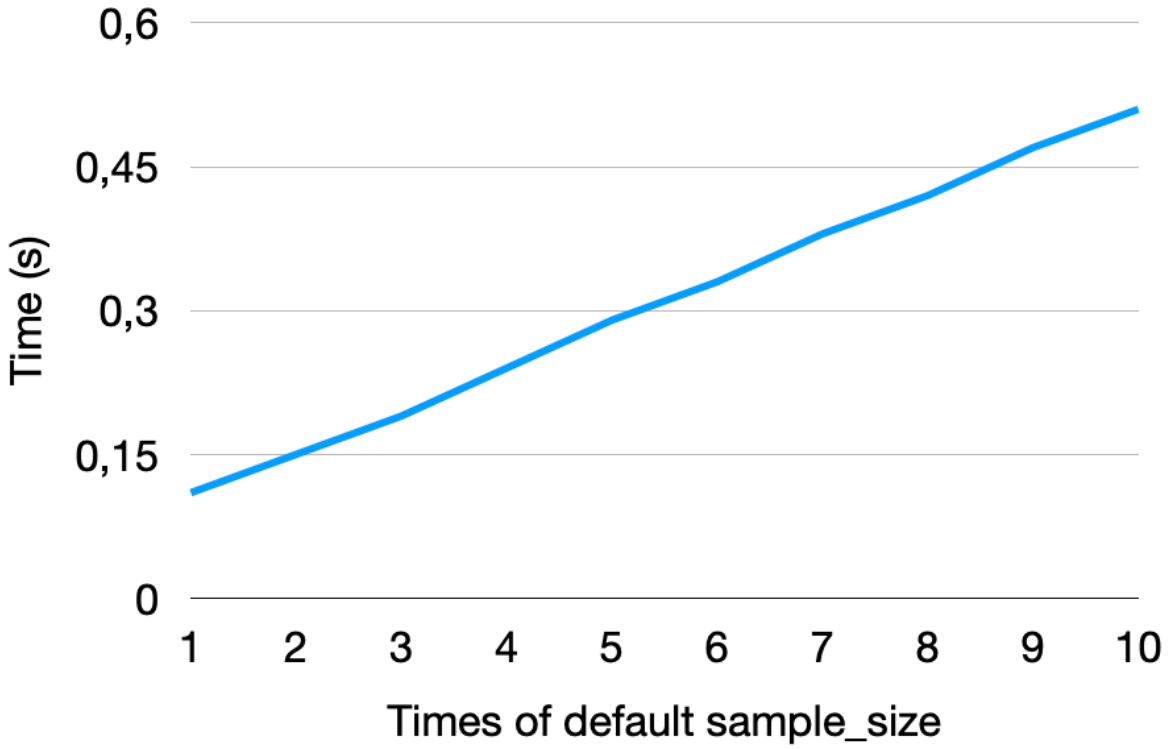
The cost of sniffing the dialect column names and types is approximately 4% of the total cost of loading the data.

Name	Time (s)
Sniffing	0.11
Loading	2.43

Varying Sampling Size

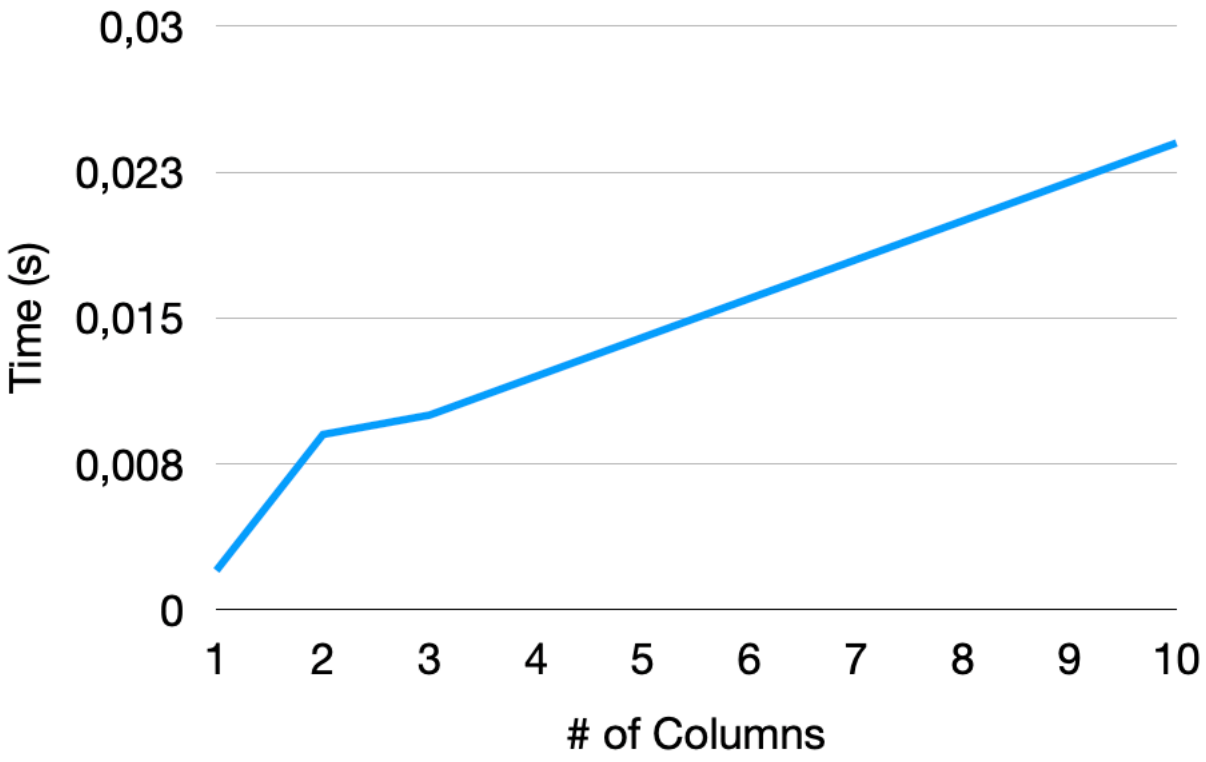
Sometimes, CSV files can have dialect options or more refined types that appear only later in the CSV file. In those cases, the `sample_size` option becomes an important tool for users to ensure that the sniffer examines enough data to make the correct decision. However, increasing the `sample_size` also leads to an increase in the total runtime of the sniffer because it uses more data to detect all possible dialects and types.

Below, you can see how increasing the default sample size by multiplier (see X axis) affects the sniffer's runtime on the NYC dataset. As expected, the total time spent on sniffing increases linearly with the total sample size.



Varying Number of Columns

The other main characteristic of a CSV file that will affect the auto-detection is the number of columns the file has. Here, we test the sniffer against a varying number of `INTEGER` type columns in files with 10,906,858 tuples. The results are depicted in the figure below. We can see that from one column to two, we have a steeper increase in runtime. That's because, for single columns, we have a simplified dialect detection due to the lack of delimiters. For the other columns, as expected, we have a more linear increase in runtime, depending on the number of columns.



Conclusion & Future Work

If you have unusual CSV files and want to query, clean up, or normalize them, DuckDB is already one of the top solutions available. It is very easy to get started. To read a CSV file with the sniffer, you can simply:

```
SELECT *
FROM 'path/to/csv_file.csv';
```

DuckDB's CSV auto-detection algorithm is an important tool to facilitate the exploration of CSV files. With its default options, it has a low impact on the total cost of loading and reading CSV files. Its main goal is to always be capable of reading files, doing a best-effort job even on files that are ill-defined.

We have a list of points related to the sniffer that we would like to improve in the future.

1. *Advanced Header Detection.* We currently determine if a CSV has a header by identifying a type mismatch between the first valid row and the remainder of the CSV file. However, this can generate false negatives if, for example, all the columns of a CSV are of a type VARCHAR. We plan on enhancing our Header Detection to perform matches with commonly used names for headers.
2. *Adding Accuracy and Speed Benchmarks.* We currently implement many accuracy and regression tests; however, due to the CSV's inherent flexibility, manually creating test cases is quite daunting. The plan moving forward is to implement a whole accuracy and regression test suite using the Pollock Benchmark
3. *Improved Sampling.* We currently execute the auto-detection algorithm on a sequential sample of data. However, it's very common that new settings are only introduced later in the file (e.g., quotes might be used only in the last 10% of the file). Hence, being able to execute the sniffer in distinct parts of the file can improve accuracy.
4. *Multi-Table CSV File.* Multiple tables can be present in the same CSV file, which is a common scenario when exporting spreadsheets to CSVs. Therefore, we would like to be able to identify and support these.
5. *Null-String Detection.* We currently do not have an algorithm in place to identify the representation of null strings.
6. *Decimal Precision Detection.* We also don't automatically detect decimal precision yet. This is something that we aim to tackle in the future.

7. *Parallelization.* Despite DuckDB's CSV Reader being fully parallelized, the sniffer is still limited to a single thread. Parallelizing it in a similar fashion to what is done with the CSV Reader (description coming in a future blog post) would significantly enhance sniffing performance and enable full-file sniffing.
8. *Sniffer as a stand-alone function.* Currently, users can utilize the `DESCRIBE` query to acquire information from the sniffer, but it only returns column names and types. We aim to expose the sniffing algorithm as a stand-alone function that provides the complete results from the sniffer. This will allow users to easily configure files using the exact same options without the need to rerun the sniffer.

Updates to the H2O.ai db-benchmark!

Publication date: 2023-11-03

Author: Tom Ebergen

TL;DR: The H2O.ai db-benchmark has been updated with new results. In addition, the AWS EC2 instance used for benchmarking has been changed to a c6id.metal for improved repeatability and fairness across libraries. DuckDB is the fastest library for both join and group by queries at almost every data size.

Skip directly to the results

The Benchmark Has Been Updated!

In April, DuckLabs published a blog post reporting updated H2O.ai db-benchmark results. Since then, the results haven't been updated. The original plan was to update the results with every DuckDB release. DuckDB 0.9.1 was recently released, and DuckLabs has updated the benchmark. While updating the benchmark, however, we noticed that our initial setup did not lend itself to being fair to all solutions. The machine used had network storage and could suffer from noisy neighbors. To avoid these issues, the whole benchmark was re-run on a c6id.metal machine.

New Benchmark Environment: c6id.metal Instance

Initially, updating the results to the benchmark showed strange results. Even using the same library versions from the prior update, some solutions regressed and others improved. We believe this variance came from the AWS EC2 instance type we chose: `m4.10xlarge`. The `m4.10xlarge` has 40 virtual CPUs and EBS storage. EBS storage is highly available network block storage for EC2 instances. When running compute-heavy benchmarks, a machine like the `m4.10xlarge` can suffer from the following issues:

- **Network storage** is an issue for benchmarking solutions that interact with storage frequently. For the 500 MB and 5 GB workloads, network storage was not an issue on the `m4.10xlarge` since all solutions could execute the queries in memory. For the 50 GB workload, however, network storage was an issue for the solutions that could not execute queries in memory. While the `m4.10xlarge` has dedicated EBS bandwidth, any read/write from storage is still happening over the network, which is usually slower than physically mounted storage. Solutions that frequently read and write to storage for the 50 GB queries end up doing this over the network. This network time becomes a chunk of the execution time of the query. If the network has variable performance, the query performance is then also variable.
- **Noisy neighbors** is a common issue when benchmarking on virtual CPUs. The previous machine most likely shared its compute hardware with other (neighboring) AWS EC2 instances. If these neighbors are also running compute heavy workloads, the physical CPU caches are repeatedly invalidated/flushed by the neighboring instance and the benchmark instance. When the CPU cache is shared between two workloads on two instances, both workloads require extra reads from memory for data that would already be in CPU cache on a non-virtual machine.

In order to be fair to all solutions, we decided to change the instance type to a metal instance with local storage. Metal instance types negate any noisy neighbor problems because the hardware is physical and not shared with any other AWS users/instances. Network storage problems are also fixed because solutions can read and write data to the local instance storage, which is physically mounted on the hardware.

Another benefit of the c6id.metal box is that it stresses parallel performance. There are 128 cores on the c6id.metal. Performance differences between solutions that can effectively use every core and solutions that cannot are clearly visible.

See the updated settings section on how settings were change for each solution when run on the new machine.

Updating the Benchmark

Moving forward we will update the benchmark when PRs with new performance numbers are provided. The PR should include a description of the changes to a solution script or a version update and new entries in the `time.csv` and `logs.csv` files. These entries will be verified using a different c6id.metal instance, and if there is limited variance, the PR will be merged and the results will be updated!

Updated Settings

1. ClickHouse
    - Storage: Any data this gets spilled to disk also needs to be on the NVMe drive. This has been changed in the new `format_and_mount.sh` script and the `clickhouse/clickhouse-mount-config.xml` file.
2. Julia (juliadf & juliads)
    - Threads: The threads were hardcoded for juliadf/juliads to 20/40 threads. Now the max number of threads are used. No option was given to spill to disk, so this was not changed/researched.
3. DuckDB
    - Storage: The DuckDB database file was specified to run on the NVMe mount.
4. Spark
    - Storage: There is an option to spill to disk. I was unsure of how to modify the storage location so that it was on the NVMe drive. Open to a PR with storage location changes and improved results!

Many solutions do not spill to disk, so they did not require any modification to use the instance storage. Other solutions use `parallel::ncores()` or default to a maximum number of cores for parallelism. Solution scripts were run in their current form on github.com/duckdblabs/db-benchmark. Please read the Updating the Benchmark section on how to re-run your solution.

Results

The first results you see are the 50 GB group by results. The benchmark runs every query twice per solution, and both runtimes are reported. The “first time” can be considered a cold run, and the “second time” can be considered a hot run. DuckDB and DuckDB-latest perform very well among all dataset sizes and variations.

The team at DuckLabs has been hard at work improving the performance of the out-of-core hash aggregates and joins. The most notable improvement is the performance of query 5 in the advanced group by queries. The cold run is almost an order of magnitude better than every other solution! DuckDB is also one of only two solutions to finish the 50 GB join query. Some solutions are experiencing timeouts on the 50 GB datasets. Solutions running the 50 GB group by queries are killed after running for 180 minutes, meaning all 10 group by queries need to finish within the 180 minutes. Solutions running the 50 GB join queries are killed after running for 360 minutes.

Link to result page

Extensions for DuckDB-Wasm

Publication date: 2023-12-18

Author: Carlo Piovesan

TL;DR: DuckDB-Wasm users can now load DuckDB extensions, allowing them to run extensions in the browser.

In this blog post, we will go over two exciting DuckDB features: the DuckDB-Wasm client and DuckDB extensions. I will discuss how these disjoint features have now been adapted to work together. These features are now available for DuckDB-Wasm users and you can try them out at shell.duckdb.org.

DuckDB Extensions

DuckDB's philosophy is to have a lean core system to ensure robustness and portability. However, a competing design goal is to be flexible and allow a wide range of functionality that is necessary to perform advanced analytics. To accommodate this, DuckDB has an extension mechanism for installing and loading extensions during runtime.

Running DuckDB Extensions Locally

For DuckDB, here is a simple end-to-end example using the command line interface:

```
INSTALL tpch;
LOAD tpch;
CALL dbgen(sf = 0.1);
PRAGMA tpch(7);
```

This script first installs the TPC-H extension from the official extension repository, which implements the popular TPC-H benchmark. It then loads the TPC-H extension, uses it to populate the database with generated data using the `dbgen` function. Finally, it runs TPC-H query 7.

This example demonstrates a case where we install an extension to complement DuckDB with a new feature (the TPC-H data generator), which is not part of the base DuckDB executable. Instead, it is downloaded from the extension repository, then loaded and executed it locally within the framework of DuckDB.

Currently, DuckDB has several extensions. These add support for filesystems, file formats, database and network protocols. Additionally, they implement new functions such as full text search.

DuckDB-Wasm

In an effort spearheaded by André Kohn, DuckDB was ported to the WebAssembly platform in 2021. WebAssembly, also known as Wasm, is a W3C standard language developed in recent years. Think of it as a machine-independent binary format that you can execute from within the sandbox of a web browser.

Thanks to DuckDB-Wasm, anyone has access to a DuckDB instance only a browser tab away, with all computation being executed locally within your browser and no data leaving your device. DuckDB-Wasm is a library that can be used in various deployments (e.g., notebooks that run inside your browser without a server). In this post, we will use the Web shell, where SQL statements are entered by the user line by line, with the behavior modeled after the DuckDB CLI shell.

DuckDB Extensions, in DuckDB-Wasm!

DuckDB-Wasm now supports DuckDB extensions. This support comes with four new key features. First, the DuckDB-Wasm library can be compiled with dynamic extension support. Second, DuckDB extensions can be compiled to a single WebAssembly module. Third, users and developers working with DuckDB-Wasm can now select the set of extensions they load. Finally, the DuckDB-Wasm shell's features are now much closer to the native CLI functionality.

Using the TPC-H Extension in DuckDB-Wasm

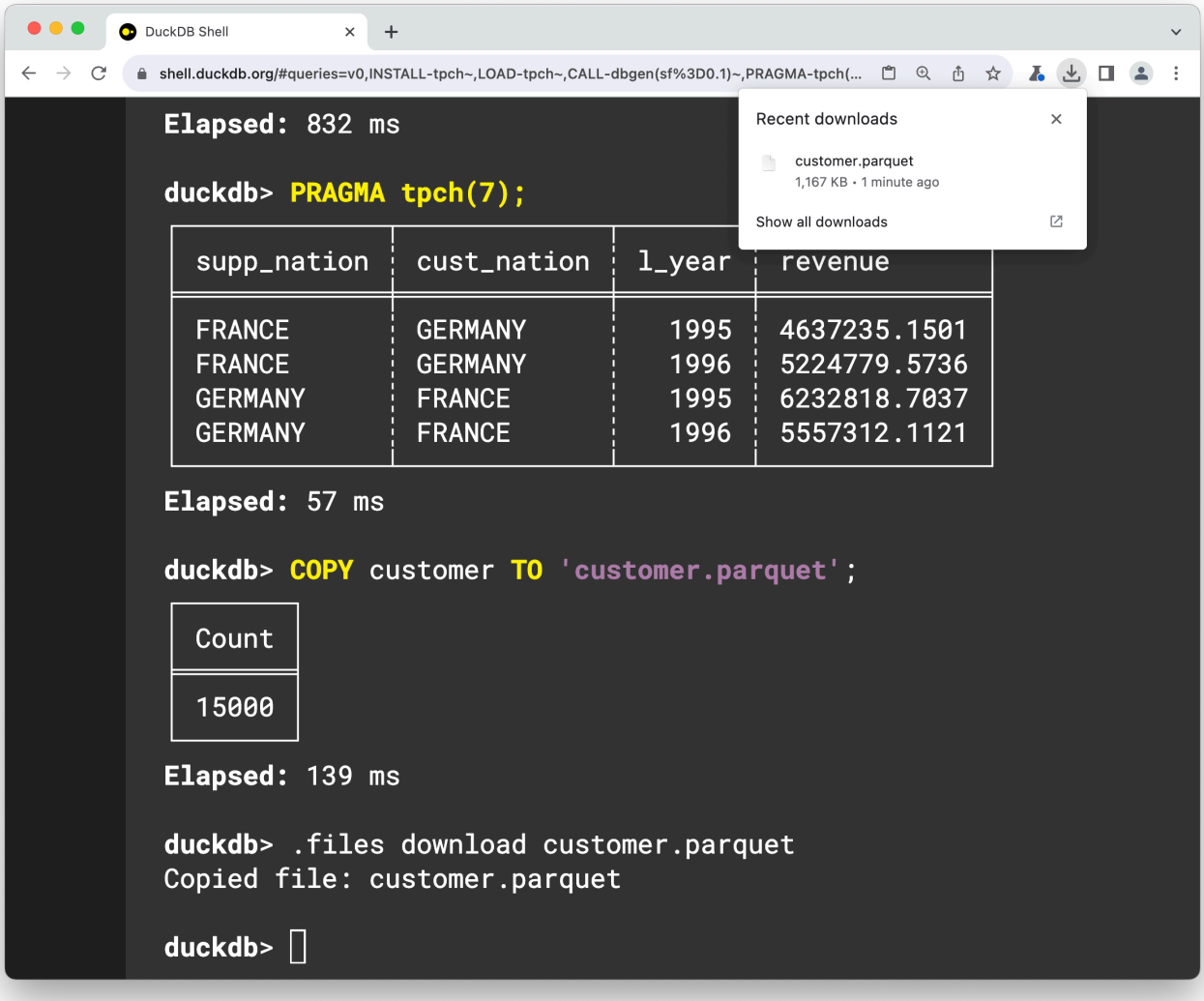
To demonstrate this, we will again use the TPC-H data generation example. To run this script in your browser, start an online DuckDB shell that runs these commands. The script will generate the TPC-H data set at scale factor 0.1, which corresponds to 100 MB in uncompressed CSV format.

Once the script is finished, you can keep executing queries, or you could even download the `customer.parquet` file (1 MB) using the following commands:

```
COPY customer TO 'customer.parquet';
.files download customer.parquet
```

This will first copy the `customer.parquet` to the DuckDB-Wasm file system, then download it via your browser.

In short, your DuckDB instance, which *runs entirely within your browser,* first installed and loaded the TPC-H extension. It then used the extension logic to generate data and convert it to a Parquet file. Finally, you could download the Parquet file as a regular file to your local file system.



Using the Spatial Extension in DuckDB-Wasm

To show the possibilities unlocked by DuckDB-Wasm extensions and test the capabilities of what's possible, what about using the spatial extension within DuckDB-Wasm? This extension implements geospatial types and functions that allow it to work with geospatial data and relevant workloads.

To install and load the spatial extension in DuckDB-Wasm, run:

```
INSTALL spatial;
LOAD spatial;
```

Using the spatial extension, the following query uses the New York taxi dataset, and calculates the area of the taxi zones for each borough:

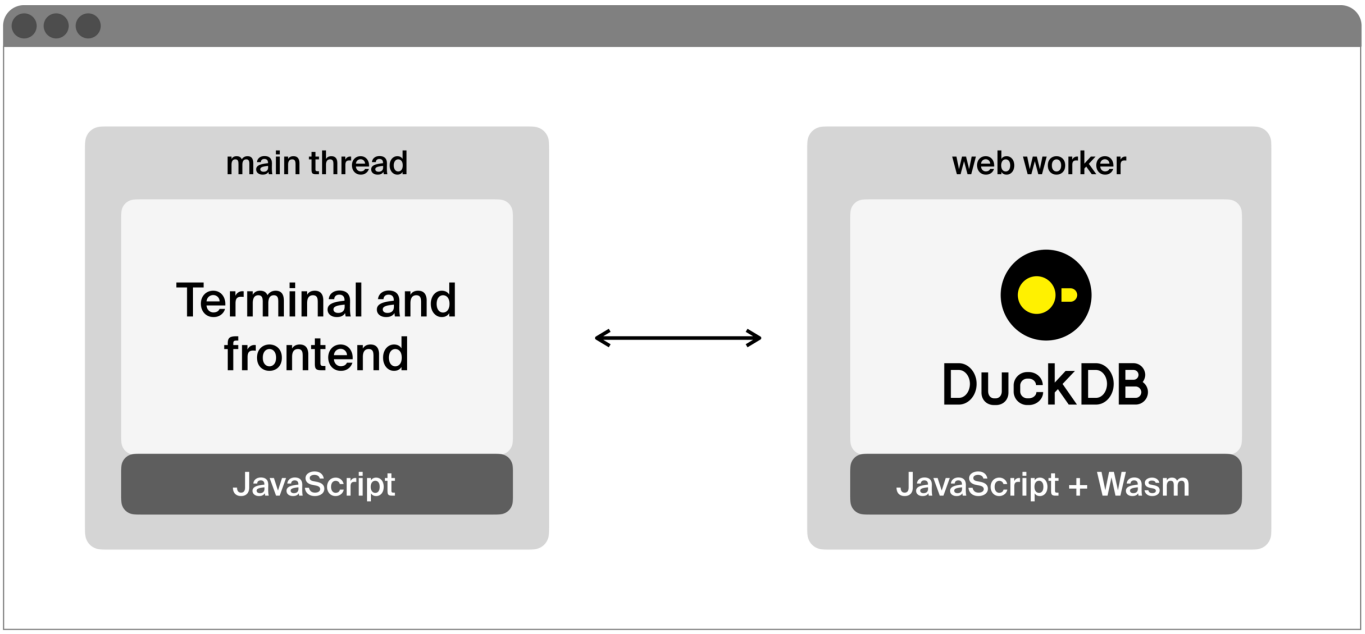
```
CREATE TABLE nyc AS
    SELECT
        borough,
        st_union_agg(geom) AS full_geom,
        st_area(full_geom) AS area,
        st_centroid(full_geom) AS centroid,
        count(*) AS count
    FROM
        st_read('https://raw.githubusercontent.com/duckdb/duckdb-spatial/main/test/data/nyc_taxi/taxi_
        zones/taxi_zones.shp')
```

```
GROUP BY borough;

SELECT borough, area, centroid::VARCHAR, count
FROM nyc;
```

Both your local DuckDB client and the online DuckDB shell will perform the same analysis.

Under the Hood

Let's dig into how this all works. The following figure shows an overview of DuckDB-Wasm's architecture. Both components in the figure run within the web browser.

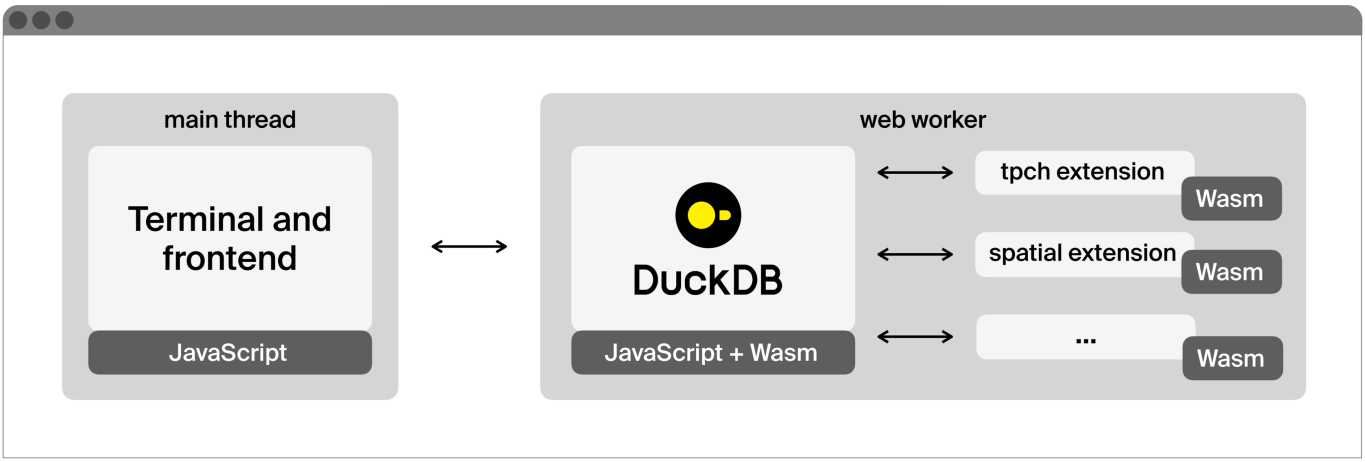


When you load DuckDB-Wasm in your browser, there are two components that will be set up:

(1) A main-thread wrapper library that acts as a bridge between users or code using DuckDB-Wasm and drives the background component.

(2) A DuckDB engine used to execute queries. This component lives in a Web Worker and communicates with the main thread component via messages. This component has a JavaScript layer that handles messages and the original DuckDB C++ logic compiled down to a single WebAssembly file.

What happens when we add extensions to the mix?



Extensions for DuckDB-Wasm are composed of a single WebAssembly module. This will encode the logic and data of the extensions, the list of functions that are going to be imported and exported, and a custom section encoding metadata that allows verification of the extension.

To make extension loading work, the DuckDB engine component blocks, fetches, and validates external WebAssembly code, then links it in, wires together import and export, and then the system will be connected and set to keep executing as if it was a single codebase.

The central code block that makes this possible is the following:

```
EM_ASM(
    {
        const xhr = new XMLHttpRequest();
        xhr.open("GET", UTF8ToString($0), false);
        xhr.responseType = "arraybuffer";
        xhr.send(null);
        var uInt8Array = xhr.response;
        // Check signatures / version compatibility left as an exercise
        WebAssembly.validate(uInt8Array);
        // Here we add the uInt8Array to Emscripten's filesystem,
        // for it to be found by dlopen
        FS.writeFile(UTF8ToString($1), new Uint8Array(uInt8Array));
    },
    filename.c_str(), basename.c_str()
);

auto lib_hdl = dlopen(basename.c_str(), RTLD_NOW | RTLD_LOCAL);
if (!lib_hdl) {
    throw IOException(
      "Extension \"%s\" could not be loaded: %s",
      filename,
      GetDLError()
    );
}
```

Here, we rely on two powerful features of Emscripten, the compiler toolchain we are using to compile DuckDB to WebAssembly.

First, `EM_ASM` allows us to inline JavaScript code directly in C++ code. It means that during runtime when we get to that block of code, the WebAssembly component will go back to JavaScript land, perform a blocking `XMLHttpRequest` on a URL such as https://extensions.duckdb.org/.../tpch.duckdb_extension.wasm, then validate that the package that has been just fetched is actually a valid WebAssembly module.

Second, we leverage Emscripten's `dlopen` implementation, which enables compatible WebAssembly modules to be linked together and act as a single composable codebase.

These enable implementing dynamic loading of extensions, when triggered via the SQL `LOAD` statement.

Developer Guide

We see two main groups of developers using extensions with DuckDB-Wasm.

- Developers working with DuckDB-Wasm: If you are building a website or a library that wraps DuckDB-Wasm, the new extension support means that there is now a wider range of functionality that can be exposed to your users.
- Developers working on DuckDB extensions: If you have written a DuckDB extension, or are thinking of doing so, consider porting it to DuckDB-Wasm. The DuckDB extension template repository contains the configuration required for compiling to DuckDB-Wasm.

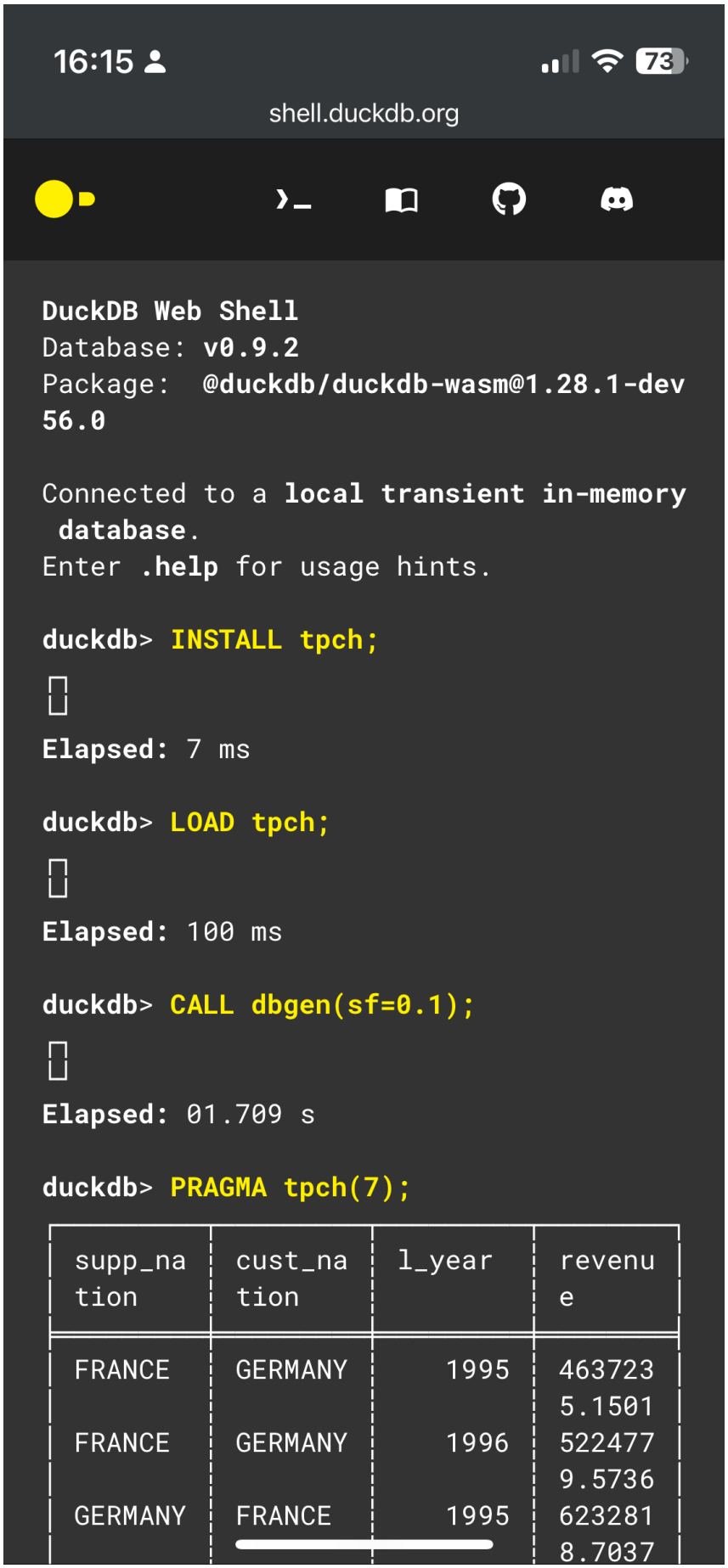
Limitations

DuckDB-Wasm extensions have a few inherent limitations. For example, it is not possible to communicate with native executables living on your machine, which is required by some extensions, such as the `postgres` scanner extension. Moreover, compilation to Wasm may not be currently supported for some libraries you are relying on, or capabilities might not be one-to-one with local executables due to additional requirements imposed on the browser, in particular around non-secure HTTP requests.

Conclusions

In this blog post, we explained how DuckDB-Wasm supports extensions, and demonstrated with multiple extensions: TPC-H, Parquet, and spatial.

Thanks to the portability of DuckDB, the scripts shown in this blog post also work on your smartphone:


16:15
73
shell.duckdb.org
DuckDB Web Shell
Database: v0.9.2
Package: @duckdb/duckdb-wasm@1.28.1-dev56.0
Connected to a local transient in-memory database.
Enter .help for usage hints.
duckdb> INSTALL tpch;
Elapsed: 7 ms
duckdb> LOAD tpch;
Elapsed: 100 ms
duckdb> CALL dbgen(sf=0.1);
Elapsed: 01.709 s
duckdb> PRAGMA tpch(7);
supp_nation
cust_nation
l_year
revenue
FRANCE
GERMANY
1995
4637235.1501
FRANCE
GERMANY
1996
5224779.5736
GERMANY
FRANCE
1995
6232818.7037

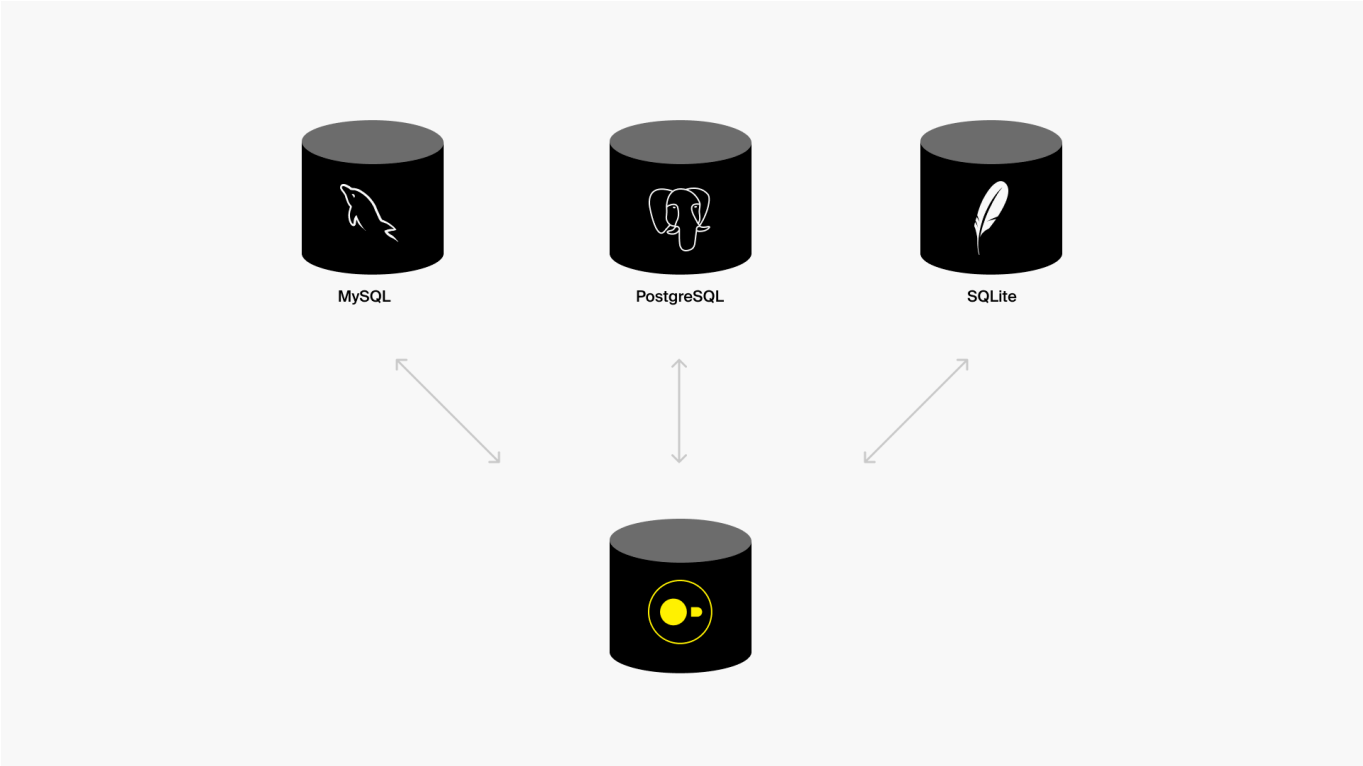
For updates on the latest developments, follow this blog and join the Wasm channel in our Discord. If you have an example of what's possible with extensions in DuckDB, let us know!

Multi-Database Support in DuckDB

Publication date: 2024-01-26

Author: Mark Raasveldt

TL;DR: DuckDB can attach MySQL, Postgres, and SQLite databases in addition to databases stored in its own format. This allows data to be read into DuckDB and moved between these systems in a convenient manner.



In modern data analysis, data must often be combined from a wide variety of different sources. Data might sit in CSV files on your machine, in Parquet files in a data lake, or in an operational database. DuckDB has strong support for moving data between many different data sources. However, this support has previously been limited to reading data and writing data to files.

DuckDB supports advanced operations on its own native storage format – such as deleting rows, updating values, or altering the schema of a table. It supports all of these operations using ACID semantics. This guarantees that your database is always left in a sane state – operations are atomic and do not partially complete.

DuckDB now has a pluggable storage and transactional layer. This flexible layer allows new storage back-ends to be created by DuckDB extensions. These storage back-ends can support all database operations in the same way that DuckDB supports them, including inserting data and even modifying schemas.

The MySQL, Postgres, and SQLite extensions implement this new pluggable storage and transactional layer, allowing DuckDB to connect to those systems and operate on them in the same way that it operates on its own native storage engine.

These extensions enable a number of useful features. For example, using these extensions you can:

- Export data from SQLite to JSON
- Read data from Parquet into Postgres
- Move data from MySQL to Postgres

... and much more.

Attaching Databases

The ATTACH statement can be used to attach a new database to the system. By default, a native DuckDB file will be attached. The TYPE parameter can be used to specify a different storage type. Alternatively, the `{type}:` prefix can be used.

For example, using the SQLite extension, we can open a SQLite database file and query it as we would query a DuckDB database.

```
ATTACH 'sakila.db' AS sakila (TYPE sqlite);
SELECT title, release_year, length FROM sakila.film LIMIT 5;
```

title varchar	release_year varchar	length int64
ACADEMY DINOSAUR	2006	86
ACE GOLDFINGER	2006	48
ADAPTATION HOLES	2006	50
AFFAIR PREJUDICE	2006	117
AFRICAN EGG	2006	130

The USE command switches the main database.

```
USE sakila;
SELECT first_name, last_name FROM actor LIMIT 5;
```

first_name varchar	last_name varchar
PENELOPE	GUINESS
NICK	WAHLBERG
ED	CHASE
JENNIFER	DAVIS
JOHNNY	LOLLOBRIGIDA

The SQLite database can be manipulated as if it were a native DuckDB database. For example, we can create a new table, populate it with values from a Parquet file, delete a few rows from the table and alter the schema of the table.

```
CREATE TABLE lineitem AS FROM 'lineitem.parquet' LIMIT 1000;
DELETE FROM lineitem WHERE l_returnflag = 'N';
ALTER TABLE lineitem DROP COLUMN l_comment;
```

The `duckdb_databases` table contains a list of all attached databases and their types.

```
SELECT database_name, path, type FROM duckdb_databases;
```

database_name varchar	path varchar	type varchar
sakila	sakila.db	sqlite
memory	NULL	duckdb

Mix and Match

While attaching to different database types is useful – it becomes even more powerful when used in combination. For example, we can attach both a SQLite, MySQL and a Postgres database.

```
ATTACH 'sqlite:sakila.db' AS sqlite;
ATTACH 'postgres:dbname=postgresscanner' AS postgres;
ATTACH 'mysql:user=root database=mysqlscanner' AS mysql;
```

Now we can move data between these attached databases and query them together. Let's copy the `film` table to MySQL, and the `actor` table to Postgres:

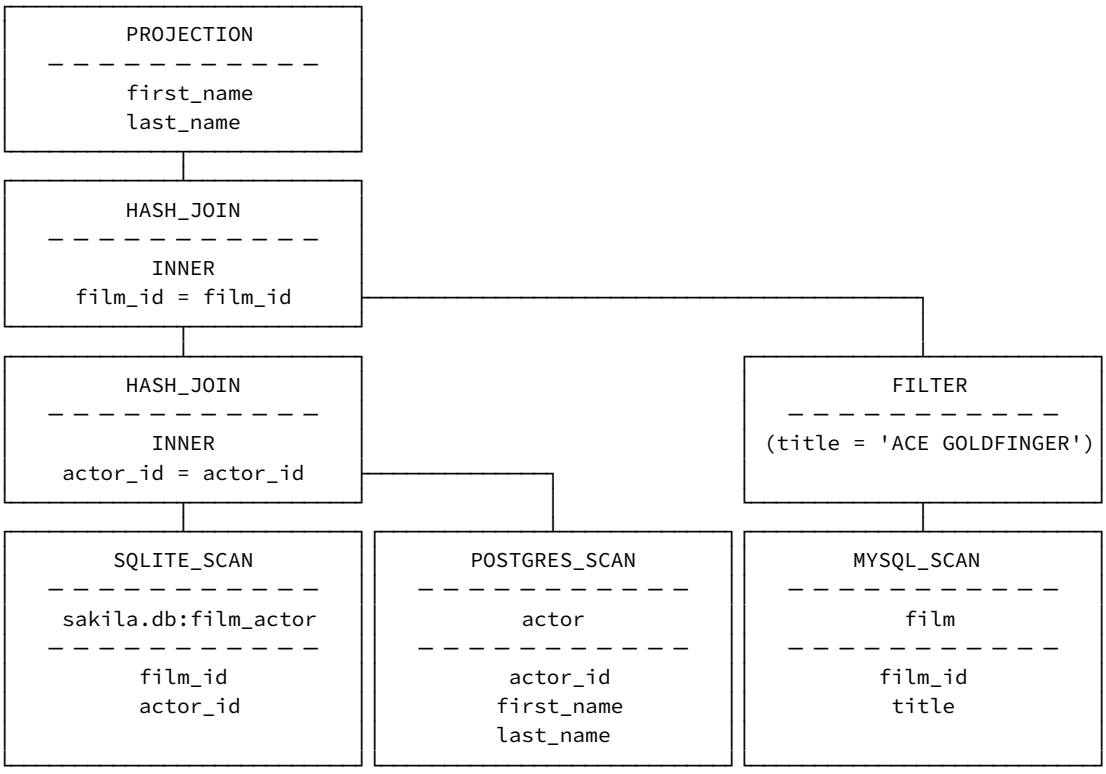
```
CREATE TABLE mysql.film AS FROM sqlite.film;
CREATE TABLE postgres.actor AS FROM sqlite.actor;
```

We can now join tables from these three attached databases together. Let's find all of the actors that starred in `Ace Goldfinger`.

```
SELECT first_name, last_name
FROM mysql.film
JOIN sqlite.film_actor ON (film.film_id = film_actor.film_id)
JOIN postgres.actor ON (actor.actor_id = film_actor.actor_id)
WHERE title = 'ACE GOLDFINGER';
```

first_name varchar	last_name varchar
BOB	FAWCETT
MINNIE	ZELLWEGER
SEAN	GUINESS
CHRIS	DEPP

Running `EXPLAIN` on the query shows how the data from the different engines is combined into the final query result.



Transactions

All statements executed within DuckDB are executed within a transaction. If an explicit `BEGIN TRANSACTION` is not called, every statement will execute in its own transaction. This also applies to queries that are executed over other storage engines. These storage engines also support explicit `BEGIN`, `COMMIT` and `ROLLBACK` statements.

For example, we can begin a transaction within our attached `SQLite` database, make a change, and then roll it back. The original data will be restored.

```
BEGIN;
TRUNCATE film;
SELECT title, release_year, length FROM film;
```

title	release_year	length
varchar	varchar	int64
0 rows		

```
ROLLBACK;
SELECT title, release_year, length FROM film LIMIT 5;
```

title	release_year	length
varchar	varchar	int64
ACADEMY DINOSAUR	2006	86
ACE GOLDFINGER	2006	48
ADAPTATION HOLES	2006	50
AFFAIR PREJUDICE	2006	117
AFRICAN EGG	2006	130

Multi-Database Transactions

Every storage engine has their own transactions that are stand-alone and managed by the storage engine itself. Opening a transaction in Postgres, for example, calls `BEGIN TRANSACTION` in the Postgres client. The transaction is managed by Postgres itself. Similarly, when the transaction is committed or rolled back, the storage engine handles this by itself.

Transactions are used both for **reading** and for **writing** data. For reading data, they are used to provide a consistent snapshot of the database. For writing, they are used to ensure all data in a transaction is packed together and written at the same time.

When executing a transaction that involves multiple attached databases we need to open multiple transactions: one per attached database that is used in the transaction. While this is not a problem when **reading** from the database, it becomes complicated when **writing**. In particular, when we want to `COMMIT` a transaction it is challenging to ensure that either (a) every database has successfully committed, or (b) every database has rolled back.

For that reason, it is currently not supported to **write** to multiple attached databases in a single transaction. Instead, an error is thrown when this is attempted:

```
BEGIN;
CREATE TABLE postgres.new_table (i INTEGER);
CREATE TABLE mysql.new_table (i INTEGER);
```

```
Error: Attempting to write to database "mysql" in a transaction that has
already modified database "postgres" - a single transaction can only write
to a single attached database.
```

Copying Data between Databases

`CREATE TABLE AS`, `INSERT INTO` and `COPY` can be used to copy data between different attached databases. The dedicated `COPY FROM DATABASE ... TO` can be used to copy all data from one database to another. This includes all tables and views that are stored in the source database.

```
-- attach a Postgres database
ATTACH 'postgres:dbname=postgresscanner' AS postgres;
```

```
-- attach a DuckDB file
ATTACH 'database.db' AS ddb;
-- export all tables and views from the Postgres database to the DuckDB file
COPY FROM DATABASE postgres TO ddb;
```

Directly Opening a Database

The explicit ATTACH statement is not required to connect to a different database type. When instantiating a DuckDB instance a connection can be made directly to a different database type using the `{type}:` prefix. For example, to connect to a SQLite file, use `sqlite:file.db`. To connect to a Postgres instance, use `postgres:dbname=postgresscanner`. This can be done in any client, including the CLI. For instance:

CLI:

```
duckdb sqlite:file.db
```

Python:

```
import duckdb
con = duckdb.connect('sqlite:file.db')
```

This is equivalent to attaching the storage engine and running USE afterwards.

Conclusion

DuckDB's pluggable storage engine architecture enables many use cases. By attaching multiple databases, data can be extracted in a transactionally safe manner for bulk ETL or ELT workloads, as well as for on-the-fly data virtualization workloads. These techniques also work well in combination, for example, by moving data in bulk on a regular cadence, while filling in the last few data points on the fly.

Pluggable storage engines also unlock new ways to handle concurrent writers in a data platform. Each separate process could write its output to a transactional database, and the results could be combined within DuckDB – all in a transactionally safe manner. Then, data analysis tasks can occur on the centralized DuckDB database for improved performance.

We look forward to hearing the many creative ways you are able to use this feature!

Future Work

We intend to continue enhancing the performance and capabilities of the existing extensions. In addition, all of these features can be leveraged by the community to connect to other databases.

Announcing DuckDB 0.10.0

Publication date: 2024-02-13

Authors: Mark Raasveldt, Hannes Mühleisen

TL;DR: The DuckDB team is happy to announce the latest DuckDB release (0.10.0). This release is named Fusca after the Velvet scoter native to Europe.

To install the new version, please visit the installation guide. The full release notes can be found on GitHub.

What's New in 0.10.0

There have been too many changes to discuss them each in detail, but we would like to highlight several particularly exciting features!

Below is a summary of those new features with examples, starting with a change in our SQL dialect that is designed to produce more intuitive results by default.

Breaking SQL Changes

Implicit Cast to VARCHAR. Previously, DuckDB would automatically allow any type to be implicitly cast to VARCHAR during function bind-

ing. As a result it was possible to e.g., compute the substring of an integer without using an implicit cast. Starting with this release, you will need to use an explicit cast here instead.

```
SELECT substring(42, 1, 1) AS substr;
```

```
No function matches the given name and argument types 'substring(...)'.
You might need to add explicit type casts.
```

To use an explicit cast, run:

```
SELECT substring(42::VARCHAR, 1, 1) AS substr;
```

substr
varchar
4

Alternatively, the `old_implicit_casting` setting can be used to revert this behavior, e.g.:

```
SET old_implicit_casting = true;
SELECT substring(42, 1, 1) AS substr;
```

substr
varchar
4

Literal Typing. Previously, integer and string literals behaved identically to the `INTEGER` and `VARCHAR` types. Starting with this release, `INTEGER_LITERAL` and `STRING_LITERAL` are separate types that have their own binding rules.

- `INTEGER_LITERAL` types can be implicitly converted to any integer type in which the value fits
- `STRING_LITERAL` types can be implicitly converted to **any** other type

This aligns DuckDB with Postgres, and makes operations on literals more intuitive. For example, we can compare string literals with dates – but we cannot compare VARCHAR values with dates.

```
SELECT d > '1992-01-01' AS result
FROM (VALUES (DATE '1992-01-01')) t(d);
```

result
boolean
false

```
SELECT d > '1992-01-01'::VARCHAR AS result
FROM (VALUES (DATE '1992-01-01')) t(d);
```

```
Binder Error:
Cannot compare values of type DATE and type VARCHAR - an explicit cast is required
```

Backward Compatibility

Backward compatibility refers to the ability of a newer DuckDB version to read storage files created by an older DuckDB version. This release is the first release of DuckDB that supports backward compatibility in the storage format. DuckDB v0.10 can read and operate on files created by the previous DuckDB version – DuckDB v0.9. This is made possible by the implementation of a new serialization framework.

Write with v0.9:

```
duckdb_092 v092.db
```

```
CREATE TABLE lineitem AS
FROM lineitem.parquet;
```

Read with v0.10:

```
duckdb_0100 v092.db
```

```
SELECT l_orderkey, l_partkey, l_comment
FROM lineitem
LIMIT 1;
```

l_orderkey int32	l_partkey int32	l_comment varchar
1	155190	to beans x-ray carefull

For future DuckDB versions, our goal is to ensure that any DuckDB version released **after** can read files created by previous versions, starting from this release. We want to ensure that the file format is fully backward compatible. This allows you to keep data stored in DuckDB files around and guarantees that you will be able to read the files without having to worry about which version the file was written with or having to convert files between versions.

Forward Compatibility

Forward compatibility refers to the ability of an older DuckDB version to read storage files produced by a newer DuckDB version. DuckDB v0.9 is **partially** forward compatible with DuckDB v0.10. Certain files created by DuckDB v0.10 can be read by DuckDB v0.9.

Write with v0.10:

```
duckdb_0100 v010.db
```

```
CREATE TABLE lineitem AS
FROM lineitem.parquet;
```

Read with v0.9:

```
duckdb_092 v010.db
```

```
SELECT l_orderkey, l_partkey, l_comment
FROM lineitem
LIMIT 1;
```

l_orderkey int32	l_partkey int32	l_comment varchar
1	155190	to beans x-ray carefull

Forward compatibility is provided on a **best effort** basis. While stability of the storage format is important – there are still many improvements and innovations that we want to make to the storage format in the future. As such, forward compatibility may be (partially) broken on occasion.

For this release, DuckDB v0.9 is able to read files created by DuckDB v0.10 provided that:

- The database file does not contain views
- The database file does not contain new types (`ARRAY`, `UHUGEINT`)
- The database file does not contain indexes (`PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, explicit indexes)
- The database file does not contain new compression methods (`ALP`). As ALP is automatically used to compress `FLOAT` and `DOUBLE` columns – that means forward compatibility in practice often does not work for `FLOAT` and `DOUBLE` columns unless `ALP` is explicitly disabled through configuration.

We expect that as the format stabilizes and matures this will happen less frequently – and we hope to offer better guarantees in allowing DuckDB to read files written by future DuckDB versions.

CSV Reader Rework

CSV Reader Rework. The CSV reader has received a major overhaul in this release. The new CSV reader uses efficient state machine transitions to speed through CSV files. This has greatly sped up performance of the CSV reader, particularly in multi-threaded scenarios. In addition, in the case of malformed CSV files, reported error messages should be more clear.

Below is a benchmark comparing the loading time of 11 million rows of the NYC Taxi dataset from a CSV file on an M1 Max with 10 cores:

Version	Load time
v0.9.2	2.6 s
v0.10.0	1.2 s

Furthermore, many optimizations have been done that make running queries over CSV files directly significantly faster as well. Below is a benchmark comparing the execution time of a `SELECT count(*)` query directly over the NYC Taxi CSV file.

Version	Query time
v0.9.2	1.8 s
v0.10.0	0.3 s

Fixed-Length Arrays

Fixed-Length Arrays. This release introduces the fixed-length array type. Fixed-length arrays are similar to lists, however, every value must have the same fixed number of elements in them.

```
CREATE TABLE vectors (v DOUBLE[3]);
INSERT INTO vectors VALUES ([1, 2, 3]);
```

Fixed-length arrays can be operated on faster than variable-length lists as the size of each list element is known ahead of time. This release also introduces specialized functions that operate over these arrays, such as `array_cross_product`, `array_cosine_similarity`, and `array_inner_product`.

```
SELECT array_cross_product(v, [1, 1, 1]) AS result
FROM vectors;
```

```
┌──────────────────┐
│      result      │
│     double[3]    │
├──────────────────┤
│ [-1.0, 2.0, -1.0] │
└──────────────────┘
```

See the Array Type page in the documentation for more information.

Multi-Database Support

DuckDB can now attach MySQL, Postgres, and SQLite databases in addition to databases stored in its own format. This allows data to be read into DuckDB and moved between these systems in a convenient manner, as attached databases are fully functional, appear just as regular tables, and can be updated in a safe, transactional manner. More information about multi-database support can be found in our recent blog post.

```
ATTACH 'sqlite:sakila.db' AS sqlite;
ATTACH 'postgres:dbname=postgresscanner' AS postgres;
ATTACH 'mysql:user=root database=mysqlscanner' AS mysql;
```

Secret Manager

DuckDB integrates with several cloud storage systems such as S3 that require access credentials to access data. In the current version of DuckDB, authentication information is configured through DuckDB settings, e.g., `SET s3_access_key_id = '...';`{:.language-sql .highlight}. While this worked, it had several shortcomings. For example, it was not possible to set different credentials for different S3 buckets without modifying the settings between queries. Because settings are not considered secret, it was also possible to query them using `duckdb_settings()`.

With this release, DuckDB adds a new "Secrets Manager" to manage secrets in a better way. We now have a unified user interface for secrets across all backends that use them. Secrets can be scoped, so different storage prefixes can have different secrets, allowing, for example, joining across organizations in a single query. Secrets can also be persisted, so that they do not need to be specified every time DuckDB is launched.

Secrets are typed, their type identifies which service they are for. For example, this release can manage secrets for S3, Google Cloud Storage, Cloudflare R2 and Azure Blob Storage. For each type, there are one or more "secret providers" that specify how the secret is created. Secrets can also have an optional scope, which is a file path prefix that the secret applies to. When fetching a secret for a path, the secret scopes are compared to the path, returning the matching secret for the path. In the case of multiple matching secrets, the longest prefix is chosen.

Finally, secrets can be temporary or persistent. Temporary secrets are used by default – and are stored in-memory for the life span of the DuckDB instance similar to how settings worked previously. Persistent secrets are stored in unencrypted binary format in the `~/.duckdb/stored_secrets` directory. On startup of DuckDB, persistent secrets are read from this directory and automatically loaded.

For example, to create a temporary unscoped secret to access S3, we can now use the following syntax:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

If two secrets exist for a service type, the scope can be used to decide which one should be used. For example:

```
CREATE SECRET secret1 (
    TYPE s3,
    KEY_ID 'my_key1',
    SECRET 'my_secret1',
    SCOPE 's3://<my-bucket>'
);

CREATE SECRET secret2 (
    TYPE s3,
    KEY_ID 'my_key2',
    SECRET 'my_secret2',
    SCOPE 's3://<my-other-bucket>'
);
```

Now, if the user queries something from `s3://<my-other-bucket>/something`{:.language-sql .highlight}, `secret2` will be chosen automatically for that request.

Secrets can be listed using the built-in table-producing function, e.g., by using `FROM duckdb_secrets();`{:.language-sql .highlight}. Sensitive information will be redacted.

In order to persist secrets between DuckDB database instances, we can now use the `CREATE PERSISTENT SECRET` command, e.g.:

```
CREATE PERSISTENT SECRET my_persistent_secret (
    TYPE s3,
    KEY_ID 'my_key',
    SECRET 'my_secret'
);
```

As mentioned, this will write the secret (unencrypted, so beware) to the `~/.duckdb/stored_secrets` directory.

See the Create Secret page in the documentation for more information.

Temporary Memory Manager

DuckDB has support for larger-than-memory operations, which means that memory-hungry operators such as aggregations and joins can offload part of their intermediate results to temporary files on disk should there not be enough memory available.

Before, those operators started offloading to disk if their memory usage reached around 60% of the available memory (as defined by the memory limit). This works well if there is exactly one of these operations happening at the same time. If multiple memory-intensive operations are happening simultaneously, their combined memory usage may exceed the memory limit, causing DuckDB to throw an error.

This release introduces the so-called "Temporary Memory Manager", which manages the temporary memory of concurrent operations. It works as follows: Memory-intensive operations register themselves with the Temporary Manager. Each registration is guaranteed some minimum amount of memory by the manager depending on the number of threads and the current memory limit. Then, the memory-intensive operations communicate how much memory they would currently like to use. The manager can approve this or respond with a reduced allocation. In a case of a reduced allocation, the operator will need to dynamically reduce its memory requirements, for example by switching algorithms.

For example, a hash join might adapt its operation and perform a partitioned hash join instead of a full in-memory one if not enough memory is available.

Here is an example:

```
PRAGMA memory_limit = '5GB';
SET temp_directory = '/tmp/duckdb_temporary_memory_manager';

CREATE TABLE tbl AS
SELECT range AS i,
       range AS j
FROM range(100_000_000);

SELECT max(i),
       max(t1.j),
       max(t2.j),
       max(t3.j),
FROM tbl AS t1
JOIN tbl AS t2 USING (i)
JOIN tbl AS t3 USING (i);
```

Note that a temporary directory has to be set here, because the operators actually need to offload data to disk to complete this query given this memory limit.

With the new version 0.10.0, this query completes in ca. 5 s on a MacBook, while it would error out on the previous version with `Error: Out of Memory Error: failed to pin block of size ....`

Adaptive Lossless Floating-Point Compression (ALP)

Floating point numbers are notoriously difficult to compress efficiently, both in terms of compression ratio as well as speed of compression and decompression. In the past, DuckDB had support for the then state-of-the-art "Chimp" and the "Patas" compression methods. Turns out, those were not the last word in floating point compression. Researchers Azim Afroozeh, Leonard Kuffo and (the one and only) Peter Boncz have recently published a paper titled "ALP: Adaptive Lossless floating-Point Compression" at SIGMOD, a top-tier academic conference for data management research. In an uncommon yet highly commendable move, they have also sent a pull request to DuckDB. The new compression scheme replaces Chimp and Patas. Inside DuckDB, ALP is **x2-4 times faster** than Patas (at decompression) achieving **compression ratios twice as high** (sometimes even much more).

Compression	Load	Query	Size
ALP	0.434 s	0.020 s	184 MB
Patas	0.603 s	0.080 s	275 MB
Uncompressed	0.316 s	0.012 s	489 MB

As a user, you don't have to do anything to make use of the new ALP compression method, DuckDB will automatically decide during checkpointing whether using ALP is beneficial for the specific dataset.

CLI Improvements

The command line client has seen a lot of work this release. In particular, multi-line editing has been made the default mode, and has seen many improvements. The query history is now also multi-line. Syntax highlighting has improved – missing brackets and unclosed quotes are highlighted as errors, and matching brackets are highlighted when the cursor moves over them. Compatibility with read-line has also been greatly extended.

```
D SELECT
·     l_returnflag,
·     l_linestatus, -- comment
·     sum(l_quantity) AS sum_qty, -- comment two
·     sum(l_extendedprice) AS sum_base_price,
·     sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
·     sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
▸     avg(l_quantity) AS avg_qty,
·     avg(l_extendedprice) AS avg_price,
·     avg(l_discount AS avg_disc,
·     count(*) AS count_order
· FROM
·     lineitem
· WHERE
·     l_shipdate <= CAST('1998-09-02' AS date)
· GROUP BY
·     l_returnflag,
·     l_linestatus
· ORDER BY
·     l_returnflag,
·     l_linestatus
```

See the extended CLI docs for more information.

Final Thoughts

These were a few highlights – but there are many more features and improvements in this release. Below are a few more highlights. The full release notes can be found on GitHub.

New Features

- `COMMENT ON`
- `COPY FROM DATABASE`
- `UHUGEINT` type
- Window `EXCLUDE` and Window `DISTINCT` support
- Parquet encryption support
- Indexes for Lambda parameters
- `EXCEPT ALL`/`INTERSECT ALL`
- `DESCRIBE`/`SHOW`/`SUMMARIZE` as subquery
- Support recursive CTEs in correlated subqueries

New Functions

- `parquet_kv_metadata` and `parquet_file_metadata` functions
- `read_text`/`read_blob` table functions
- `list_reduce`, `list_where`, `list_zip`, `list_select`, `list_grade_up`

Storage Improvements

- Vacuuming partial deletes
- Parallel checkpointing
- Checksum WAL

Optimizations

- Parallel streaming query result
- Struct filter pushdown
- `first(x ORDER BY y)` optimizations

Acknowledgments

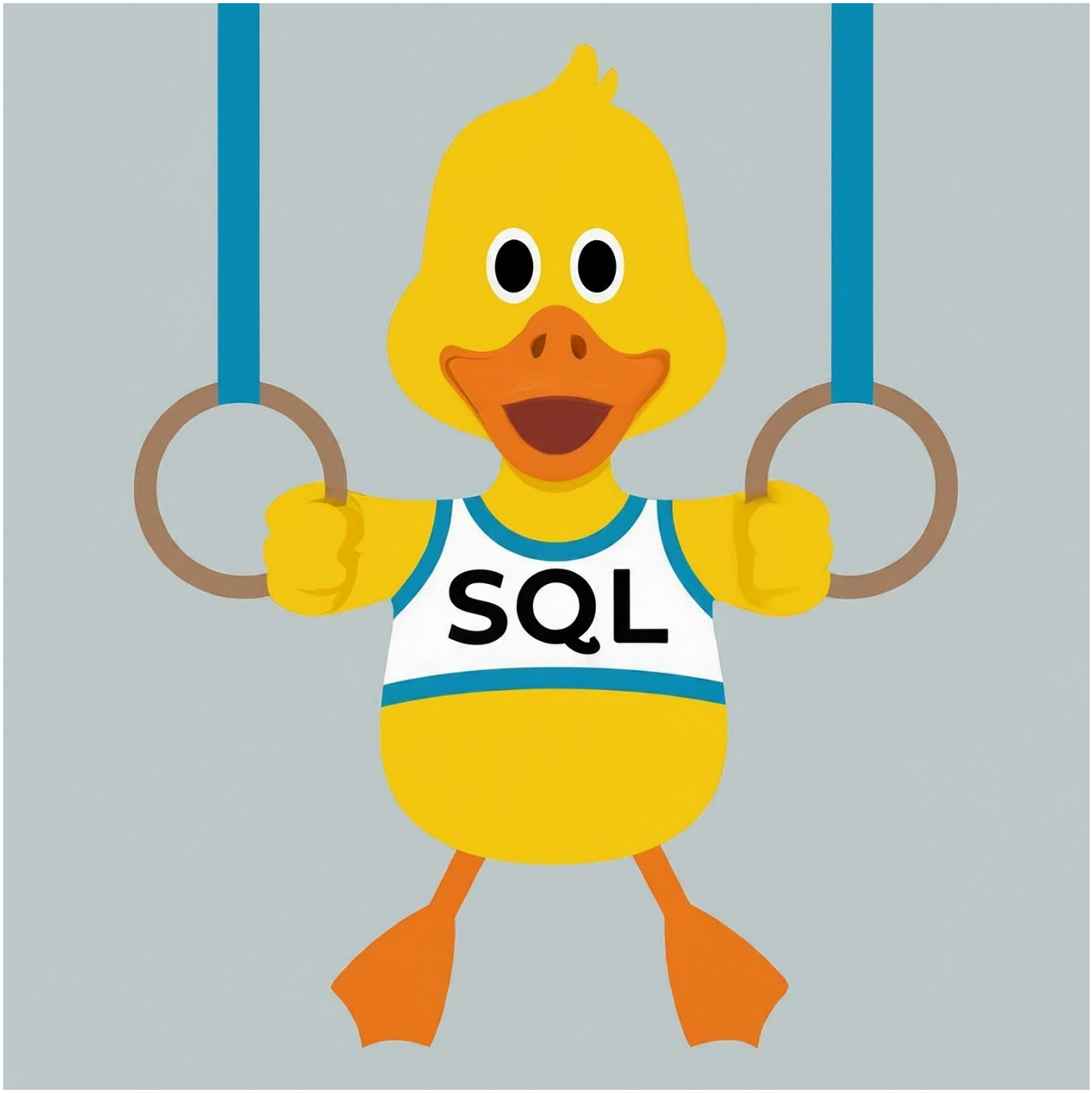
We would like to thank all of the contributors for their hard work on improving DuckDB.

SQL Gymnastics: Bending SQL into Flexible New Shapes

Publication date: 2024-03-01

Author: Alex Monahan

TL;DR: Combining multiple features of DuckDB's friendly SQL allows for highly flexible queries that can be reused across tables.



DuckDB's especially friendly SQL dialect simplifies common query operations. However, these features also unlock new and flexible ways

to write advanced SQL! In this post we will combine multiple friendly features to both move closer to real-world use cases and stretch your imagination. These queries are useful in their own right, but their component pieces are even more valuable to have in your toolbox.

What is the craziest thing you have built with SQL? We want to hear about it! Tag DuckDB on X (the site formerly known as Twitter) or LinkedIn, and join the DuckDB Discord community.

Traditional SQL Is Too Rigid to Reuse

SQL queries are typically crafted specifically for the unique tables within a database. This limits reusability. For example, have you ever seen a library of high-level SQL helper functions? SQL as a language typically is not flexible enough to build reusable functions. Today, we are flying towards a more flexible future!

Dynamic Aggregates Macro

In SQL, typically the columns to `SELECT` and `GROUP BY` must be specified individually. However, in many business intelligence workloads, groupings and aggregate functions must be easily user-adjustable. Imagine an interactive charting workflow – first I want to plot total company revenue over time. Then if I see a dip in revenue in that first plot, I want to adjust the plot to group the revenue by business unit to see which section of the company caused the issue. This typically requires templated SQL, using a language that compiles down to SQL (like Malloy), or building a SQL string using another programming language. How much we can do with just SQL?

Let's have a look at a flexible SQL-only approach and then break down how it is constructed.

col1	col2	col3	col4
1	1	1	1
2	2	0	1
3	3	1	1
4	4	0	1
5	0	1	1
6	1	0	1
7	2	1	1
8	3	0	1
9	4	1	1
10	0	0	1

Creating the Macro

The macro below accepts lists of columns to include or exclude, a list of columns to aggregate, and an aggregate function to apply. All of these can be passed in as parameters from the host language that is querying the database.

```
-- We use a table macro (or function) for reusability
CREATE OR REPLACE MACRO dynamic_aggregates(
        included_columns,
        excluded_columns,
        aggregated_columns,
        aggregate_function
    ) AS TABLE (
    FROM example
    SELECT
        -- Use a COLUMNS expression to only select the columns
        -- we include or do not exclude
        COLUMNS(c -> (
```

```
            -- If we are not using an input parameter (list is empty),
            -- ignore it
            (list_contains(included_columns, c) OR
             len(included_columns) = 0)
            AND
            (NOT list_contains(excluded_columns, c) OR
             len(excluded_columns) = 0)
            )),
        -- Use the list_aggregate function to apply an aggregate
        -- function of our choice
        list_aggregate(
            -- Convert to a list (to enable the use of list_aggregate)
            list(
                -- Use a COLUMNS expression to choose which columns
                -- to aggregate
                COLUMNS(c -> list_contains(aggregated_columns, c))
            ), aggregate_function
        )
    GROUP BY ALL -- Group by all selected but non-aggregated columns
    ORDER BY ALL -- Order by each column from left to right
);
```

Executing the Macro Now we can use that macro for many different aggregation operations. For illustrative purposes, the 3 queries below show different ways to achieve identical results.

Select col3 and col4, and take the minimum values of col1 and col2:

```
FROM dynamic_aggregates(
    ['col3', 'col4'], [], ['col1', 'col2'], 'min'
);
```

Select all columns except col1 and col2, and take the minimum values of col1 and col2:

```
FROM dynamic_aggregates(
    [], ['col1', 'col2'], ['col1', 'col2'], 'min'
);
```

If the same column is in both the included and excluded list, it is excluded (exclusions win ties). If we include col2, col3, and col4, but we exclude col2, then it is as if we only included col3 and col4:

```
FROM dynamic_aggregates(
    ['col2', 'col3', 'col4'], ['col2'], ['col1', 'col2'], 'min'
);
```

Executing either of those queries will return this result:

col3	col4	list_aggregate(list(example.col1), 'min')	list_aggregate(list(example.col2), 'min')
0	1	2	0
1	1	1	0

Understanding the Design The first step of our flexible table macro is to choose a specific table using DuckDB's FROM-first syntax. Well that's not very dynamic! If we wanted to, we could work around this by creating a copy of this macro for each table we want to expose to our application. However, we will show another approach in our next example, and completely solve the issue in a follow up blog post with an in-development DuckDB feature. Stay tuned!

Then we SELECT our grouping columns based on the list parameters that were passed in. The COLUMNS expression will execute a lambda function to decide which columns meet the criteria to be selected.

The first portion of the lambda function checks if a column name was passed in within the `included_columns` list. However, if we choose not to use an inclusion rule (by passing in a blank `included_columns` list), we want to ignore that parameter. If the list is blank,

`len(included_columns) = 0` will evaluate to `true` and effectively disable the filtering on `included_columns`. This is a common pattern for optional filtering that is generically useful across a variety of SQL queries. (Shout out to my mentor and friend Paul Bloomquist for teaching me this pattern!)

We repeat that pattern for `excluded_columns` so that it will be used if populated, but ignored if left blank. The `excluded_columns` list will also win ties, so that if a column is in both lists, it will be excluded.

Next, we apply our aggregate function to the columns we want to aggregate. It is easiest to follow the logic of this part of the query by working from the innermost portion outward. The COLUMNS expression will acquire the columns that are in our `aggregated_columns` list. Then, we do a little bit of gymnastics (it had to happen sometime...).

If we were to apply a typical aggregation function (like `sum` or `min`), it would need to be specified statically in our macro. To pass it in dynamically as a string (potentially all the way from the application code calling this SQL statement), we take advantage of a unique property of the `list_aggregate` function. It accepts the name of a function (as a string) in its second parameter. So, to use this unique property, we use the `list` aggregate function to transform all the values within each group into a list. Then we use the `list_aggregate` function to apply the `aggregate_function` we passed into the macro to each list.

Almost done! Now `GROUP BY ALL` will automatically choose to group by the columns returned by the first COLUMNS expression. The `ORDER BY ALL` expression will order each column in ascending order, moving from left to right.

We made it!

> Extra credit! In the next release of DuckDB, version 0.10.1, we will be able to apply a dynamic alias to the result of a COLUMNS expression. For example, each new aggregate column could be renamed in the pattern `agg_[the original column name]`. This will unlock the ability to chain together these type of macros, as the naming will be predictable.

Takeaways Several of the approaches used within this macro can be applied in a variety of ways in your SQL workflows. Using a lambda function in combination with the COLUMNS expression can allow you to select any arbitrary list of columns. The `OR len(my_list) = 0` trick allows list parameters to be ignored when blank. Once you have that arbitrary set of columns, you can even apply a dynamically chosen aggregation function to those columns using `list` and `list_aggregate`.

However, we still had to specify a table at the start. We are also limited to aggregate functions that are available to be used with `list_aggregate`. Let's relax those two constraints!

Creating Version 2 of the Macro

This approach takes advantage of two key concepts:

- Macros can be used to create temporary aggregate functions
- A macro can query a Common Table Expression (CTE) / `WITH` clause that is in scope during execution

```
CREATE OR REPLACE MACRO dynamic_aggregates_any_cte_any_func(
    included_columns,
    excluded_columns,
    aggregated_columns
    /* No more aggregate_function */
) AS TABLE (
    FROM any_cte -- No longer a fixed table!
    SELECT
        COLUMNS(c -> (
            (list_contains(included_columns, c) OR
            len(included_columns) = 0)
            AND
            (NOT list_contains(excluded_columns, c) OR
            len(excluded_columns) = 0)
            )),
        -- We no longer convert to a list,
        -- and we refer to the latest definition of any_func
        any_func(COLUMNS(c -> list_contains(aggregated_columns, c)))
    GROUP BY ALL
```

```
    ORDER BY ALL
);
```

Executing Version 2 When we call this macro, there is additional complexity. We no longer execute a single statement, and our logic is no longer completely parameterizable (so some templating or SQL construction will be needed). However, we can execute this macro against any arbitrary CTE, using any arbitrary aggregation function. Pretty powerful and very reusable!

```
-- We can define or redefine any_func right before calling the macro
CREATE OR REPLACE TEMP FUNCTION any_func(x)
    AS 100.0 * sum(x) / count(x);

-- Any table structure is valid for this CTE!
WITH any_cte AS (
    SELECT
        x % 11 AS id,
        x % 5 AS my_group,
        x % 2 AS another_group,
        1 AS one_big_group
    FROM range(1, 101) t(x)
)
FROM dynamic_aggregates_any_cte_any_func(
    ['another_group', 'one_big_group'], [], ['id', 'my_group']
);
```

another_group	one_big_group	any_func(any_cte.id)	any_func(any_cte.my_group)
0	1	502.0	200.0
1	1	490.0	200.0

Understanding Version 2 Instead of querying the very boldly named `example` table, we query the possibly more generically named `any_cte`. Note that `any_cte` has a different schema than our prior example – the columns in `any_cte` can be anything! When the macro is created, `any_cte` doesn't even exist. When the macro is executed, it searches for a table-like object named `any_cte`, and it was defined in the CTE as the macro was called.

Similarly, `any_func` does not exist initially. It only needs to be created (or recreated) at some point before the macro is executed. Its only requirements are to be an aggregate function that operates on a single column.

> `FUNCTION` and `MACRO` are synonyms in DuckDB and can be used interchangeably!

Takeaways from Version 2 A macro can act on any arbitrary table by using a CTE at the time it is called. This makes our macro far more reusable – it can work on any table! Not only that, but any custom aggregate function can be used.

Look how far we have stretched SQL – we have made a truly reusable SQL function! The table is dynamic, the grouping columns are dynamic, the aggregated columns are dynamic, and so is the aggregate function. Our daily gymnastics stretches have paid off. However, stay tuned for a way to achieve similar results with a simpler approach in a future post.

Custom Summaries for Any Dataset

Next we have a truly production-grade example! This query powers a portion of the MotherDuck Web UI's Column Explorer component. Hamilton Ulmer led the creation of this component and is the author of this query as well! The purpose of the Column Explorer, and this query, is to get a high-level overview of the data in all columns within a dataset as quickly and easily as possible.

DuckDB has a built-in `SUMMARIZE` keyword that can calculate similar metrics across an entire table. However, for larger datasets, `SUMMARIZE` can take a couple of seconds to load. This query provides a custom summarization capability that can be tailored to the properties of your data that you are most interested in.

Traditionally, databases required that every column be referred to explicitly, and work best when data is arranged in separate columns. This query takes advantage of DuckDB's ability to apply functions to all columns at once, its ability to UNPIVOT (or stack) columns, and its STRUCT data type to store key/value pairs. The result is a clean, pivoted summary of all the rows and columns in a table.

Let's take a look at the entire function, then break it down piece by piece.

This example dataset comes from Hugging Face, which hosts DuckDB-accessible Parquet files for many of their datasets. First, we create a local table populated from this remote Parquet file.

Creation

```
CREATE OR REPLACE TABLE spotify_tracks AS
    FROM 'https://huggingface.co/datasets/maharshipandya/spotify-tracks-
    dataset/resolve/refs%2Fconvert%2Fparquet/default/train/0000.parquet?download=true';
```

Then we create and execute our `custom_summarize` macro. We use the same `any_cte` trick from above to allow this to be reused on any query result or table.

```
CREATE OR REPLACE MACRO custom_summarize() AS TABLE (
    WITH metrics AS (
        FROM any_cte
        SELECT
            {
                name: first(alias(COLUMNS(*))),
                type: first(typeof(COLUMNS(*))),
                max: max(COLUMNS(*))::VARCHAR,
                min: min(COLUMNS(*))::VARCHAR,
                approx_unique: approx_count_distinct(COLUMNS(*)),
                nulls: count(*) - count(COLUMNS(*)),
            }
    ), stacked_metrics AS (
        UNPIVOT metrics
        ON COLUMNS(*)
    )
    SELECT value.* FROM stacked_metrics
);
```

Execution

The `spotify_tracks` dataset is effectively renamed to `any_cte` and then summarized.

```
WITH any_cte AS (FROM spotify_tracks)
FROM custom_summarize();
```

The result contains one row for every column in the raw dataset, and several columns of summary statistics.

name	type	max	min	approx_unique	nulls
Unnamed: 0	BIGINT	113999	0	114089	0
track_id	VARCHAR	7zz7iNGIWhmfFE7zlXkMma	0000vdREvCVMxbQTkS888c	89815	0
artists	VARCHAR	▯▯Ryuzo	!nvite	31545	1
album_name	VARCHAR	▯▯▯▯ ▯▯ ▯▯▯▯ Pt. 4 Original Television Soundtrack	! ! ! ! ! Whispers ! ! ! ! !	47093	1
track_name	VARCHAR	▯▯▯▯▯ ▯▯	!I'll Be Back!	72745	1
popularity	BIGINT	100	0	99	0
duration_ms	BIGINT	5237295	0	50168	0
explicit	BOOLEAN	true	false	2	0

name	type	max	min	approx_unique	nulls
danceability	DOUBLE	0.985	0.0	1180	0
energy	DOUBLE	1.0	0.0	2090	0
key	BIGINT	11	0	12	0
loudness	DOUBLE	4.532	-49.531	19436	0
mode	BIGINT	1	0	2	0
speechiness	DOUBLE	0.965	0.0	1475	0
acousticness	DOUBLE	0.996	0.0	4976	0
instrumentalness	DOUBLE	1.0	0.0	5302	0
liveness	DOUBLE	1.0	0.0	1717	0
valence	DOUBLE	0.995	0.0	1787	0
tempo	DOUBLE	243.372	0.0	46221	0
time_signature	BIGINT	5	0	5	0
track_genre	VARCHAR	world-music	acoustic	115	0

So how was this query constructed? Let's break down each CTE step by step.

Step by Step Breakdown

Metrics CTE First let's have a look at the `metrics` CTE and the shape of the data that is returned:

```
FROM any_cte
SELECT
    {
        name: first(alias(COLUMNS(*))),
        type: first(typeof(COLUMNS(*))),
        max: max(COLUMNS(*))::VARCHAR,
        min: min(COLUMNS(*))::VARCHAR,
        approx_unique: approx_count_distinct(COLUMNS(*)),
        nulls: count(*) - count(COLUMNS(*)),
    };
```

main.struct_pack(”name” := first(alias(subset.”Unnamed: 0”)), ...	main.struct_pack(”name” := first(alias(subset.track_id)), ...	...	main.struct_pack(”name” := first(alias(subset.time_signature)), ...	main.struct_pack(”name” := first(alias(subset.track_genre)), ...
{'name': Unnamed: 0, 'type': BIGINT, 'max': 113999, 'min': 0, 'approx_unique': 114089, 'nulls': 0}	{'name': track_id, 'type': VARCHAR, 'max': 7zz7iNGIWhmfFE7zlXkMma, 'min': 0000vdREvCVMxbQTkS888c, 'approx_unique': 89815, 'nulls': 0}	...	{'name': time_signature, 'type': BIGINT, 'max': 5, 'min': 0, 'approx_unique': 5, 'nulls': 0}	{'name': track_genre, 'type': VARCHAR, 'max': world-music, 'min': acoustic, 'approx_unique': 115, 'nulls': 0}

This intermediate result maintains the same number of columns as the original dataset, but only returns a single row of summary statistics. The names of the columns are truncated due to their length. The default naming of `COLUMNS` expressions will be improved in DuckDB 0.10.1, so names will be much cleaner!

The data in each column is organized into a STRUCT of key-value pairs. You can also see that a clean name of the original column is stored within the STRUCT thanks to the use of the `alias` function. While we have calculated the summary statistics, the format of those statistics is difficult to visually interpret.

The query achieves this structure using the `COLUMNS(*)` expression to apply multiple summary metrics to all columns, and the `{...}` syntax to create a STRUCT. The keys of the struct represent the names of the metrics (and what we want to use as the column names in the final result). We use this approach since we want to transpose the columns to rows and then split the summary metrics into their own columns.

`stacked_metrics` CTE Next, the data is unpivoted to reshape the table from one row and multiple columns to two columns and multiple rows.

```
UNPIVOT metrics
ON COLUMNS(*);
```

name	value
main.struct_pack(”name” := first(alias(spotify_tracks.”Unnamed: 0”)), ...	{'name': Unnamed: 0, 'type': BIGINT, 'max': 113999, 'min': 0, 'approx_unique': 114089, 'nulls': 0}
main.struct_pack(”name” := first(alias(spotify_tracks.track_id)), ...	{'name': track_id, 'type': VARCHAR, 'max': 7zz7iNGIWhmfFE7zlXkMma, 'min': 0000vdREvCVMxbQTkS888c, 'approx_unique': 89815, 'nulls': 0}
...	...
main.struct_pack(”name” := first(alias(spotify_tracks.time_signature)), ...	{'name': time_signature, 'type': BIGINT, 'max': 5, 'min': 0, 'approx_unique': 5, 'nulls': 0}
main.struct_pack(”name” := first(alias(spotify_tracks.track_genre)), ...	{'name': track_genre, 'type': VARCHAR, 'max': world-music, 'min': acoustic, 'approx_unique': 115, 'nulls': 0}

By unpivoting on `COLUMNS(*)`, we take all columns and pivot them downward into two columns: one for the auto-generated `name` of the column, and one for the `value` that was within that column.

Return the Results The final step is the most gymnastics-like portion of this query. We explode the `value` column's struct format so that each key becomes its own column using the `STRUCT.*` syntax. This is another way to make a query less reliant on column names – the split occurs automatically based on the keys in the struct.

```
SELECT value.*
FROM stacked_metrics;
```

We have now split apart the data into multiple columns, so the summary metrics are nice and interpretable.

name	type	max	min	approx_unique	nulls
Unnamed: 0	BIGINT	113999	0	114089	0
track_id	VARCHAR	7zz7iNGIWhmfFE7zlXkMma	0000vdREvCVMxbQTkS888c	89815	0
artists	VARCHAR	▯▯Ryuzo	!nvite	31545	1
album_name	VARCHAR	▯▯▯ ▯▯ ▯▯▯ Pt. 4 Original Television Soundtrack	! ! ! ! ! Whispers ! ! ! ! !	47093	1
track_name	VARCHAR	▯▯▯▯ ▯▯	!I'll Be Back!	72745	1
popularity	BIGINT	100	0	99	0
duration_ms	BIGINT	5237295	0	50168	0
explicit	BOOLEAN	true	false	2	0
danceability	DOUBLE	0.985	0.0	1180	0
energy	DOUBLE	1.0	0.0	2090	0
key	BIGINT	11	0	12	0

name	type	max	min	approx_ unique	nulls
loudness	DOUBLE	4.532	-49.531	19436	0
mode	BIGINT	1	0	2	0
speechiness	DOUBLE	0.965	0.0	1475	0
acousticness	DOUBLE	0.996	0.0	4976	0
instrumentalness	DOUBLE	1.0	0.0	5302	0
liveness	DOUBLE	1.0	0.0	1717	0
valence	DOUBLE	0.995	0.0	1787	0
tempo	DOUBLE	243.372	0.0	46221	0
time_signature	BIGINT	5	0	5	0
track_genre	VARCHAR	world-music	acoustic	115	0

Conclusion

We have shown that it is now possible to build reusable SQL macros in a highly flexible way. You can now build a macro that:

- Operates on any dataset
- Selects any columns
- Groups by any columns
- Aggregates any number of columns with any function.

Phew!

Along the way we have covered some useful tricks to have in your toolbox:

- Applying a macro to any dataset using a CTE
- Selecting a dynamic list of columns by combining the `COLUMNS` expression with a lambda and the `list_contains` function
- Passing in an aggregate function as a string using `list_aggregate`
- Applying any custom aggregation function within a macro
- Making list parameters optional using `OR len(list_parameter) = 0`
- Using the `alias` function with a `COLUMNS` expression to store the original name of all columns
- Summarizing all columns and then transposing that summary using `UNPIVOT` and `STRUCT.*`

The combination of these friendly SQL features is more powerful than using any one individually. We hope that we have inspired you to take your SQL to new limits!

As always, we welcome your feedback and suggestions. We also have more flexibility in mind that will be demonstrated in future posts. Please share the times you have stretched SQL in imaginative ways!

Happy analyzing!

Dependency Management in DuckDB Extensions

Publication date: 2024-03-22

Author: Sam Ansmink

TL;DR: While core DuckDB has zero external dependencies, building extensions with dependencies is now very simple, with built-in support for vcpkg, an open-source package manager with support for over 2000 C/C++ packages. Interested in building your own? Check out the extension template.

Introduction

Ever since the birth of DuckDB, one of its main pillars has been its strict no-external-dependencies philosophy. Paraphrasing this 2019 SIGMOD paper on DuckDB: *To achieve the requirement of having practical "embeddability" and portability, the database needs to run in whatever environment the host does. Dependencies on external libraries (e.g., openssh) for either compile- or runtime have been found to be problematic.*

In this blog post, we will cover how DuckDB manages to stay true to this philosophy without forcing DuckDB developers down the path of complete abstinence. Along the way, we will show practical examples of how external dependencies are possible, and how you can use this when creating your own DuckDB extension.

The Difficulties of Complete Abstinence

Having no external dependencies is conceptually very simple. However, in a real-world system with real-world requirements, it is difficult to achieve. Many features require complex implementations of protocols and algorithms, and many high-quality libraries exist that implement them. What this means for DuckDB (and most other systems, for that matter) is that there are basically three options for handling requirements with potential external dependencies:

1. Inlining external code
2. Rewriting the external dependency
3. Breaking the no-dependency rule

The first two options are pretty straightforward: to avoid depending on some external software, just make it part of the codebase. By doing so, the unpredictable nature of depending on somebody else is now eliminated! DuckDB has applied both inlining and rewriting to prevent dependencies. For example, the Postgres parser and MbedTLS libraries are inlined into DuckDB, whereas the S3 support is provided using a custom implementation of the AWS S3 protocol.

Okay, great – problem solved, right? Well, not so fast. Most people with some software engineering experience will realize that both inlining and rewriting come with serious drawbacks. The most fundamental issue is probably related to code maintenance. Every significant piece of software needs some level of maintenance. Ranging from fixing bugs to dealing with changing (build) environments or requirements, code will need to be modified to stay functional and relevant. When inlining/rewriting dependencies, this also copies over the maintenance burden.

For DuckDB, this historically meant that for each dependency, very careful consideration was made to balance the increased maintenance burden against the necessity of dependency. Including a dependency meant the responsibility of maintaining it, so this decision was never taken lightly. This works well in many cases and has the added benefit of forcing developers to think critically about including a dependency and not mindlessly bolt on library after library. However, for some dependencies, this just doesn't work. Take, for example, the SDKs of large cloud providers. They tend to be pretty massive, very frequently updated, and packed with arguably essential functionality for an increasingly mature analytical database. This leaves an awkward choice: either not provide these essential features or break the no-dependency rule.

DuckDB Extensions

This is where extensions come in. Extensions provide an elegant solution to the dilemma of dependencies by allowing fine-grained breakage of the no-dependency rule. Moving dependencies out of DuckDB's core into extensions, the core codebase can remain, and does remain, dependency-free. This means that DuckDB's "Practical embeddability and portability" remains unthreatened. On the other hand, DuckDB can still provide features that inevitably require depending on some 3rd party library. Furthermore, by moving dependencies to extensions, each extension can have different levels of exposure to instability from dependencies. For example, some extensions may choose to depend only on highly mature, stable libraries with good portability, whereas others may choose to include more experimental dependencies with limited portability. This choice is then forwarded to the user by allowing them to choose which extension to use.

At DuckDB, this realization of the importance of extensions and its relation to the no-dependency rule came very early, and consequently extensibility has been ingrained into DuckDB's design since its early days. Today, many parts of DuckDB can be extended. For example, you can add functions (table, scalar, copy, aggregation), filesystems, parsers, optimizer rules, and much more. Many new features that are added to DuckDB are added in extensions and are grouped by either functionality or by set of dependencies. Some examples of extensions are the SQLite extension for reading/writing to/from SQLite files or the Spatial extension which offers support for a wide range of geospatial processing features. DuckDB's extensions are distributed as loadable binaries for most major platforms (including DuckDB-Wasm), allowing loading and installing extensions with two simple SQL statements:

```
INSTALL spatial;
LOAD spatial;
```

For most core extensions maintained by the DuckDB team, there is even an auto-install and auto-load feature which will detect the required extensions for a SQL statement and automatically install and load them. For a detailed description of which extensions are available and how to use them, check out the docs.

Dependency Management

So far, we've seen how DuckDB avoids external dependencies in its core codebase by moving them out of the core repository into extensions. However, we're not out of the woods yet. As DuckDB is written in C++, the most natural way to write extensions is C++. In C++, though, there is no standard tooling like a package manager and the answer to the question of how to do dependency management in C++ has been, for many years: *"Through much pain and anguish."* Given DuckDB's focus on portability and support for many platforms, managing dependencies manually is not feasible: dependencies generally are built from source, with each their own intricacies requiring special build flags and configuration for different platforms. With a growing ecosystem of extensions, this would quickly turn into an unmaintainable mess.

Fortunately, much has changed in the C++ landscape over the past few years. Today, good dependency managers do exist. One of them is Microsoft's vcpkg. It has become a highly notable player among C++ dependency managers, as proven by its 20k+ GitHub stars and native support from CLion and Visual Studio. vcpkg contains over 2000 dependencies such as Apache Arrow, yyjson, and various cloud provider SDKs.

For anyone who has ever used a package manager, using vcpkg will feel quite natural. Dependencies are specified in a `vcpkg.json` file, and vcpkg is hooked into the build system. Now, when building, vcpkg ensures that the dependencies specified in the `vcpkg.json` are built and available. vcpkg supports integration with multiple build systems, with a focus on its seamless CMake integration.

Using vcpkg with DuckDB

Now that we covered DuckDB extensions and vcpkg, we have shown how DuckDB can manage dependencies without sacrificing portability, maintainability and stability more than necessary. Next, we'll make things a bit more tangible by looking at one of DuckDB's extensions and how it uses vcpkg to manage its dependencies.

Example: Azure extension

The Azure extension provides functionality related to Microsoft Azure, one of the major cloud providers. DuckDB's Azure extension depends on the Azure C++ SDK to support reading directly from Azure Storage. To do so it adds a custom filesystem and secret type, which can be used to easily query from authenticated Azure containers:

```
CREATE SECRET az1 (
    TYPE azure,
    CONNECTION_STRING '<redacted>'
);
SELECT column_a, column_b
FROM 'az://my-container/some-file.parquet';
```

To implement these features, the Azure extension depends on different parts of the Azure SDK. These are specified in the Azure extensions `vcpkg.json`:

```
{
  "dependencies": [
    "azure-identity-cpp",
    "azure-storage-blobs-cpp",
    "azure-storage-files-datalake-cpp"
  ]
}
```

Then, in the Azure extension's `CMakelists.txt` file, we find the following lines:

```
find_package(azure-identity-cpp CONFIG)
find_package(azure-storage-blobs-cpp CONFIG)
find_package(azure-storage-files-datalake-cpp CONFIG)

target_link_libraries(${EXTENSION_NAME} Azure::azure-identity Azure::azure-storage-blobs
Azure::azure-storage-files-datalake)
target_include_directories(${EXTENSION_NAME} PRIVATE Azure::azure-identity Azure::azure-storage-blobs
Azure::azure-storage-files-datalake)
```

And that's basically it! Every time the Azure extension is built, vcpkg will be called first to ensure `azure-identity-cpp`, `azure-storage-blobs-cpp` and `azure-storage-files-datalake-cpp` are built using the correct platform-specific flags and available in CMake through `find_package`.

Building Your Own DuckDB Extension

Up until this part, we've focused on managing dependencies from a point-of-view of the developers of core DuckDB contributors. However, all of this applies to anyone who wants to build an extension. DuckDB maintains a C++ Extension Template, which contains all the necessary build scripts, CI/CD pipeline and vcpkg configuration to build, test and deploy a DuckDB extension in minutes. It can automatically build the loadable extension binaries for all available platforms, including Wasm.

Setting up the Extension Template

To demonstrate how simple this process is, let's go through all the steps of building a DuckDB extension from scratch, including adding a vcpkg-managed external dependency.

Firstly, you will need to install vcpkg:

```
git clone https://github.com/Microsoft/vcpkg.git
./vcpkg/bootstrap-vcpkg.sh
export VCPKG_TOOLCHAIN_PATH=`pwd`/vcpkg/scripts/buildsystems/vcpkg.cmake
```

Then, you create a GitHub repository based on the template by clicking “Use this template”.

Now to clone your newly created extension repo (including its submodules) and initialize the template:

```
git clone --recurse-submodules \
    https://github.com/<your_username>/<your_extension_repo>
cd your-extension-repo
./scripts/bootstrap-template.py url_parser
```

Finally, to confirm everything works as expected, run the tests:

```
make test
```

Adding Functionality

In its current state, the extension is, of course, a little boring. Therefore, let's add some functionality! To keep things simple, we'll add a scalar function that parses a URL and returns the scheme. We'll call the function `url_scheme`. We start by adding a dependency to the boost url library in our `vcpkg.json` file:

```
{
  "dependencies": [
    "boost-url"
  ]
}
```

Then, we follow up with changing our `CMakelists.txt` to ensure our dependencies are correctly included in the build.

```
find_package(Boost REQUIRED COMPONENTS url)
target_link_libraries(${EXTENSION_NAME} Boost::url)
target_link_libraries(${LOADABLE_EXTENSION_NAME} Boost::url)
```

Then, in `src/url_parser_extension.cpp`, we remove the default example functions and replace them with our implementation of the `url_scheme` function:

```
inline void UrlParserScalarFun(DataChunk &args, ExpressionState &state, Vector &result) {
  auto &name_vector = args.data[0];
  UnaryExecutor::Execute<string_t, string_t>(
    name_vector, result, args.size(),
    [&](string_t url) {
          string url_string = url.GetString();
          boost::system::result<boost::urls::url_view> parse_result = boost::urls::parse_uri( url_string );
          if (parse_result.has_error() || !parse_result.value().has_scheme()) {
              return string_t();
          }
          string scheme = parse_result.value().scheme();
          return StringVector::AddString(result, scheme);
      });
}

static void LoadInternal(DatabaseInstance &instance) {
  auto url_parser_scalar_function = ScalarFunction("url_scheme", {LogicalType::VARCHAR},
  LogicalType::VARCHAR, UrlParserScalarFun);
  ExtensionUtil::RegisterFunction(instance, url_parser_scalar_function);
}
```

With our extension written, we can run `make` to build both DuckDB and the extension. After the build is finished, we are ready to try out our extension. Since the build process also builds a fresh DuckDB binary with the extension loaded automatically, all we need to do is run `./build/release/duckdb`, and we can use our newly added scalar function:

```
SELECT url_scheme('https://github.com/duckdb/duckdb');
```

Finally, as we are well-behaved developers, we add some tests by overwriting the default test `test/sql/url_parser.test` with:

```
require url_parser

# Confirm the extension works
query I
SELECT url_scheme('https://github.com/duckdb/duckdb')
----
https

# On parser errors or not finding a scheme, the result is also an empty string
query I
SELECT url_scheme('not:\a/valid_url')
----
(empty)
```

Now all that's left to do is confirm everything works as expected with `make test`, and push these changes to the remote repository. Then, GitHub Actions will take over and ensure the extension is built for all of DuckDB's supported platforms.

For more details, check out the template repository. Also, the example extension we built in this blog is published on GitHub. Note that in the demo, the Wasm and MinGW builds have been disabled due to outstanding issues with the boost-url dependency for building on these platforms. As these issues are fixed upstream, re-enabling their builds for the extension is very simple. Of course, as the author of this extension, it could make a lot of sense to fix these compile issues yourself in vcpkg and fix them not only for this extension, but for the whole open-source community!

Conclusion

In this blog post, we've explored DuckDB's journey towards managing dependencies in its extension ecosystem while upholding its core philosophy of zero external dependencies. By leveraging the power of extensions, DuckDB can maintain its portability and embeddability while still providing essential features that require external dependencies. To simplify managing dependencies, Microsoft's vcpkg is integrated into DuckDB's extension build systems both for DuckDB-maintained extension and third-party extensions.

If this blog post sparked your interest in creating your own DuckDB extension, check out the C++ Extension Template, the DuckDB docs on extensions, and the very handy duckdb-extension-radar repository that tracks public DuckDB extensions. Additionally, DuckDB has a Discord server where you can ask for help on extensions or anything DuckDB-related in general.

42.parquet – A Zip Bomb for the Big Data Age

Publication date: 2024-03-26

Author: Hannes Mühleisen

TL;DR: A 42 kB Parquet file can contain over 4 PB of data.

Apache Parquet has become the de-facto standard for tabular data interchange. It is greatly superior to its scary cousin CSV by using a binary, columnar and *compressed* data representation. In addition, Parquet files come with enough metadata so that files can be correctly interpreted without additional information. Most modern data tools and services support reading and writing Parquet files.

However, Parquet files are not without their dangers: For example, corrupt files can crash readers that are not being very careful in interpreting internal offsets and such. But even perfectly valid files can be problematic and lead to crashes and service downtime as we will show below.

A pretty well-known attack on naive firewalls and virus scanners is a Zip Bomb, one famous example being 42.zip, named so because of course 42 is the perfect number and the file is only 42 kilobytes large. This perfectly-valid zip file has a bunch of other zip files in it, which again contain other zip files and so on. Eventually, if one would try to unpack all of that, you would end up with 4 petabytes of data. Big Data indeed.

Parquet files support various methods to compress data. How big of a table can one create with a Parquet file that is only 42 kilobytes large in the spirit of a zip bomb? Let's find out! For reasons of portability, we have implemented our own Parquet reader and writers for DuckDB. It is unavoidable to learn a great deal about the Parquet format when implementing it.

A Parquet file is made up of one or more row groups, which contain columns, which in turn contain so-called pages that contain the actual data in encoded format. Among other encodings, Parquet supports dictionary encoding, where we first have a page with a dictionary, followed by data pages that refer to the dictionary instead of containing plain values. This is more efficient for columns where long values such as categorical strings repeat often, because the dictionary references can be much smaller.

Let's exploit that. We write a dictionary with a single value and refer to it over and over. In our example, we use a single 64-bit integer, the biggest possible value because why not. Then, we refer back to this dictionary entry using the `RLE_DICTIONARY` run-length encoding specified in Parquet. The specified encoding is a bit weird because for some reason it combines bit packing and run-length encoding but essentially we can use the biggest run-length possible, which is `2^31-1`, a little over 2 billion. Since the dictionary is tiny (one entry), the value we repeat is 0, referring to the only entry. Including its required metadata headers and footers (like all metadata in Parquet, this is encoded using Thrift), this file is only 133 bytes large. 133 bytes to represent 2 billion 8-byte integers is not too bad, even if they're all the same.

But we can go up from there. Columns can contain multiple pages referring to *the same* dictionary, so we can just repeat our data page over and over, each time only adding 31 bytes to the file, but 2 billion values to the table the file represents. We can also use another trick to blow up the data size: as mentioned, Parquet files contain one or more row groups, those are stored in a Thrift footer at the end of the file. Each column in this row group contains byte offsets (`data_page_offset` and friends) into the file where the pages for the columns are stored. Nothing keeps us from adding multiple row groups that *all refer to the same byte offset*, the one where we stored our slightly mischievous dictionary and data pages. Each row group we add logically repeats all the pages. Of course, adding row groups also requires metadata storage, so there is some sort of trade-off between adding pages (2 billion values) and row groups (2x whatever other row group it duplicates).

With some fiddling, we found that if we repeat the data page 1000 times and repeat the row group 290 times, we end up with a Parquet file (obtainable from the `fortytwodotparquet` repository) that is 42 kilobytes large, yet contains *622 trillion* values (622,770,257,630,000 to be exact). If one would materialize this table in memory, it would require over *4 petabytes* of memory, finally a real example of Big Data, coincidentally roughly the same size as the original `42.zip` mentioned above.

We've made the script that we use to generate this file available as well, we hope it can be used to test Parquet readers better. We hope to have shown that Parquet files can be considered harmful and should certainly not be shoved into some pipeline without being extra careful. And while DuckDB *can* read data from our file (e.g., with a `LIMIT`), if you would make it read through it all, you better get some coffee.

No Memory? No Problem. External Aggregation in DuckDB

Publication date: 2024-03-29

Author: Laurens Kuiper

TL;DR: Since the 0.9.0 release, DuckDB's fully parallel aggregate hash table can efficiently aggregate over many more groups than fit in memory.

Most grouped aggregation queries yield just a few output rows. For example, "How many flights departed from each European capital in the past ten years?" yields one row per European capital, even if the table containing all the flight information has millions of rows. This is not always the case, as "How many orders did each customer place in the past ten years?" yields one row per customer, which could be millions, which significantly increases the memory consumption of the query. However, even if the aggregation does not fit in memory, DuckDB can still complete the query.

Not interested in the implementation? Jump straight to the experiments!

Introduction

Around two years ago, we published our first blog post on DuckDB's hash aggregation, titled "Parallel Grouped Aggregation in DuckDB". So why are we writing another blog post now?

Unlike most database systems, which are servers, DuckDB is used in all kinds of environments, which may not have much memory. However, some database queries, like aggregations with many unique groups, require a lot of memory. The laptop I am writing this on has 16 GB of RAM. What if a query needs 20 GB? If this happens:

```
Out of Memory Error: could not allocate block of size X (Y/Z used)
```

The query is aborted. Sadly, we can't download more RAM. But luckily, this laptop also has a fast SSD with 1 TB of storage. In many cases, we don't need all 20 GB of data to be in memory simultaneously, and we can temporarily place some data in storage. If we load it back whenever needed, we can still complete the query. We must be careful to use storage sparingly because despite modern SSDs being fast, they are still much slower than memory.

In a nutshell, that's what this post is about. Since the 0.9.0 release, DuckDB's hash aggregation can process more unique groups than fit in memory by offloading data to storage. In this post, we'll explain how this works. If you want to know what hash aggregation is, how hash collisions are resolved, or how DuckDB's hash table is structured, check out our first blog post on hash aggregation.

Memory Management

Most database systems store persistent data on "pages". Upon request, these pages can be read from the *database file* in storage, put into memory, and written back again if necessary. The common wisdom is to make all pages the same size: This allows pages to be swapped and avoids fragmentation in memory and storage. When the database is started, a portion of memory is allocated and reserved for these pages, called the "buffer pool". The database component that is responsible for managing the buffer pool is aptly called the "buffer manager".

The remaining memory is reserved for short-lived, i.e., *temporary*, memory allocations, such as hash tables for aggregation. These allocations are done differently, which is good because if there are many unique groups, hash tables may need to be very large, so we wouldn't have been able to use the fixed-size pages for that anyway. If we have more temporary data than fits in memory, operators like aggregation have to decide when to selectively write data to a *temporary file* in storage.

... At least, that's the traditional way of doing things. This made little sense for DuckDB. Why should we manage persistent and temporary data so differently? The difference is that *persistent* data should be *persisted*, and *temporary* data should not. Why can't a buffer manager manage both?

DuckDB's buffer manager is not traditional. Most persistent and temporary data is stored on fixed-size pages and managed by the buffer manager. The buffer manager tries to make the best use of your memory. That means we don't reserve a portion of memory for a buffer pool. This allows DuckDB to use all memory for persistent data, not just a portion if that's what's best for your workload. If you're doing large aggregations that need a lot of memory, DuckDB can evict the persistent data from memory to free up space for a large hash table.

Because DuckDB's buffer manager manages *all* memory, both persistent and temporary data, it is much better at choosing when to write temporary data to storage than operators like aggregation could ever be. Leaving the responsibility of offloading to the buffer manager also saves us the effort of implementing reading and writing data to a temporary file in every operator that needs to process data that does not fit in memory.

Why don't buffer managers in other database systems manage temporary data? There are two problems: *Memory Fragmentation* and *Invalid References*.

Memory Fragmentation

Hash tables and other data structures used in query operators don't exactly have a fixed size like the pages used for persistent data. We also don't want to have a lot of pages with variable sizes floating around in memory alongside the pages with a fixed size, as this would cause memory fragmentation.

Ideally, we would use the fixed size for *all* of our memory allocations, but this is not a good idea: Sometimes, the most efficient way to process a query requires allocating, for example, a large array. So, we settled for using a fixed size for *almost all* of our allocations. These short-lived allocations are immediately deallocated after use, unlike the fixed-size pages for persistent data, which are kept around. These allocations do not cause fragmentation with each other because jemalloc, which DuckDB uses for allocating memory when possible, categorizes allocations using size classes and maintains separate arenas for them.

Invalid References

Temporary data usually cannot be written to storage as-is because it often contains pointers. For example, DuckDB implements the string type proposed by Umbra, which has a fixed width. Strings longer than 12 characters are not stored within the string type, but *somewhere else*, and a pointer to this "somewhere else" is stored instead.

This creates a problem when we want to offload data to storage. Let's say this "somewhere else" where strings longer than 12 characters are stored is one of those pages that the buffer manager can offload to storage at any time to free up some memory. If the page is offloaded and then loaded back, it will most likely be loaded into a different address in memory. The pointers that pointed to the long strings are now *invalid* because they still point to the previous address!

The usual way of writing data containing pointers to storage is by *serializing* it first. When reading it back into memory, it has to be *deserialized* again. (De-)serialization can be an expensive operation, hence why data formats like Arrow Flight exist, which try to minimize the cost. However, we can't use Arrow here because Arrow is a column-major layout, but a row-major layout is more efficient for hash tables.

We could create a row-major version of Arrow Flight, but we can just avoid (de-)serialization altogether: We've created a specialized row-major *page layout* that actually uses the old invalidated pointers to *recompute* new valid pointers after reading the data back into memory.

The page layout places fixed-size rows and variable-size data like strings on separate pages. The size of the rows is fixed for a query: After a SQL query is issued, DuckDB creates and executes a query plan. So, even before executing the said plan, we already know which columns we need, their types, and how wide these types are.

As shown in the image below, a small amount of "MetaData" is needed to recompute the pointers. The fixed-size rows are stored in "Row Pages", and variable-size rows in "Var Pages".

Remember that there are pointers within the fixed-size rows pointing to variable-size data. The MetaData describes which fixed-size rows point to which Var Page and the last known address of the Var Page. For example, MetaData 1 describes 5 rows stored in Row Page 1 at offset 0, with variable-size data stored in Var Page 1, which had an address of `0x42`.

Let's say the buffer manager decides to offload Var Page 1. When we request Var Page 1 again, it's loaded into address `0x500`. The pointers within those 5 rows are now invalid. For example, one of the rows contains the pointer `0x48`, which means that it is stored at offset `0x48`

`- 0x42 = 6` in Var Page 1. We can recompute the pointer by adding the offset to the new address of the page: `0x500 + 6 = 0x506`. Pointer recomputation is done for rows with their strings stored on the same Row and Var Page, so we create a new MetaData every time a Row Page or Var Page is full.

The advantage of pointer recomputation over (de-)serialization is that it can be done lazily. We can check whether the Var Page was offloaded by comparing the pointer in the MetaData with the current pointer to the page. We don't have to recompute the pointers if they are the same.

External Aggregation

Now that we've figured out how to deal with temporary data, it's finally time to talk about hash aggregation. The first big challenge is to perform the aggregation in parallel.

DuckDB uses Morsel-Driven Parallelism to parallelize query execution, which essentially means that query operators, such as aggregation, must be parallelism-aware. This differs from plan-driven parallelism, keeping operators unaware of parallelism.

To briefly summarize our first blog post on aggregation: In DuckDB, all active threads have their own thread-local hash table, which they sink input data into. This will keep threads busy until all input data has been read. Multiple threads will likely have the *exact same group* in their hash table. Therefore, the thread-local hash tables must be combined to complete the grouped aggregation. This can be done in parallel by partitioning the hash tables and assigning each thread to combine the data from each partition. For the most part, we still use this same approach. You'll see this in the image below, which illustrates our new implementation.

We call the first phase *Thread-Local Pre-Aggregation*. The input data are *morsels*, chunks of around 100,000 rows. These are assigned to active threads, which sink them into their thread-local hash table until all input data has been read. We use *linear probing* to resolve collisions and *salt* to reduce the overhead of dealing with said collisions. This is explained in our first blog post on aggregation, so I won't repeat it here.

Now that we've explained what *hasn't* changed, we can talk about what *has* changed. The first difference compared to last time is the way that we partition. Before, if we had, for example, 32 threads, each thread would create 32 hash tables, one for each partition. This totals a whopping 1024 hash tables, which did not scale well when even more threads were active. Now, each thread has one hash table, *but the data within each hash table is partitioned*. The data is also stored on the specialized page layout we presented earlier so that it can easily be offloaded to storage.

The second difference is that the hash tables are not *resized* during Thread-Local Pre-Aggregation. We keep the hash tables' size small, reducing the amount of cache misses during this phase. This means that the hash table will be full at some point. When it's full, we reset it and start over. We can do this because we'll finish the aggregation later in the second phase. When we reset the hash table, we "unpin" the pages that store the actual data, which tells our buffer manager it can write them to storage when it needs to free up memory.

Together, these two changes result in a low memory requirement during the first phase. Each thread only needs to keep a small hash table in memory. We may collect a lot of data by filling up the hash table many times, but the buffer manager can offload almost all of it if needed.

For the second phase, *Partition-Wise Aggregation*, the thread-local partitioned data is exchanged, and each thread combines the data of a single partition into a hash table. This phase is mostly the same as before, except that we now sometimes create many more partitions than threads. Why? The hash table for one partition might fit in memory, but 8 threads could be combining a partition simultaneously, and we might not be able to fit 8 partitions in memory. The easy solution to this problem is to *over-partition*. If we make more partitions than threads, for example, 32 partitions, the size of the partitions will be smaller, and the 8 threads will combine only 8 out of the 32 partitions simultaneously, which won't require nearly as much memory.

Experiments

Aggregations that result in only a few unique groups can easily fit in memory. To evaluate our external hash aggregation implementation, we need aggregations that have many unique groups. For this purpose, we will use the H2O.ai database-like ops benchmark, which we've resurrected, and now maintain. Specifically, we will use the `G1_1e9_2e0_0_0.csv.zst` file, which is 50 GB uncompressed. The source code for the H2O.ai benchmark can be found on GitHub. You can download the file yourself from https://blobs.duckdb.org/data/G1_1e9_2e0_0_0.csv.zst (18.8 GB compressed).

We use the following queries from the benchmark to load the data:

```
SET preserve_insertion_order = false;
CREATE TABLE y (
    id1 VARCHAR, id2 VARCHAR, id3 VARCHAR,
    id4 INTEGER, id5 INTEGER, id6 INTEGER,
    v1 INTEGER, v2 INTEGER, v3 FLOAT);
COPY y FROM 'G1_1e9_2e0_0_0.csv.zst' (FORMAT csv, AUTO_DETECT true);
CREATE TYPE id1ENUM AS ENUM (SELECT id1 FROM y);
CREATE TYPE id2ENUM AS ENUM (SELECT id2 FROM y);
CREATE TABLE x (
    id1 id1ENUM, id2 id2ENUM, id3 VARCHAR,
    id4 INTEGER, id5 INTEGER, id6 INTEGER,
    v1 INTEGER, v2 INTEGER, v3 FLOAT);
INSERT INTO x (SELECT * FROM y);
DROP TABLE IF EXISTS y;
```

The H2O.ai aggregation benchmark consists of 10 queries, which vary in the number of unique groups:

```
-- Query 1: ~100 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id1, sum(v1) AS v1
FROM x
GROUP BY id1;

-- Query 2: ~10,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id1, id2, sum(v1) AS v1
FROM x
GROUP BY id1, id2;

-- Query 3: ~10,000,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id3, sum(v1) AS v1, avg(v3) AS v3
FROM x
GROUP BY id3;

-- Query 4: ~100 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id4, avg(v1) AS v1, avg(v2) AS v2, avg(v3) AS v3
FROM x
GROUP BY id4;

-- Query 5: ~1,000,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id6, sum(v1) AS v1, sum(v2) AS v2, sum(v3) AS v3
FROM x
GROUP BY id6;

-- Query 6: ~10,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT
    id4,
    id5,
    quantile_cont(v3, 0.5) AS median_v3,
    stddev(v3) AS sd_v3
FROM x
GROUP BY id4, id5;

-- Query 7: ~10,000,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id3, max(v1) - min(v2) AS range_v1_v2
FROM x
GROUP BY id3;

-- Query 8: ~10,000,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id6, v3 AS largest2_v3
```

```
FROM (
    SELECT id6, v3, row_number() OVER (
          PARTITION BY id6
          ORDER BY v3 DESC) AS order_v3
    FROM x
    WHERE v3 IS NOT NULL) sub_query
WHERE order_v3 <= 2;

-- Query 9: ~10,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id2, id4, pow(corr(v1, v2), 2) AS r2
FROM x
GROUP BY id2, id4;

-- Query 10: ~1,000,000,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id1, id2, id3, id4, id5, id6, sum(v3) AS v3, count(*) AS count
FROM x
GROUP BY id1, id2, id3, id4, id5, id6;
```

The results on the benchmark page are obtained using the `c6id.metal` AWS EC2 instance. On this instance, all the queries easily fit in memory, and having many threads doesn't hurt performance either. DuckDB only takes 8.58 seconds to complete even the largest query, query 10, which returns 1 billion unique groups. However, many people will not use such a beefy machine to crunch numbers. On my laptop, a 2020 MacBook Pro, some smaller queries will fit in memory, like query 1, but query 10 will definitely not.

The following table is a summary of the hardware used.

Specs	`c6id.metal`	Laptop	Ratio
Memory	256 GB	16 GB	16×
CPU cores	64	8	8×
CPU threads	128	8	16×
Hourly cost	$6.45	$0.00	NaN

Although the CPU cores of the AWS EC2 instance are not directly comparable with those of my laptop, the instance clearly has much more compute power and memory available. Despite the large differences in hardware, DuckDB can complete all 10 queries without a problem:

Query	`c6id.metal`	Laptop	Ratio
1	0.08	0.74	9.25×
2	0.09	0.76	8.44×
3	8.01	156.63	19.55×
4	0.26	2.07	7.96×
5	6.72	145.00	21.58×
6	17.12	19.28	1.13×
7	6.33	124.85	19.72×
8	6.53	126.35	19.35×
9	0.32	1.90	5.94×
10	8.58	264.14	30.79×

The runtime of the queries is reported in seconds, and was obtained by taking the median of 3 runs on my laptop using DuckDB 0.10.1. The `c6id.metal` instance results were obtained from the benchmark website. Despite being unable to *fit* all unique groups in my laptop's

memory, DuckDB can *compute* all unique groups and return them. The largest query, query 10, takes almost 4.5 minutes to complete. This is over 30× longer than with the beefy `c6id.metal` instance. The large difference is, of course, explained by the large differences in hardware. Interestingly, this is still faster than Spark on the `c6id.metal` instance, which takes 603.05 seconds!

Conclusion

DuckDB is constantly improving its larger-than-memory query processing capabilities. In this blog post, we showed some of the tricks DuckDB uses for spilling and loading data from storage. These tricks are implemented in DuckDB's external hash aggregation, released since 0.9.0. We took the hash aggregation for a spin on the H2O.ai benchmark, and DuckDB could complete all 50 GB queries on a laptop with only 16 GB of memory.

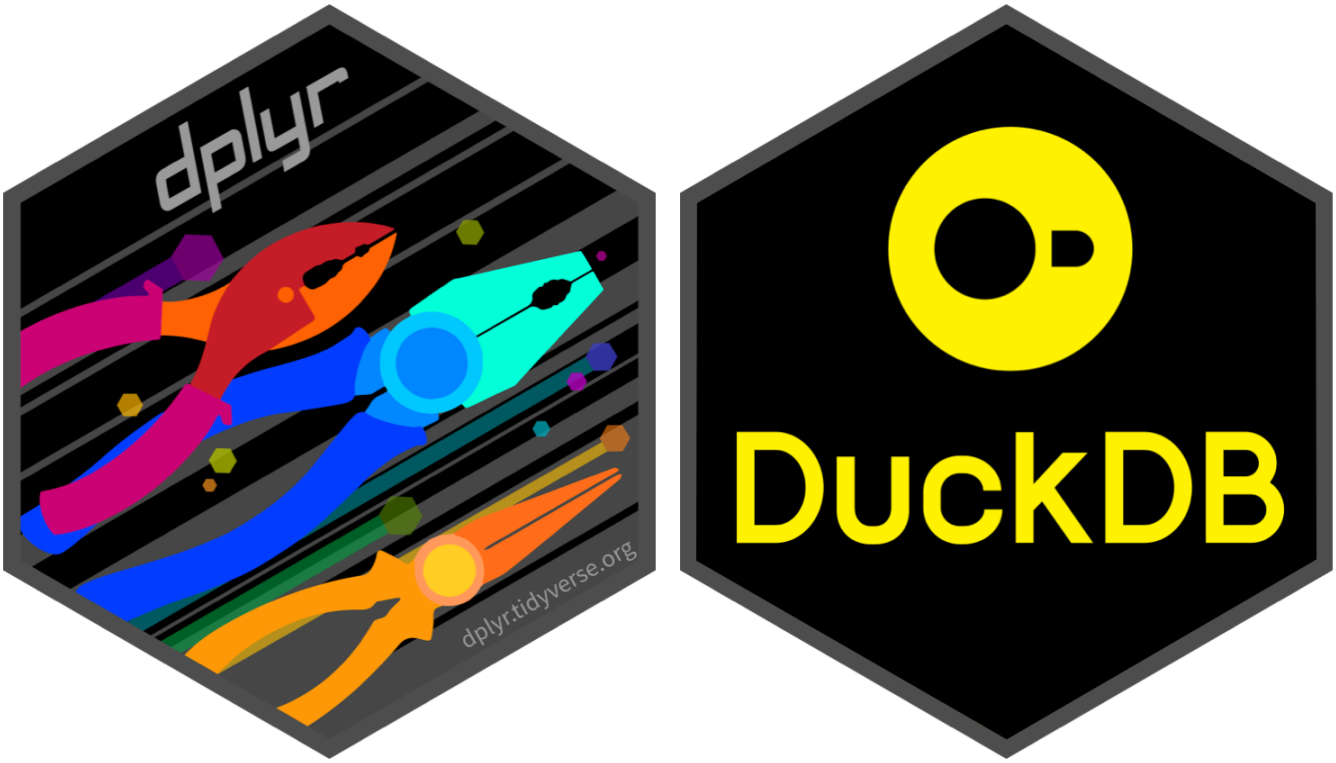
Interested in reading more? Read our paper on external aggregation.

duckplyr: dplyr Powered by DuckDB

Publication date: 2024-04-02

Author: Hannes Mühleisen

TL;DR: The new R package duckplyr translates the dplyr API to DuckDB's execution engine.



> For the duckplyr documentation, visit duckplyr.tidyverse.org.

Background

Wrangling tabular data into a form suitable for analysis can be a challenging task. Somehow, every data set is created differently. Differences between datasets exist in their logical organization of information into rows and columns or in more specific choices like the representation of dates, currency, categorical values, missing data and so on. The task is not simplified by the lack of global consensus on trivial issues like which character to use as a decimal separator. To gain new insights, we also commonly need to combine information from multiple sources, for example by joining two data sets using a common identifier. There are some common recurring operations however, that have been found to be universally useful in reshaping data for analysis. For example, the Structured (English) Query Language, or SQL ("See-Quel") for short describes a set of common operations that can be applied to tabular data like selection, projections, joins, aggregation, sorting, windowing, and more. SQL proved to be a huge success, despite its many warts and many attempts to replace it, it is still the de-facto language for data transformation with a gigantic industry behind it.

```
library("DBI")
con <- dbConnect(...)
df <- dbGetQuery(con, "SELECT something, very, complicated FROM some_table JOIN another_table BY (some_
shared_attribute) GROUP BY group_one, group_two ORDER BY some_column, and_another_column;")
```

A not very ergonomic way of pulling data into R

For data analysts in interactive programming environments like R or Python possibly from within IDEs such as RStudio or Jupyter Notebooks, using SQL to reshape data was never really a natural choice. Sure, sometimes it was required to use SQL to pull data from operational systems as shown above, but when given a choice, analysts much preferred to use the more ergonomic data reshaping facilities provided by those languages. R had built-in data wrangling from the start as part of the language with the data.frame class to represent tabular data. Later on, in 2014, Hadley Wickham defined the logical structure of tabular data for so-called "tidy" data and published the first version of the dplyr package designed to unify and simplify the previously unwieldy R commands to reshape data into a singular, unified and consistent API. In Python-land, the widely popular Pandas project extended Python with a de-facto tabular data representation along with relational-style operators albeit without any attempt at "tidiness".

At some point however, the R and Python data *processing* facilities started to creak under the ever-increasing weight of datasets that people wished to analyze. Datasets quickly grew into millions of rows. For example, one of the early datasets that required special handling was the American Community Survey dataset, because there are just so many Americans. But tools like Pandas and dplyr had been designed for convenience, not necessarily efficiency. For example, they lack the ability to parallelize data reshaping jobs on the now-common multicore processors.

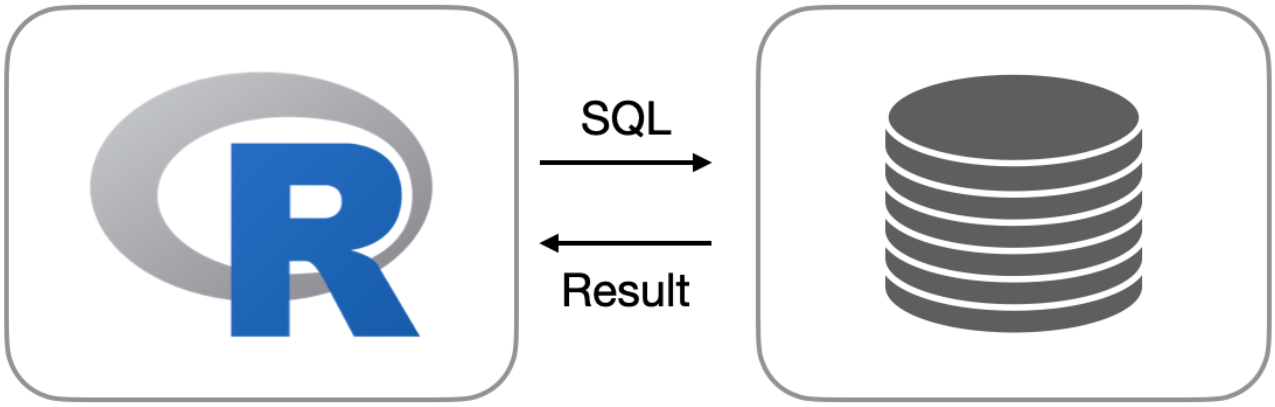
And while there was a whole set of emerging "Big Data" tools, using those from an interactive data analysis environment proved to be a poor developer experience, for example due to multi-second job startup times and very complex setup procedures far beyond the skill set of most data analysts. However, the world of relational data management systems had not stood still in the meantime. Great progress had been made to improve the efficiency of analytical data analysis from SQL: Innovations around columnar data representation, efficient query interpretation or even compilation, and automatic efficient parallelization increased query processing efficiency by several orders of magnitude. Regrettably, those innovations did not find their way into the data analysts toolkit – even as decades passed – due to lack of communication between communities and siloing of innovations into corporate, commercial, and close-source products.

There are two possible ways out of this unfortunate scenario:

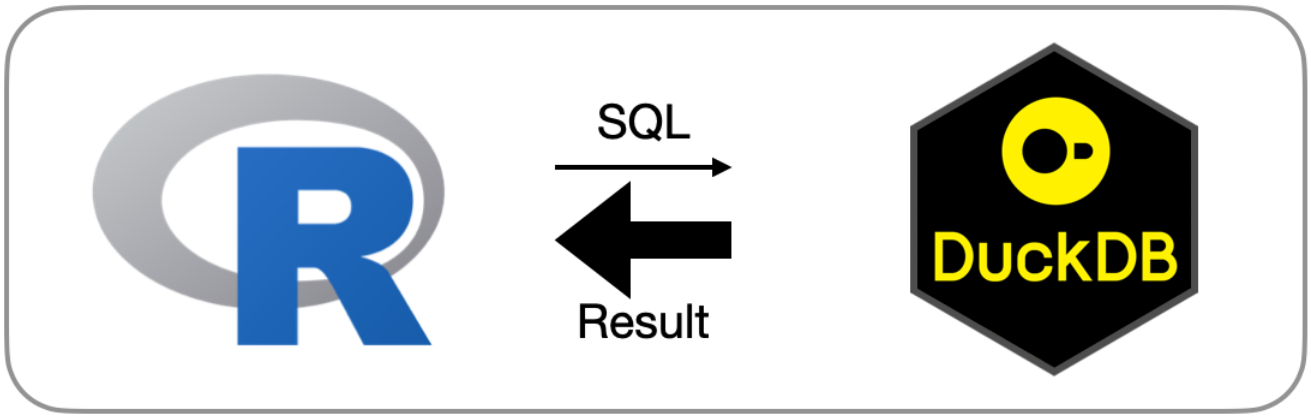
1. improve the data analysis capabilities of R and Python to be able to handle larger datasets through general efficiency improvements, optimization, and parallelization;
2. somehow integrate existing state-of-the-art technology into interactive data analysis environments.

The main issue with approach one is that building a *competitive* analytical query engine from scratch is a multi-million dollar effort requiring a team of highly specialized experts on query engine construction. There are many moving highly complex parts that all have to play together nicely. There are seemingly-obvious questions in query engines that one can get a PhD in data management systems for a solution. Recouping such a massive investment in a space where it is common that tools are built by volunteers in their spare time and released for free is challenging. That being said, there are a few commendable projects in this space like data.table or more recently pola.rs that offer greatly improved performance over older tools.

Approach two is also not without its challenges: State of the art query engine technology is often hidden behind incompatible architectures. For example, the two-tier architecture where a data management system runs on a dedicated database server and client applications use a client protocol to interact with said server is rather incompatible with interactive analysis. Setting up and maintaining a separate database "server" – even on the same computer – is still painful. Moving data back and forth between the analysis environment and the database server has been shown to be quite expensive. Unfortunately, those architectural decisions deeply influence the query engine trade-offs and are therefore difficult to change afterwards.



There has been movement in this space however: One of the stated goals of DuckDB is to unshackle state-of-the-art analytical data management technology from system architecture with its in-process architecture. Simply put, this means there is no separate database server and DuckDB instead runs within a "host" process. This host can be any application that requires data management capabilities or just an interactive data analysis environment like Python or R. Running within the host environment has another massive advantage: Moving data back and forth between the host and DuckDB is very cheap. For R and Python, DuckDB can directly run complex queries on data frames within the analysis environment without any import or conversion steps. Conversely, DuckDB's query results can directly be converted to data frames, greatly reducing the overhead of integrating with downstream libraries for plotting, further analysis or Machine Learning. DuckDB is able to efficiently execute arbitrarily complex relational queries including recursive and correlated queries. DuckDB is able to handle larger-than-memory datasets both in reading and writing but also when dealing with large intermediate results, for example resulting from aggregations with millions of groups. DuckDB has a sophisticated full query optimizer that removes the previously common manual optimization steps. DuckDB also offers persistence, tabular data being stored in files on disk. The tables in those files can be changed, too – while keeping transactional integrity. Those are unheard-of features in interactive data analysis, they are the result of decades of research and engineering in analytical data systems.



One issue remains however, DuckDB speaks SQL. While SQL is a popular language, not all analysts want to express their data transformations in SQL. One of the main issues here is that typically, queries are expressed as strings in R or Python scripts, which are sent to a database system in an opaque way. This means that those queries carry all-or-nothing semantics and it can be challenging to debug problems ("You have an error in your SQL syntax; check the manual…"). APIs like dplyr are often more convenient for the user, they allow an IDE to support things like auto-completion on functions, variable names etc. In addition, the additive nature of the dplyr API allows to build a sequence of data transformation in small steps, which reduces the cognitive load of the analyst considerably compared to writing a hundred-line SQL query. There have been some early experimental attempts to overload R's native data frame API in order to map to SQL databases, but those approaches have been found to be too limited in generality, surprising to users and generally too brittle. A better approach is needed.

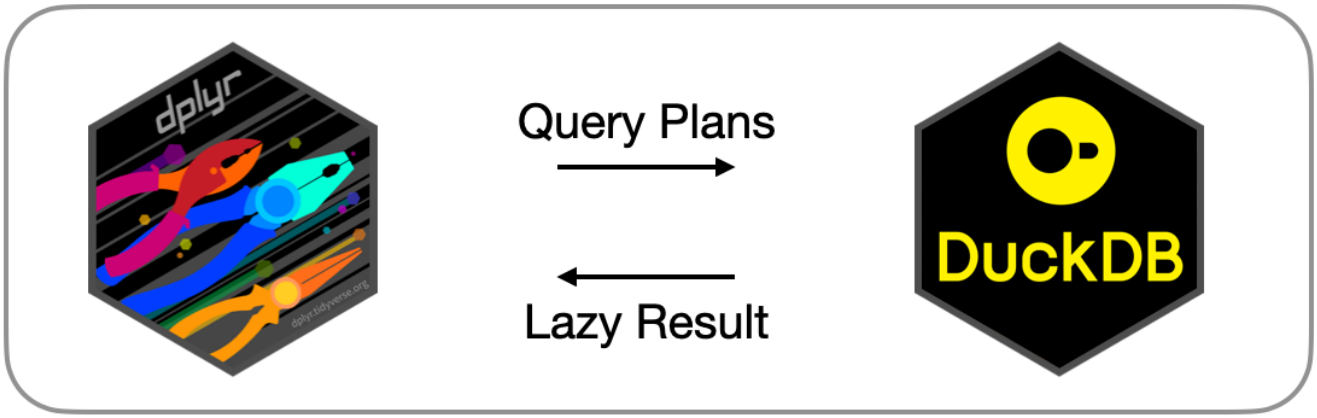
The duckplyr R Package

To address those issues, we have partnered up with the dplyr project team at Posit (formerly RStudio) and cynkra to develop **duckplyr**. duckplyr is a drop-in replacement for dplyr, powered by DuckDB for performance. Duckplyr implements several innovations in the interactive analysis space. First of all, installing duckplyr is just as easy as installing dplyr. DuckDB has been packaged for R as a stand-alone R package that contains the entire data management system code as well as wrappers for R. Both the DuckDB R package as well as duckplyr are available on CRAN, making installation on all major platforms a straightforward:

```
install.packages("duckplyr")
```

Verbs

Under the hood, duckplyr translates the sort-of-relational dplyr operations ("verbs") to DuckDB's relational query processing engine. Apart from some naming confusion, there is a mostly straightforward mapping between dplyr's verbs such as select, filter, summarise, etc. and DuckDB's project, filter and aggregate operators. A crucial difference from previous approaches is that duckplyr does not go through DuckDB's SQL interface to create query plans. Instead, duckplyr uses DuckDB's so-called "relational" API to directly construct logical query plans. This API allows to bypass the SQL parser entirely, greatly reducing the difficulty in operator, identifier, constant, and table name escaping that plagues other approaches such as dbplyr.



We have exposed the C++-level relational API to R, so that it is possible to directly construct DuckDB query plans from R. This low-level API is not meant to be used directly, but it is used by duckplyr to transform the dplyr verbs to the DuckDB relational API and thus to query plans. Here is an example:

```
library("duckplyr")
as_duckplyr_df(data.frame(n=1:10)) |>
    mutate(m=n+1) |>
    filter (m > 5) |>
    count() |>
    explain()
```

```
┌─────────────────────────┐
│        PROJECTION       │
│ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ │
│            n            │
└─────────────────────────┘
┌─────────────────────────┐
│    UNGROUPED_AGGREGATE  │
│ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ │
│      count_star()       │
└─────────────────────────┘
┌─────────────────────────┐
│          FILTER         │
│ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ │
│(+(CAST(n AS DOUBLE), 1.0)
```

```
          > 5.0)
  - - - - - - - - - - -
          EC: 10
┌─────────────────────────┐
└─────────────┬───────────┘
┌─────────────┴───────────┐
│     R_DATAFRAME_SCAN    │
  - - - - - - - - - - -
        data.frame
  - - - - - - - - - - -
            n
  - - - - - - - - - - -
          EC: 10
└─────────────────────────┘
```

We can see how a sequence of dplyr verbs mutate, filter, and count is "magically" transformed into a DuckDB query plan consisting of a scan, a filter, projections and an aggregate. We can see at the very bottom an `R_DATAFRAME_SCAN` operator is added. This operator directly reads an R data frame as if it were a table in DuckDB, without requiring actual data import. The new verb `explain()` causes DuckDB's logical query plan to be printed so that we can expect what DuckDB intends to execute based on the duckplyr sequence of verbs.

Expressions

An often overlooked yet crucial component of data transformations are so-called expressions. Expressions are (conceptually) scalar transformations of constants and columns from the data that can be used to for example produce derived columns or to transform actual column values to boolean values to be used in filters. For example, one might write an expression like `(amount - discount) * tax` to compute the actual invoiced amount without that amount actually being stored in a column or use an expression like `value > 42` in a filter expression to remove all rows where the value is less than or equal to `42`. Dplyr relies on the base R engine to evaluate expressions with some minor modifications to resolve variable names to columns in the input data. When moving evaluation of expressions over to DuckDB, the process becomes a little bit more involved. DuckDB has its own and independent expression system consisting of a built-in set of functions (e.g., `min`), scalar values and types. To transform R expressions into DuckDB expressions, we use an interesting R feature to capture un-evaluated abstract syntax trees from function arguments. By traversing the tree, we can transform R scalar values into DuckDB scalar values, R function calls into DuckDB function calls, and R-level variable references into DuckDB column references. It should be clear that this transformation cannot be perfect: There are functions in R that DuckDB simply does not support, for example those coming from the myriad of contributed packages. While we are working on expanding the set of supported expressions, there will always be some that cannot be translated. However, in the case of non-translatable expressions, we would still be able to return a result to the user. To achieve this, we have implemented a transparent fall-back mechanism that uses the existing R-level expression evaluation method in the case that an expression cannot be translated to DuckDB's expression language. For example, the following transformation `m = n + 1` can be translated:

```
as_duckplyr_df(data.frame(n=1:10)) |>
    mutate(m=n+1) |>
    explain()
```

```
┌─────────────────────────┐
│       PROJECTION        │
  - - - - - - - - - - -
            n
            m
└─────────────┬───────────┘
┌─────────────┴───────────┐
│     R_DATAFRAME_SCAN    │
  - - - - - - - - - - -
        data.frame
  - - - - - - - - - - -
            n
  - - - - - - - - - - -
          EC: 10
└─────────────────────────┘
```

While the following transformation using an inline lambda function cannot (yet):

```
as_duckplyr_df(data.frame(n=1:10)) |>
    mutate(m=(\(x) x+1)(n)) |>
    explain()
```

```
┌─────────────────────────┐
│      R_DATAFRAME_SCAN   │
│ — — — — — — — — — — — — │
│        data.frame       │
│ — — — — — — — — — — — — │
│                n        │
│                m        │
│ — — — — — — — — — — — — │
│           EC: 10        │
└─────────────────────────┘
```

It is a little hard to see (and we are working on improving this), the `explain()` output clearly differs between the two mutate expressions. In the first case, DuckDB computes the `+ 1` as part of the projection operator, in the second case, the translation failed and a fallback was used, leading to the computation happening in the R engine. The upside of automatic fallback is that things "just work". The downside is that there will usually be a performance hit from the fallback due to – for example – the lack of automatic parallelization. We are planning to add a debug mode where users can inspect the translation process and get insight into why translations fail.

Eager vs. Lazy Materialization

Dplyr and Pandas follow an execution strategy known as "eager materialization". Every time an operation is invoked on a data frame, this operation is immediately executed and the result created in memory. This can be problematic. Consider the following example, a ten million row dataset is modified by adding 1 to a column. Then, the `top_n` operation is invoked to retrieve the first ten rows only. Because of eager materialization, the addition operation is executed on ten million rows, the result is created in memory, only for almost all of it to be thrown away immediately because only the first ten rows were requested. Duckplyr solves this problem by using a so-called "lazy materialization" strategy where no action is performed initially but instead the users' intent is being captured. This means that the addition of one to ten million rows will not be performed immediately. The system is instead able to optimize the requested computation and will only perform the addition on the first few rows. Also importantly, the intermediate result of the addition is never actually created in memory, greatly reducing the memory pressure.

However, lazy computation presents a possible integration issue: The result of lazy computation has to be some sort of lazy computation placeholder object, that can be passed to another lazy operation or forced to be evaluated, e.g., via a special print method. However, this would break backwards compatibility with dplyr, where the result of each dplyr operation is a fully materialized data frame itself. This means that those results can be directly passed on to downstream operations like plotting without the plotting package having to be aware of the "lazyness" of the duckplyr result object. To address this, we have creatively used a R feature known as ALTREP. ALTREP allows R objects to have different in-memory representations, and for custom code to be executed whenever those objects are accessed. Duckplyr results are lazy placeholder objects, yes, but they appear to be bog-standard R data frames at the same time. R data frames are essentially named lists of typed vectors with a special row.names attribute. Because DuckDB's lazy query planning already knows the names and types of the resulting table, we can export the names into the lazy data frame. We do not however know the number of rows nor their contents yet. We therefore make both the actual data vectors and the row names vector that contains the data frame length lazy vectors. Those vectors carry a callback that the R engine will invoke whenever downstream code – e.g., plotting code – touches those vectors. The callback will actually trigger computation of the entire pipeline and transformation of the result to a R data frame. Duckplyr's own operations will refrain from touching those vectors, they instead continue lazily using a special lazy computation object that is also stored in the lazy data frame. This method allows duckplyr to be both lazy and not at the same time, which allows full drop-in replacement with the eagerly evaluated dplyr while keeping the lazy evaluation that is crucial for DuckDB to be able to do a full-query optimization of the various transformation steps.

Here is an example of the duality of the result of duckplyr operations using R's `inspect()` method:

```
dd <- as_duckplyr_df(data.frame(n=1:10)) |> mutate(m=n+1)
.Internal(inspect(dd))
```

```
@12daad988 19 VECSXP g0c2 [OBJ,REF(2),ATT] (len=2, tl=0)
  @13e0c9d60 13 INTSXP g0c0 [REF(4)] DUCKDB_ALTREP_REL_VECTOR n (INTEGER)
  @13e0ca1c0 14 REALSXP g0c0 [REF(4)] DUCKDB_ALTREP_REL_VECTOR m (DOUBLE)
```

```
ATTRIB:
  @12817a838 02 LISTSXP g0c0 [REF(1)]
    TAG: @13d80d420 01 SYMSXP g1c0 [MARK,REF(65535),LCK,gp=0x4000] "names" (has value)
    @12daada08 16 STRSXP g0c2 [REF(65535)] (len=2, tl=0)
      @13d852ef0 09 CHARSXP g1c1 [MARK,REF(553),gp=0x61] [ASCII] [cached] "n"
      @13e086338 09 CHARSXP g1c1 [MARK,REF(150),gp=0x61] [ASCII] [cached] "m"
    TAG: @13d80d9d0 01 SYMSXP g1c0 [MARK,REF(56009),LCK,gp=0x4000] "class" (has value)
    @12da9e208 16 STRSXP g0c2 [REF(65535)] (len=2, tl=0)
      @11ff15708 09 CHARSXP g0c2 [MARK,REF(423),gp=0x60] [ASCII] [cached] "duckplyr_df"
      @13d892308 09 CHARSXP g1c2 [MARK,REF(1513),gp=0x61,ATT] [ASCII] [cached] "data.frame"
    TAG: @13d80d1f0 01 SYMSXP g1c0 [MARK,REF(65535),LCK,gp=0x4000] "row.names" (has value)
    @13e0c9970 13 INTSXP g0c0 [REF(65535)] DUCKDB_ALTREP_REL_ROWNAMES
```

We can see that the internal structure of the data frame indeed reflects a data frame, but we can also see the special vectors DUCKDB_ALTREP_REL_VECTOR that hide the un-evaluated data vectors as well as DUCKDB_ALTREP_REL_ROWNAMES that hide the fact that the true dimensions of the data frame are not yet known.

Benchmark: TPC-H Q1

Let's finish with a quick demonstration of duckplyr's performance improvements. We use the data generator from the well known TPC-H benchmark, which is helpfully available as a DuckDB extension. With the "scale factor" of 1, the following DuckDB/R one-liner will generate a data set with a little over 6 million rows and store it in the R data frame named "lineitem":

```
lineitem <- duckdb:::sql("INSTALL tpch; LOAD tpch; CALL dbgen(sf=1); FROM lineitem;")
```

We have transformed the TPC-H benchmark query 1 from its original SQL formulation to dplyr syntax:

```
tpch_01 <- function() {
  lineitem |>
    select(l_shipdate, l_returnflag, l_linestatus, l_quantity, l_extendedprice, l_discount, l_tax) |>
    filter(l_shipdate <= !!as.Date("1998-09-02")) |>
    select(l_returnflag, l_linestatus, l_quantity, l_extendedprice, l_discount, l_tax) |>
    summarise(
      sum_qty = sum(l_quantity),
      sum_base_price = sum(l_extendedprice),
      sum_disc_price = sum(l_extendedprice * (1 - l_discount)),
      sum_charge = sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),
      avg_qty = mean(l_quantity),
      avg_price = mean(l_extendedprice),
      avg_disc = mean(l_discount),
      count_order = n(),
      .by = c(l_returnflag, l_linestatus)
    ) |>
    arrange(l_returnflag, l_linestatus)
}
```

We can now execute this function with both dplyr and duckplyr and observe the time required to compute the result. "Stock" dplyr takes ca. 400 milliseconds on my MacBook for this query, duckplyr requires only ca 70 milliseconds. Again, this time includes all the magic transforming the sequence of dplyr verbs into a relational operator tree, optimizing said tree, converting the input R data frame into a DuckDB intermediate on-the-fly, and transforming the (admittedly small) result back to a R data frame. Of course, the data set used here is still relatively small and the query is not that complex either, essentially a single grouped aggregation. The differences will be much more pronounced for more complex transformations on larger data sets. duckplyr can also directly access large collections of e.g., Parquet files on storage, and push down filters into those scans, which can also greatly improve performance.

Conclusion

The duckplyr package for R wraps DuckDB's state-of-the-art analytical query processing techniques in a dplyr-compatible API. We have gone to great lengths to ensure compatibility despite switching execution paradigms from eager to lazy and having to translate expressions to a different environment. We continue to work to expand duckplyr's capabilities but would love to hear your experiences trying it out.

Here are two recordings from last year's posit::conf where we present DuckDB for R and duckplyr:

- In-Process Analytical Data Management with DuckDB – posit::conf(2023)
- duckplyr: Tight Integration of duckdb with R and the tidyverse – posit::conf(2023)

Vector Similarity Search in DuckDB

Publication date: 2024-05-03

Author: Max Gabrielsson

TL;DR: This blog post shows a preview of DuckDB's new `vss` extension, which introduces support for HNSW (Hierarchical Navigable Small Worlds) indexes to accelerate vector similarity search.

In DuckDB v0.10.0, we introduced the ARRAY data type, which stores fixed-sized lists, to complement the existing variable-size `LIST` data type.

The initial motivation for adding this data type was to provide optimized operations for lists that can utilize the positional semantics of their child elements and avoid branching as all lists have the same length. Think e.g., the sort of array manipulations you'd do in NumPy: stacking, shifting, multiplying – you name it. Additionally, we wanted to improve our interoperability with Apache Arrow, as previously Arrow's fixed-size list types would be converted to regular variable-size lists when ingested into DuckDB, losing some type information.

However, as the hype for **vector embeddings** and **semantic similarity search** was growing, we also snuck in a couple of distance metric functions for this new ARRAY type: `array_distance`, `array_negative_inner_product` and `array_cosine_distance`

> If you're one of today's lucky 10,000 and haven't heard of word embeddings or vector search, the short version is that it's a technique used to represent documents, images, entities – *data* as high-dimensional *vectors* and then search for *similar* vectors in a vector space, using some sort of mathematical "distance" expression to measure similarity. This is used in a wide range of applications, from natural language processing to recommendation systems and image recognition, and has recently seen a surge in popularity due to the advent of generative AI and availability of pre-trained models.

This got the community really excited! While we (DuckLabs) initially went on record saying that we would not be adding a vector similarity search index to DuckDB as we deemed it to be too far out of scope, we were very interested in supporting custom indexes through extensions in general. Shoot, I've been *personally* nagging on about wanting to plug-in an "R-Tree" index since the inception of DuckDBs spatial extension! So when one of our client projects evolved into creating a proof-of-concept custom "HNSW" index extension, we said that we'd give it a shot. And... well, one thing led to another.

Fast forward to now and we're happy to announce the availability of the `vss` vector similarity search extension for DuckDB! While some may say we're late to the vector search party, we'd like to think the party is just getting started!

Alright, so what's in `vss`?

The Vector Similarity Search (VSS) Extension

On the surface, `vss` seems like a comparatively small DuckDB extension. It does not provide any new data types, scalar functions or copy functions, but rather a single new index type: `HNSW` (Hierarchical Navigable Small Worlds), which is a graph-based index structure that is particularly well-suited for high-dimensional vector similarity search.

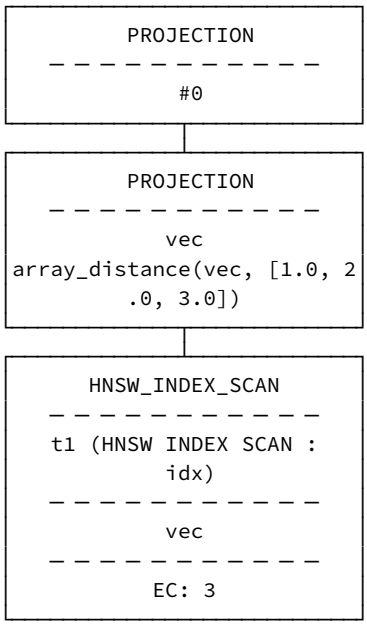
```
-- Create a table with an array column
CREATE TABLE embeddings (vec FLOAT[3]);

-- Create an HNSW index on the column
CREATE INDEX idx ON embeddings USING HNSW (vec);
```

This index type can't be used to enforce constraints or uniqueness like the built-in ART index, and can't be used to speed up joins or index regular columns either. Instead, the `HNSW` index is only applicable to columns of the `ARRAY` type containing `FLOAT` elements and will only be used to accelerate queries calculating the "distance" between a constant `FLOAT ARRAY` and the `FLOAT ARRAY`'s in the indexed column, ordered by the resulting distance and returning the top-n results. That is, queries of the form:

```
SELECT *
FROM embeddings
ORDER BY array_distance(vec, [1, 2, 3]::FLOAT[3])
LIMIT 3;
```

will have their logical plan optimized to become a projection over a new HNSW index scan operator, removing the limit and sort altogether. We can verify this by checking the `EXPLAIN` output:

```
EXPLAIN
SELECT *
FROM embeddings
ORDER BY array_distance(vec, [1, 2, 3]::FLOAT[3])
LIMIT 3;
```



You can pass the HNSW index creation statement a `metric` parameter to decide what kind of distance metric to use. The supported metrics are `l2sq`, `cosine` and `inner_product`, matching the three built-in distance functions: `array_distance`, `array_cosine_distance` and `array_negative_inner_product`. The default is `l2sq`, which uses Euclidean distance (`array_distance`):

```
CREATE INDEX l2sq_idx ON embeddings USING HNSW (vec)
WITH (metric = 'l2sq');
```

To use cosine distance (`array_cosine_distance`):

```
CREATE INDEX cos_idx ON embeddings USING HNSW (vec)
WITH (metric = 'cosine');
```

To use inner product (`array_negative_inner_product`):

```
CREATE INDEX ip_idx ON embeddings USING HNSW (vec)
WITH (metric = 'ip');
```

Implementation

The `vss` extension is based on the `usearch` library, which provides a flexible C++ implementation of the HNSW index data structure boasting very impressive performance benchmarks. While we currently only use a subset of all the functionality and tuning options provided by `usearch`, we're excited to explore how we can leverage more of its features in the future. So far we're mostly happy that it aligns so nicely with DuckDB's development ethos. Much like DuckDB itself, `usearch` is written in portable C++11 with no external dependencies and released under a permissive license, making it super smooth to integrate into our extension build and distribution pipeline.

Limitations

The big limitation as of now is that the HNSW index can only be created in in-memory databases, unless the `SET hnsw_enable_experimental_persistence = <bool>`{:.language-sql .highlight} configuration parameter is set to `true`. If this parameter is not set, any attempt to create an HNSW index in a disk-backed database will result in an error message, but if the parameter is set, the index will not only be created in memory, but also persisted to disk as part of the DuckDB database file during checkpointing. After restarting or loading a database file with a persisted HNSW index, the index will be lazily loaded back into memory whenever the associated table is first accessed, which is significantly faster than having to re-create the index from scratch.

The reasoning for locking this feature behind an experimental flag is that we still have some known issues related to persistence of custom indexes that we want to address before enabling it by default. In particular, WAL recovery is not yet properly implemented for custom indexes, meaning that if a crash occurs or the database is shut down unexpectedly while there are uncommitted changes to a HNSW-indexed table, you can end up with data loss or corruption of the index. While it is technically possible to recover from a unexpected shutdown manually by first starting DuckDB separately, loading the `vss` extension and then ATTACHing the database file, which ensures that the HNSW index functionality is available during WAL-playback, you should not rely on this for production workloads.

We're actively working on addressing this and other issues related to index persistence, which will hopefully make it into DuckDB v0.10.3, but for now we recommend using the HNSW index in in-memory databases only.

At runtime however, much like the ART the HNSW index must be able to fit into RAM in its entirety, and the memory allocated by the HNSW at runtime is allocated "outside" of the DuckDB memory management system, meaning that it won't respect DuckDB's `memory_limit` configuration parameter.

Another current limitation with the HNSW index so far is that it only supports the `FLOAT` (a 32-bit, single-precision floating point) type for the array elements and only distance metrics corresponding to the three built in distance functions, `array_distance`, `array_negative_inner_product` and `array_cosine_distance`. But this is also something we're looking to expand upon in the near future as it is much less of a technical limitation and more of a "we haven't gotten around to it yet" limitation.

Conclusion

The `vss` extension for DuckDB is a new extension that adds support for creating HNSW indexes on fixed-size list columns in DuckDB, accelerating vector similarity search queries. The extension can currently be installed on DuckDB v0.10.2 on all supported platforms (including Wasm!) by running `INSTALL vss; LOAD vss`. The `vss` extension treads new ground for DuckDB extensions by providing a custom index type and we're excited to refine and expand on this functionality going forward.

While we're still working on addressing some of the limitations above, particularly those related to persistence (and performance), we still really want to share this early version the `vss` extension as we believe this will open up a lot of cool opportunities for the community. So make sure to check out the `vss` extension documentation for more information on how to work with this extension!

This work was made possible by the sponsorship of a DuckLabs customer! If you are interested in similar work for specific capabilities, please reach out to DuckLabs. Alternatively, we're happy to welcome contributors! Please reach out to the DuckLabs team over on Discord or on the `vss` extension GitHub repository to keep up with the latest developments.

Access 150k+ Datasets from Hugging Face with DuckDB

Publication date: 2024-05-29

Author: The Hugging Face and DuckDB teams

TL;DR: DuckDB can now read data from Hugging Face via the `hf://` prefix.

We are excited to announce that we added support for `hf://` paths in DuckDB, providing access to more than 150,000 datasets for artificial intelligence. We worked with Hugging Face to democratize the access, manipulation, and exploration of datasets used to train and evaluate AI models.

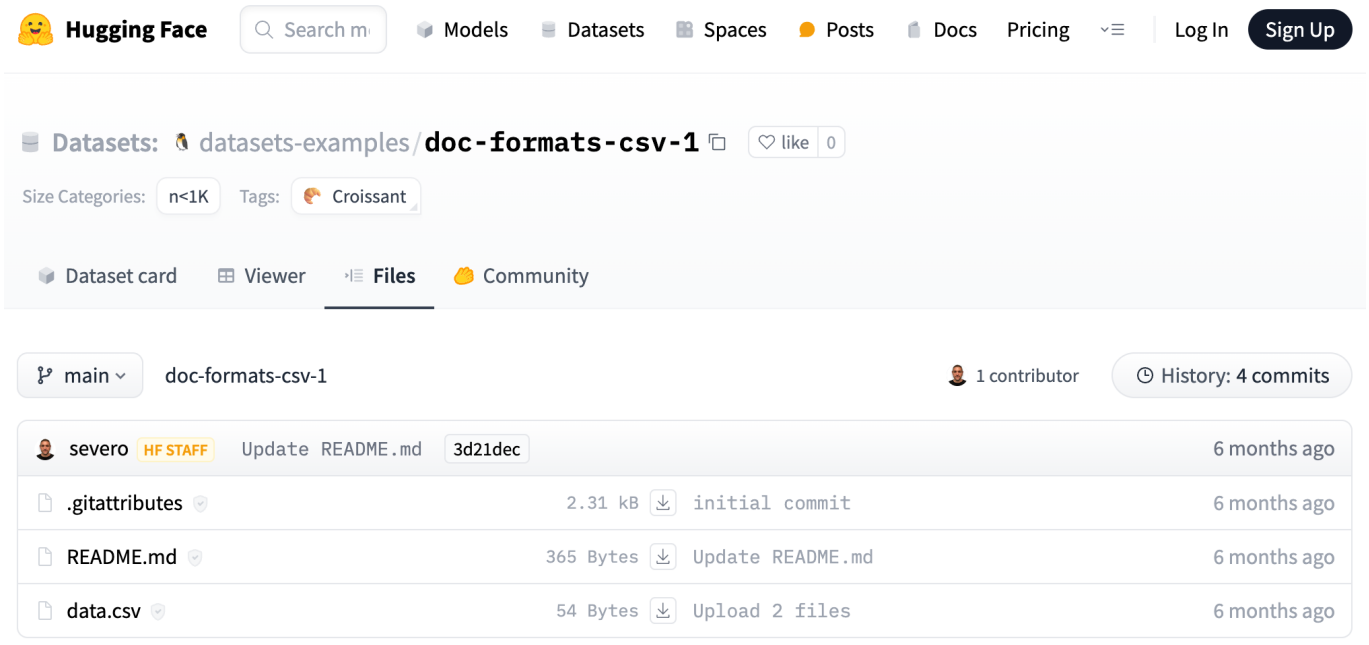
Dataset Repositories

Hugging Face is a popular central platform where users can store, share, and collaborate on machine learning models, datasets, and other resources.

A dataset typically includes the following content:

- A README file: This plain text file provides an overview of the repository and its contents. It often describes the purpose, usage, and specific requirements or dependencies.
- Data files: Depending on the type of repository, it can include data files like CSV, Parquet, JSONL, etc. These are the core components of the repository.

A typical repository looks like this:



Read Using `hf://` Paths

You often need to read files in various formats (such as CSV, JSONL, and Parquet) when working with data. As of version v0.10.3, DuckDB has native support for `hf://` paths as part of the `httpfs` extension, allowing easy access to all these formats.

Now, it is possible to query them using the URL pattern below:

`hf://datasets/<my_username>/<my_dataset>/<path_to_file>`

For example, to read a CSV file, you can use the following query:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

Where:

- `datasets-examples` is the name of the user/organization
- `doc-formats-csv-1` is the name of the dataset repository
- `data.csv` is the file path in the repository

The result of the query is:

kind	sound
dog	woof
cat	meow
pokemon	pika
human	hello

To read a JSONL file, you can run:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-jsonl-1/data.jsonl';
```

Finally, for reading a Parquet file, use the following query:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-parquet-1/data/train-00000-of-00001.parquet';
```

Each of these commands reads the data from the specified file format and displays it in a structured tabular format. Choose the appropriate command based on the file format you are working with.

Creating a Local Table

To avoid accessing the remote endpoint for every query, you can save the data in a DuckDB table by running a `CREATE TABLE ... AS` command. For example:

```
CREATE TABLE data AS
    SELECT *
    FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

Then, simply query the `data` table as follows:

```
SELECT *
FROM data;
```

Multiple Files

You might need to query multiple files simultaneously when working with large datasets. Let's see a quick sample using the cais/mmlu (Measuring Massive Multitask Language Understanding) dataset. This dataset captures a test consisting of multiple-choice questions from various branches of knowledge. It covers 57 tasks, including elementary mathematics, US history, computer science, law, and more. To attain high accuracy on this test, AI models must possess extensive world knowledge and problem-solving ability.

First, let's count the number of rows in individual files. To get the row count from a single file in the cais/mmlu dataset, use the following query:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/dev-00000-of-00001.parquet';
```

count
5

Similarly, for another file (`test-00000-of-00001.parquet`) in the same dataset, we can run:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/test-00000-of-00001.parquet';
```

count
152

To query all files under a specific format, you can use a glob pattern. Here's how you can count the rows in all files that match the pattern `*.parquet`:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet';
```

count
173

By using glob patterns, you can efficiently handle large datasets and perform comprehensive queries across multiple files, simplifying your data inspections and processing tasks. Here, you can see how you can look for questions that contain the word "planet" in astronomy:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet'
WHERE question LIKE '%planet%';
```

count
21

And see some examples:

```
SELECT question
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet'
WHERE question LIKE '%planet%'
LIMIT 3;
```

question
Why isn't there a planet where the asteroid belt is located?
On which planet in our solar system can you find the Great Red Spot?
The lithosphere of a planet is the layer that consists of

Versioning and Revisions

In Hugging Face repositories, dataset versions or revisions are different dataset updates. Each version is a snapshot at a specific time, allowing you to track changes and improvements. In git terms, it can be understood as a branch or specific commit.

You can query different dataset versions/revisions by using the following URL:

```
hf://datasets/<my_username>/<my_dataset>@<my_branch>/<path_to_file>
```

For example:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1@~parquet/**/*.parquet';
```

kind	sound
dog	woof
cat	meow
pokemon	pika
human	hello

The previous query will read all Parquet files under the `~parquet` revision. This is a special branch where Hugging Face automatically generates the Parquet files of every dataset to enable efficient scanning.

Authentication

Configure your Hugging Face Token in the DuckDB Secrets Manager to access private or gated datasets. First, visit Hugging Face Settings – Tokens to obtain your access token. Second, set it in your DuckDB session using DuckDB's Secrets Manager. DuckDB supports two providers for managing secrets:

- `CONFIG`: The user must pass all configuration information into the `CREATE SECRET` statement. To create a secret using the `CONFIG` provider, use the following command:

  ```
  CREATE SECRET hf_token (
     TYPE huggingface,
     TOKEN 'your_hf_token'
  );
  ```

- `credential_chain`: Automatically tries to fetch credentials. For the Hugging Face token, it will try to get it from `~/.cache/huggingface/token`. To create a secret using the `credential_chain` provider, use the following command:

  ```
  CREATE SECRET hf_token (
     TYPE huggingface,
     PROVIDER credential_chain
  );
  ```

Conclusion

The integration of `hf://` paths in DuckDB significantly streamlines accessing and querying over 150,000 datasets available on Hugging Face. This feature democratizes data manipulation and exploration, making it easier for users to interact with various file formats such as CSV, JSON, JSONL, and Parquet. By utilizing `hf://` paths, users can execute complex queries, efficiently handle large datasets, and harness the extensive resources of Hugging Face repositories.

The integration supports seamless access to individual files, multiple files using glob patterns, and different dataset versions. DuckDB's robust capabilities ensure a flexible and streamlined data processing experience. This integration is a significant leap forward in making

AI dataset access more accessible and efficient for researchers and developers, fostering innovation and accelerating progress in machine learning.

Want to learn more about leveraging DuckDB with Hugging Face datasets? Explore the detailed guide.

Analyzing Railway Traffic in the Netherlands

Publication date: 2024-05-31

Author: Gábor Szárnyas

TL;DR: We use a real-world railway dataset to demonstrate some of DuckDB's key features, including querying different file formats, connecting to remote endpoints, and using advanced SQL features.

Introduction

The Netherlands, the birthplace of DuckDB, has an area of about 42,000 km^2 with a population of about 18 million people. The high density of the country is a key factor in its extensive railway network, which consists of 3,223 km of tracks and 397 stations.

Information about this network's stations and services is available in the form of open datasets. These high-quality datasets are maintained by the team behind the Rijden de Treinen *(Are the trains running?)* application.

In this post, we'll demonstrate some of DuckDB's analytical capabilities on the Dutch railway network dataset. Unlike most of our other blog posts, this one doesn't introduce a new feature or release: instead, it demonstrates several existing features using a single domain. Some of the queries explained in this blog post are shown in simplified form on DuckDB's landing page.

Loading the Data

For our initial queries, we'll use the 2023 railway services dataset. To get this dataset, download the `services-2023.csv.gz` file (330 MB) and load it into DuckDB.

First, start the DuckDB command line client on a persistent database:

```
duckdb railway.db
```

Then, load the `services-2023.csv.gz` file into the `services` table.

```
CREATE TABLE services AS
    FROM 'services-2023.csv.gz';
```

Despite the seemingly simple query, there is quite a lot going on here. Let's deconstruct the query:

- First, there is no need to explicitly define a schema for our `services` table, nor is it necessary to use a `COPY ... FROM` statement. DuckDB automatically detects that the `'services-2023.csv.gz'` refers to a gzip-compressed CSV file, so it calls the `read_csv` function, which decompresses the file and infers its schema from its content using the CSV sniffer.
- Second, the query makes use of DuckDB's FROM-first syntax, which allows users to omit the `SELECT *` clause. Hence, the SQL statement `FROM 'services-2023.csv.gz';` is a shorthand for `SELECT * FROM 'services-2023.csv.gz';`.
- Third, the query creates a table called `services` and populates it with the result from the CSV reader. This is achieved using a `CREATE TABLE ... AS` statement.

Using DuckDB v0.10.3, loading the dataset takes approximately 5 seconds on an M2 MacBook Pro. To check the amount of data loaded, we can run the following query which pretty-prints the number of rows in the `services` table:

```
SELECT format('{:,}', count(*)) AS num_services
FROM services;
```

num_services
21,239,393

We can see that more than 21 million train services ran in the Netherlands in 2023.

Finding the Busiest Station per Month

Let's ask a simple query first: *What were the busiest railway stations in the Netherlands in the first 6 months of 2023?*

First, for every month, let's compute the number of services passing through each station. To do so, we extract the month from the service's date using the `month` function, then perform a group-by aggregation with a `count(*)`:

```
SELECT
    month("Service:Date") AS month,
    "Stop:Station name" AS station,
    count(*) AS num_services
FROM services
GROUP BY month, station
LIMIT 5;
```

Note that this query showcases a common redundancy in SQL: we list the names of non-aggregated columns in both the `SELECT` and the `GROUP BY` clauses. Using DuckDB's `GROUP BY ALL` feature, we can eliminate this. At the same time, let's also turn this result into an intermediate table called `services_per_month` using a `CREATE TABLE ... AS` statement:

```
CREATE TABLE services_per_month AS
    SELECT
        month("Service:Date") AS month,
        "Stop:Station name" AS station,
        count(*) AS num_services
    FROM services
    GROUP BY ALL;
```

To answer the question, we can use the `arg_max(arg, val)` aggregation function, which returns the column `arg` in the row with the maximum value `val`. We filter on the month and return the results:

```
SELECT
    month,
    arg_max(station, num_services) AS station,
    max(num_services) AS num_services
FROM services_per_month
WHERE month <= 6
GROUP BY ALL;
```

month	station	num_services
1	Utrecht Centraal	34760
2	Utrecht Centraal	32300
3	Utrecht Centraal	37386
4	Amsterdam Centraal	33426
5	Utrecht Centraal	35383
6	Utrecht Centraal	35632

Maybe surprisingly, in most months, the busiest railway station is not in Amsterdam but in the country's 4th largest city, Utrecht, thanks to its central geographic location.

Finding the Top-3 Busiest Stations for Each Summer Month

Let's change the question to: *Which are the top-3 busiest stations for each summer month?* The `arg_max()` function only helps us find the top-1 value but it is not sufficient for finding top-k results.

Using a Window Function (OVER)

DuckDB has extensive support for SQL features, including window functions and we can use the `rank()` function to find top-k values. Additionally, we use `make_date` to reconstruct the date, `strftime` to turn it into the month's name and `array_agg`:

```
SELECT month, month_name, array_agg(station) AS top3_stations
FROM (
    SELECT
        month,
        strftime(make_date(2023, month, 1), '%B') AS month_name,
        rank() OVER
            (PARTITION BY month ORDER BY num_services DESC) AS rank,
        station,
        num_services
    FROM services_per_month
    WHERE month BETWEEN 6 AND 8
)
WHERE rank <= 3
GROUP BY ALL
ORDER BY month;
```

This gives the following result:

month	month_name	top3_stations
6	June	[Utrecht Centraal, Amsterdam Centraal, Schiphol Airport]
7	July	[Utrecht Centraal, Amsterdam Centraal, Schiphol Airport]
8	August	[Utrecht Centraal, Amsterdam Centraal, Amsterdam Sloterdijk]

We can see that the top 3 spots are shared between four stations: Utrecht Centraal, Amsterdam Centraal, Schiphol Airport, and Amsterdam Sloterdijk.

Using the `max_by(arg, val, n)` Function

Starting with DuckDB version 1.1.0, you can use a variant of the `max_by` function that accepts a third parameter, n, for the number of rows. The resulting code is more concise and faster than the one using a window function.

```
SELECT
    month,
    strftime(make_date(2023, month, 1), '%B') AS month_name,
    max_by(station, num_services, 3) AS stations,
FROM services_per_month
WHERE month BETWEEN 6 AND 8
GROUP BY ALL
ORDER BY month;
```

Directly Querying Parquet Files through HTTPS or S3

DuckDB supports querying remote files, including CSV and Parquet, via the HTTP(S) protocol and the S3 API. For example, we can run the following query:

```
SELECT "Service:Date", "Stop:Station name"
FROM 'https://blobs.duckdb.org/nl-railway/services-2023.parquet'
LIMIT 3;
```

It returns the following result:

Service:Date	Stop:Station name
2023-01-01	Rotterdam Centraal
2023-01-01	Delft
2023-01-01	Den Haag HS

Using the remote Parquet file, the query for answering *Which are the top-3 busiest stations for each summer month?* can be run directly on a remote Parquet file without creating any local tables. To do this, we can define the `services_per_month` table as a common table expression in the `WITH` clause. The rest of the query remains the same:

```
WITH services_per_month AS (
    SELECT
        month("Service:Date") AS month,
        "Stop:Station name" AS station,
        count(*) AS num_services
    FROM 'https://blobs.duckdb.org/nl-railway/services-2023.parquet'
    GROUP BY ALL
)
SELECT month, month_name, array_agg(station) AS top3_stations
FROM (
    SELECT
        month,
        strftime(make_date(2023, month, 1), '%B') AS month_name,
        rank() OVER
            (PARTITION BY month ORDER BY num_services DESC) AS rank,
        station,
        num_services
    FROM services_per_month
    WHERE month BETWEEN 6 AND 8
)
WHERE rank <= 3
GROUP BY ALL
ORDER BY month;
```

This query yields the same result as the query above, and completes (depending on the network speed) in about 1–2 seconds. This speed is possible because DuckDB doesn't need to download the whole Parquet file to evaluate the query: while the file size is 309 MB, it only uses about 20 MB of network traffic, approximately 6% of the total file size.

The reduction in network traffic is possible because of partial reading along both the columns and the rows of the data. First, Parquet's columnar layout allows the reader to only access the required columns. Second, the zonemaps available in the Parquet file's metadata allow the filter pushdown optimization (e.g., the reader only fetches row groups with dates in the summer months). Both of these optimizations are implemented via HTTP range requests, saving considerable traffic and time when running queries on remote Parquet files.

Largest Distance between Train Stations in the Netherlands

Let's answer the following question: *Which two train stations in the Netherlands have the largest distance between them when traveling via rail?* For this, we'll use two datasets. The first, `stations-2022-01.csv`, contains information on the railway stations (station name, country, etc.). We can simply load and query this dataset as follows:

```
CREATE TABLE stations AS
    FROM 'https://blobs.duckdb.org/data/stations-2022-01.csv';
```

```
SELECT
    id,
    name_short,
    name_long,
    country,
    printf('%.2f', geo_lat) AS latitude,
    printf('%.2f', geo_lng) AS longitude
FROM stations
LIMIT 5;
```

id	name_short	name_long	country	latitude	longitude
266	Den Bosch	's-Hertogenbosch	NL	51.69	5.29
269	Dn Bosch O	's-Hertogenbosch Oost	NL	51.70	5.32
227	't Harde	't Harde	NL	52.41	5.89
8	Aachen	Aachen Hbf	D	50.77	6.09
818	Aachen W	Aachen West	D	50.78	6.07

The second dataset, `tariff-distances-2022-01.csv`, contains the station distances. The distances are defined as the shortest route on the railway network and they are used to calculate the tariffs for ticket. Let's peek into this file:

```
head -n 9 tariff-distances-2022-01.csv | cut -d, -f1-9
```

```
Station,AC,AH,AHP,AHPR,AHZ,AKL,AKM,ALM
AC,XXX,82,83,85,90,71,188,32
AH,82,XXX,1,3,8,77,153,98
AHP,83,1,XXX,2,9,78,152,99
AHPR,85,3,2,XXX,11,80,150,101
AHZ,90,8,9,11,XXX,69,161,106
AKL,71,77,78,80,69,XXX,211,96
AKM,188,153,152,150,161,211,XXX,158
ALM,32,98,99,101,106,96,158,XXX
```

We can see that the distances are encoded as a matrix with the diagonal entries set to `XXX`. As explained in the dataset's description, this string implies that the two stations are the same station. If we just load the values as `XXX`, the CSV reader will assume that all columns have the type `VARCHAR` instead of numeric values. While this can be cleaned up later, it's a lot easier to avoid this problem altogether. To do so, we use the `read_csv` function and set the `nullstr` parameter to `XXX`:

```
CREATE TABLE distances AS
    FROM read_csv(
        'https://blobs.duckdb.org/data/tariff-distances-2022-01.csv',
        nullstr = 'XXX'
    );
```

Then, using the `DESCRIBE` statement, we can confirm that DuckDB has inferred the column correctly as `BIGINT`:

```
FROM (DESCRIBE distances)
LIMIT 5;
```

column_name	column_type	null	key	default	extra
Station	VARCHAR	YES	NULL	NULL	NULL
AC	BIGINT	YES	NULL	NULL	NULL
AH	BIGINT	YES	NULL	NULL	NULL
AHP	BIGINT	YES	NULL	NULL	NULL
AHPR	BIGINT	YES	NULL	NULL	NULL

To show the first 9 columns, we can run the following query with the #1, #2, etc. column indexes in the SELECT statement:

```
SELECT #1, #2, #3, #4, #5, #6, #7, #8, #9
FROM distances
LIMIT 8;
```

Station	AC	AH	AHP	AHPR	AHZ	AKL	AKM	ALM
AC	NULL	82	83	85	90	71	188	32
AH	82	NULL	1	3	8	77	153	98
AHP	83	1	NULL	2	9	78	152	99
AHPR	85	3	2	NULL	11	80	150	101
AHZ	90	8	9	11	NULL	69	161	106
AKL	71	77	78	80	69	NULL	211	96
AKM	188	153	152	150	161	211	NULL	158
ALM	32	98	99	101	106	96	158	NULL

We can see that the data was loaded correctly but the wide table format is a bit unwieldy for further processing: to query for pairs of stations, we need to first turn it into a long table using the UNPIVOT statement. Naïvely, we would write something like the following:

```
CREATE TABLE distances_long AS
    UNPIVOT distances
    ON AC, AH, AHP, ...
```

However, we have almost 400 stations, so spelling out their names would be quite tedious. Fortunately, DuckDB has a trick to help with this: the COLUMNS(*) expression lists all columns and its optional EXCLUDE clause can remove given column names from the list. Therefore, the expression `COLUMNS(* EXCLUDE station)` lists all column names except `station`, precisely what we need for the UNPIVOT command:

```
CREATE TABLE distances_long AS
    UNPIVOT distances
    ON COLUMNS (* EXCLUDE station)
    INTO NAME other_station VALUE distance;
```

This results in the following table:

```
SELECT station, other_station, distance
FROM distances_long
LIMIT 3;
```

Station	other_station	distance
AC	AH	82
AC	AHP	83
AC	AHPR	85

Now we can join the `distances_long` table on the `stations` table along both the start and end stations, then filter for stations which are located in the Netherlands. We introduce symmetry breaking (`station < other_station`) to ensure that the same pair of stations only occurs once in the output. Finally, we select the top-3 results:

```
SELECT
    s1.name_long AS station1,
    s2.name_long AS station2,
    distances_long.distance
```

```
FROM distances_long
JOIN stations s1 ON distances_long.station = s1.code
JOIN stations s2 ON distances_long.other_station = s2.code
WHERE s1.country = 'NL'
  AND s2.country = 'NL'
  AND station < other_station
ORDER BY distance DESC
LIMIT 3;
```

The results show that there are pairs of train stations, which are at least 425 km away – quite the distance for such a small country!

station1	station2	distance
Eemshaven	Vlissingen	426
Eemshaven	Vlissingen Souburg	425
Bad Nieuweschans	Vlissingen	425

Conclusion

In this post, we demonstrated some of DuckDB's key features, including automatic detection of formats based on filenames, auto-inferencing the schema of CSV files, direct Parquet querying, remote querying, window functions, unpivot, several friendly SQL features (such as `FROM`-first, `GROUP BY ALL`, and `COLUMNS(*)`), and so on. The combination of these allows for formulating queries using different file formats (CSV, Parquet), data sources (local, HTTPS, S3), and SQL features. This helps users answer queries quickly and efficiently.

In the next installment, we'll take a look at temporal data using AsOf joins and geospatial data using the DuckDB `spatial` extension.

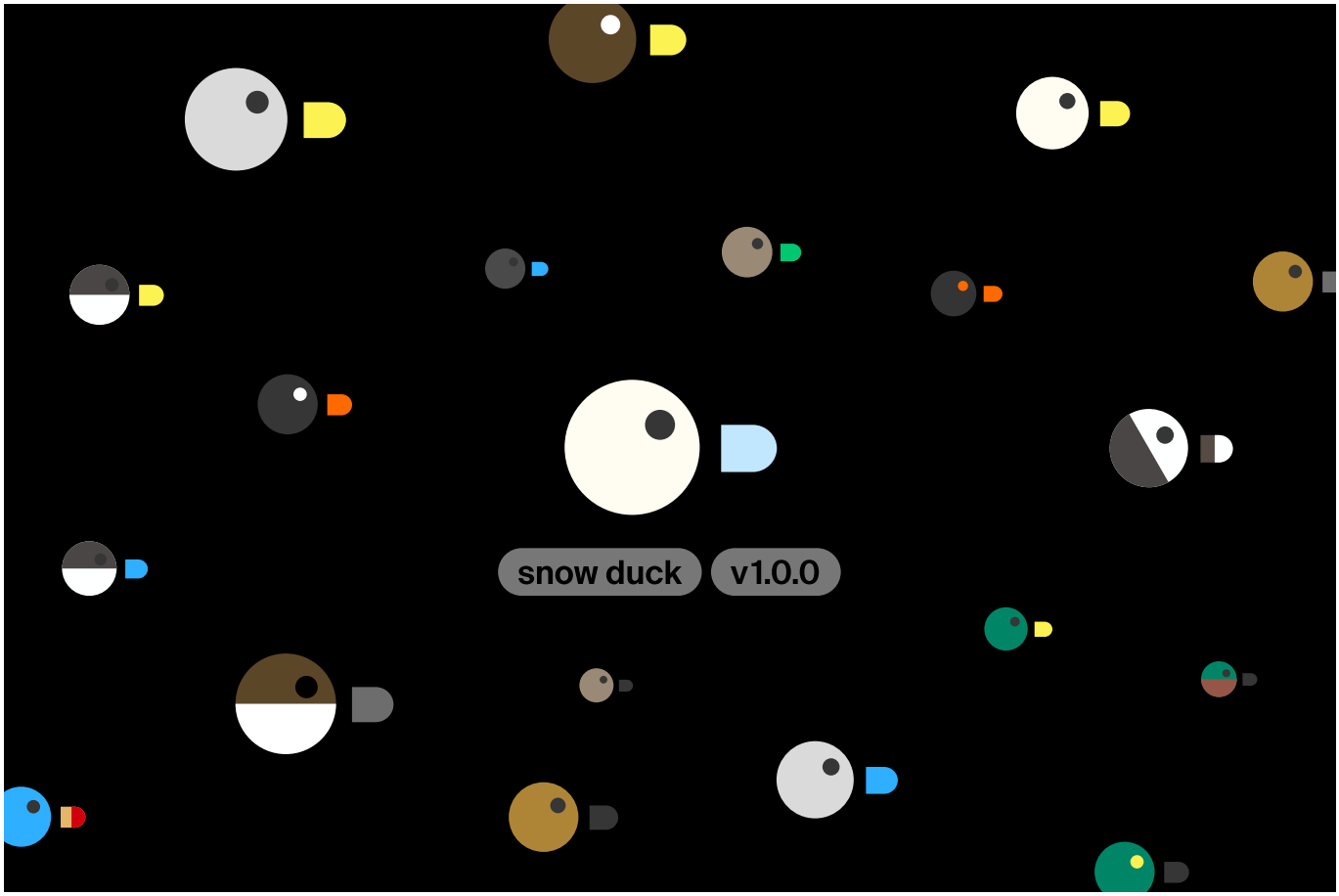
Announcing DuckDB 1.0.0

Publication date: 2024-06-03

Authors: Mark Raasveldt, Hannes Mühleisen

TL;DR: The DuckDB team is very happy to announce that today we're releasing DuckDB version 1.0.0, codename "Snow Duck" (anas nivis).

To install the new version, please visit the installation guide. For the release notes, see the release page.



It has been almost six years since the first source code was written for the project back in 2018, and a *lot* has happened since: There are now over 300 000 lines of C++ engine code, over 42 000 commits and almost 4 000 issues were opened and closed again. DuckDB has also gained significant popularity: the project has attracted tens of thousands of stars and followers on GitHub and social media platforms. Download counts are in the millions each month, and download traffic just for extensions is upwards of four terabytes *each day*. There are even books being written about DuckDB, and – most importantly – now even Wikipedia considers DuckDB notable, albeit barely.

Why Now?

Of course, version numbers are somewhat arbitrary and "feely", despite attempts at making them more mechanical. We could have released DuckDB 1.0.0 back in 2018, or we could have waited ten more years. There is never a great moment, because software (with the exception of TeX) is never "done". Why choose today?

Data management systems – even purely analytical ones – are such core components of any application that there is always an implicit contract of trust between their developers and users. Users rely on databases to provide correct query results and to not lose their data. At the same time, system developers need to be aware of their responsibility of not breaking people's applications willy-nilly. Intuitively, version 1.0.0 means something else for a data management system than it means for an egg timer app (no offense). From the very beginning, we were committed to making DuckDB a reliable base for people to build their applications on. This is also why the 1.0.0 release is named after the non-existent *snow duck (anas nivis),* harking back to Apple's Snow Leopard release some years ago.

For us, one of the major blockers to releasing 1.0.0 was the storage format. DuckDB has its own custom-built data storage format. This format allows users to manage many (possibly very large) tables in a single file with full transactional semantics and state-of-the-art compression. Of course, designing a new file format is not without its challenges, and we had to make significant changes to the format over time. This led to the suboptimal situation that whenever a new DuckDB version was released, the files created with the old version did not work with the new DuckDB version and had to be manually upgraded. This problem was addressed in v0.10.0 back in February – where we introduced backward compatibility and limited forward compatibility for DuckDB's storage format. This feature has now been used in the wild for a while without serious issues – providing us with the confidence to offer a guarantee that DuckDB files created with DuckDB 1.0.0 will be compatible with future DuckDB versions.

Stability

The core theme of the 1.0.0 release is stability. This contrasts it with previous releases where we have had blog posts talk about long lists of new features. Instead, the 1.0.0 release has very limited new features (a few might have snuck in). Instead, our focus has been on stability.

We've observed the frankly staggering growth in the amount and breadth of use of DuckDB in the wild, and have not seen an increase in serious issues being reported. Meanwhile, there are thousands of test cases with millions of test queries being run every night. We run loads of microbenchmarks and standardized benchmark suites to spot performance regressions. DuckDB is constantly being tortured by various fuzzers that construct all manners of wild SQL queries to make sure we don't miss weird corner cases. All told, this has built the necessary confidence in us to release a 1.0.0.

Another core aspect of stability with the 1.0.0 release is stability across versions. While never breaking anyone's workflow is likely impossible, we plan to be much more careful with user-facing changes going forward. In particular, we plan to focus on providing stability for the SQL dialect, as well as the C API. While we do not guarantee that we will never change semantics in these layers in the future – we will try to provide ample warning when doing so, as well as providing workarounds that allow previously working code to keep on working.

Looking ahead

Unlike many open-source projects, DuckDB also has a healthy long-term funding strategy. DuckLabs, the company that employs DuckDB's core contributors, has not had any outside investments, and as a result, the company is fully owned by the team. Labs' business model is to provide consulting and support services for DuckDB, and we're happy to report that this is going well. With the revenue from contracts, we fund long-term and strategic DuckDB development with a team of almost 20 people. At the same time, the intellectual property in the project is guarded by the independent DuckDB Foundation. This non-profit foundation ensures that DuckDB will be around long-term under the MIT license.

Regarding long-term plans, there are, of course, many things on the roadmap still. One thing we're very excited about is the ability to expand the extension environment around DuckDB. Extensions are plug-ins that can add new SQL-level functions, file formats, optimizers, etc. while keeping the DuckDB core mean and lean. There are already an impressive number of third-party extensions to DuckDB, and we're working hard to streamline the process of building and distributing community-contributed extensions. We think DuckDB can become the basis for the next revolution in data through community extensions connected by a high-performance data fabric accessible through a unified SQL interface.

Of course, there will be issues found in today's release. But rest assured, there will be a 1.0.1 release. There will be a 1.1.0. And there might also be a 2.0.0 at some point. We're in this for the long run, all of us, together. We have the team and the structures and resources to do so.

Acknowledgments

First of all, we are very, very grateful to you all. Our massive and heartfelt thanks go to everyone who has contributed code, filed issues or engaged in discussions, promoted DuckDB in their environment, and, of course, all DuckDB users. We could not have done it without you!

We would also like to thank the CWI Database Architectures group for providing us with the environment and expertise to build DuckDB, the organizations that provided us with research grants early on, the excellent customers of DuckLabs that make it all work (especially the early ones), and the generous donors to the DuckDB Foundation. We are particularly grateful to our long-standing Gold sponsors MotherDuck, Voltron Data and Posit.

Finally, we would like to thank the excellent and amazing team at DuckLabs.

So join us now in being nostalgic, teary-eyed and excited for what's to come for DuckDB and celebrate the release of DuckDB 1.0.0 with us. We certainly will.

Mark and Hannes

Native Delta Lake Support in DuckDB

Publication date: 2024-06-10

Author: Sam Ansmink

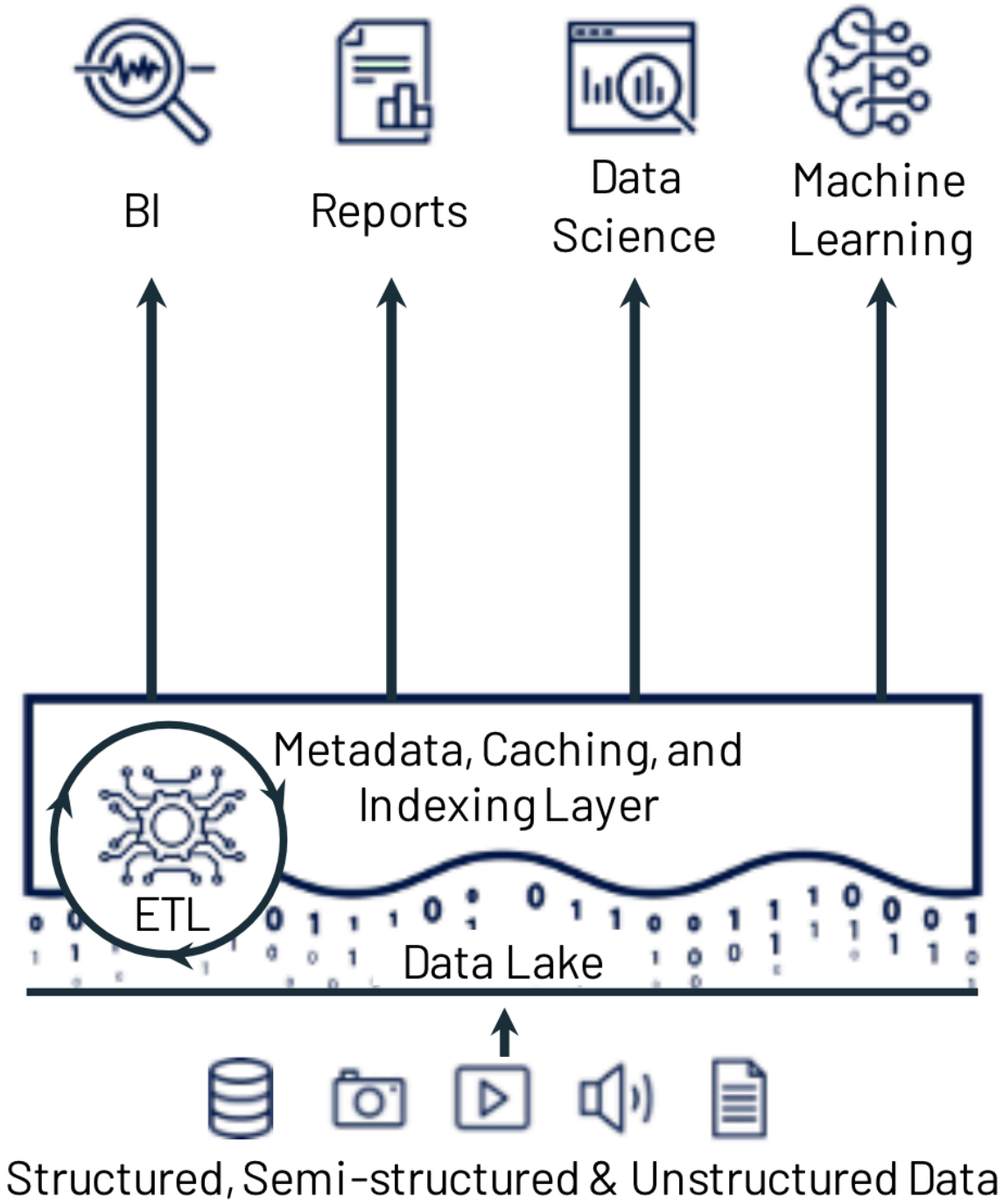
TL;DR: DuckDB now has native support for Delta Lake, an open-source lakehouse framework, with the `delta` extension.

Over the past few months, DuckLabs has teamed up with Databricks to add first-party support for Delta Lake in DuckDB using the new `delta-kernel-rs` project. In this blog post we'll give you a short overview of Delta Lake, Delta Kernel and, of course, present the new DuckDB Delta extension.

If you're already dearly familiar with Delta Lake and Delta Kernel, or you are just here to know how to boogie, feel free to skip to the juicy bits on how to use the DuckDB with Delta.

Intro

Delta Lake is an open-source storage framework that enables building a lakehouse architecture. So to understand Delta Lake, we need to understand what the lakehouse architecture is. The lakehouse is a data management architecture that strives to combine the cost-effectiveness of cheap object storage with a smart management layer. In simple terms, lakehouse architectures are a collection of files in various formats, with some additional metadata layers on top. These metadata layers aim to provide extra functionality on top of the raw collection of files such as ACID transactions, time travel, partition- and schema evolution, statistics, and much more. What a lakehouse architecture enables, is to run various types of data-intensive applications such as data analytics and machine learning applications, directly on a vast collection of structured, semi-structured and unstructured data, without the need for an intermediate data warehousing step. If you're ready for the deep dive, we recommend reading the CIDR 2021 paper "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics" by Michael Armbrust et al. However, if you're (understandably) hesitant to dive into dense scientific literature, this image sums it up pretty well:



Lakehouse architecture (image source: Armburst et al., CIDR 2021)

Delta Lake

Now let's zoom in a little on our star of the show for tonight, Delta Lake. Delta Lake (or simply "Delta") is currently one of the leading open-source lakehouse formats, along with Apache Iceberg and Apache Hudi. The easiest way to get a feeling for what a Delta table is, is to think of a Delta table as a "collection of Parquet files with some metadata". With this slight oversimplification in mind, we will now create a Delta table and examine the files that are created, to improve our understanding. To do this, we'll set up Python with the packages: `duckdb`, `pandas` and `deltalake`:

```
pip install duckdb pandas deltalake
```

Then, we use DuckDB to create some dataframes with test data, and write that to a Delta table using the `deltalake` package:

```
import duckdb
from deltalake import DeltaTable, write_deltalake
con = duckdb.connect()
df1 = con.query("SELECT i AS id, i % 2 AS part, 'value-' || i AS value FROM range(0, 5) tbl(i)").df()
df2 = con.query("SELECT i AS id, i % 2 AS part, 'value-' || i AS value FROM range(5, 10) tbl(i)").df()
write_deltalake(f"./my_delta_table", df1,  partition_by=["part"])
write_deltalake(f"./my_delta_table", df2,  partition_by=["part"], mode='append')
```

With this script run, we have created a basic Delta table containing 10 rows, split across two partitions that we added in two separate steps. To double-check that everything is going to plan, let's use DuckDB to query the table:

```
SELECT *
FROM delta_scan('./my_delta_table')
ORDER BY id;
```

id	part	value
0	0	value-0
1	1	value-1
2	0	value-2
3	1	value-3
4	0	value-4
5	1	value-5
6	0	value-6
7	1	value-7
8	0	value-8
9	1	value-9

That looks great! All our expected data is there. Now let's take a look at what files have actually been created using `tree`:

```
tree ./my_delta_table`
```

```
my_delta_table
├── _delta_log
│   ├── 00000000000000000000.json
│   └── 00000000000000000001.json
├── part=0
│   ├── 0-f45132f6-2231-4dbd-aabb-1af29bf8724a-0.parquet
│   └── 1-76c82535-d1e7-4c2f-b700-669019d94a0a-0.parquet
└── part=1
    ├── 0-f45132f6-2231-4dbd-aabb-1af29bf8724a-0.parquet
    └── 1-76c82535-d1e7-4c2f-b700-669019d94a0a-0.parquet
```

The `tree` output shows 2 different types of files. While a Delta table can contain various other types of files, these form the basis of any Delta table.

Firstly, there are **data files** in Parquet format. The data files contain all the data that is stored in the table. This is very similar to how data is stored when DuckDB is used to write partitioned Parquet files.

Secondly, there are **delta files** in JSON format. The Delta files contain a log of the changes that have been made to the table. By replaying this log, a reader can construct a valid view of the table. To illustrate this, let's take a small peek into one of the first Delta log files:

```
cat my_delta_table/_delta_log/00000000000000000000.json
```

```
...
{ "add": {
    "path": "part=1/0-f45132f6-2231-4dbd-aabb-1af29bf8724a-0.parquet",
    "partitionValues": { "part": "1" }
  },
  ...
}
{ "add": {
    "path": "part=0/0-f45132f6-2231-4dbd-aabb-1af29bf8724a-0.parquet",
    "partitionValues": { "part": "0" },
  },
  ...
}
...
```

As we can see, this log file contains two `add` objects that describe some data being added to respectively the `1` and `0` partitions. Note also that the partition values themselves are stored in these Delta files explicitly, so even though the file structure looks very similar to a Hive-style partitioning scheme, the folder names are not actually used by Delta internally. Instead, the partition values are read from the metadata.

Now with this simple example, we've shown the basics of how Delta works. For a more thorough understanding of the internals, we refer to the official Delta specification, which is, by protocol specification standards, quite easy to read. The official specification describes in detail how Delta handles every detail, from the basics described here to more complex things like checkpointing, deletes, schema evolution, and much more.

Implementation

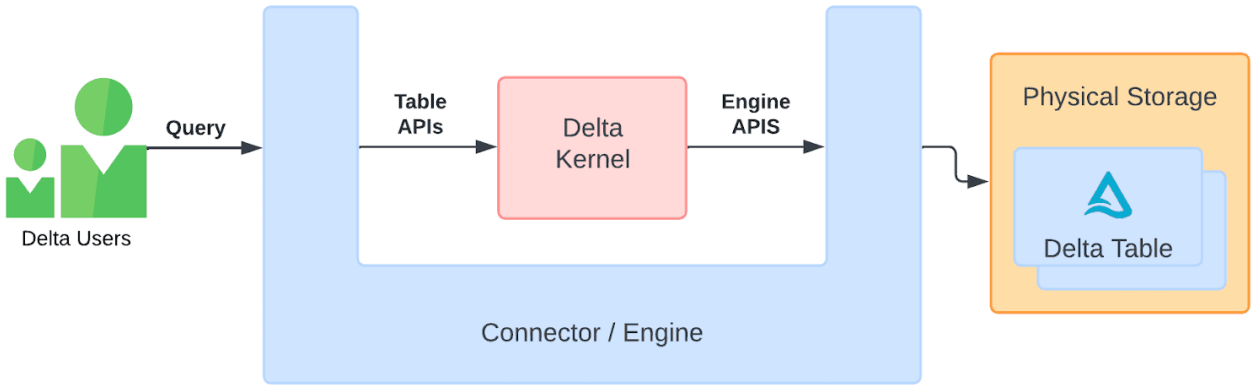
The Delta Kernel

Supporting a relatively complex protocol such as Delta, requires significant development and maintenance effort. For this reason, when looking to add support for such a protocol to an engine, the logical choice would be to look for a ready-to-use library to take care of this. In the case of Delta Lake, we could, for example, opt for the `delta-rs` library. However, when it comes to implementing a native DuckDB Delta extension, this is problematic: if we were to use the `delta-rs` library for implementing the DuckDB extension, all interaction with the Delta tables would go through the `delta-rs` library. But remember, a Delta table is effectively *"just a bunch of Parquet files with some metadata"*. Therefore, this would mean that when DuckDB wants to read a Delta table, the data files will be read by the `delta-rs` Parquet reader, using the`delta-rs` filesystem. But that's annoying: DuckDB already comes shipped with an excellent Parquet reader. Also, DuckDB already has support for a variety of filesystems with its own credential management system. By using a library like `delta-rs` for DuckDB's Delta extension this would actually run into a variety of problems:

- increased extension binary size
- inconsistent user experience between `delta_scan` and `read_parquet`
- increased maintenance load

Now to solve these problems, we would prefer to have some library that implements **only the Delta protocol** while letting DuckDB handle all the things it already knows how to handle.

Fortunately for us, this library exists and it's called the Delta Kernel Project. The Delta Kernel is a "set of libraries for building Delta connectors that can read from and write into Delta tables without the need to understand the Delta protocol details". This is done by exposing two relatively simple sets of APIs that an engine would implement, as shown in the image below:



For more details on the `delta-kernel-rs` project, we refer to this excellent blog post, which goes in-depth into the internals and design rationale.

Now while the `delta-kernel-rs` library is still experimental, it has recently launched its v0.1.0 version, and already offers a lot of functionality. Furthermore, because `delta-kernel-rs` exposes a C/C++ foreign function interface, integrating it into a DuckDB extension has been very straightforward.

DuckDB Delta Extension `delta_scan`

Now we're ready to dive into the nitty-gritties of the DuckDB Delta extension internals. To start, the Delta extension currently implements a single table function: `delta_scan`. It's a simple but powerful function that scans a Delta Table.

To understand how this function is implemented, we first need to establish the four main components involved:

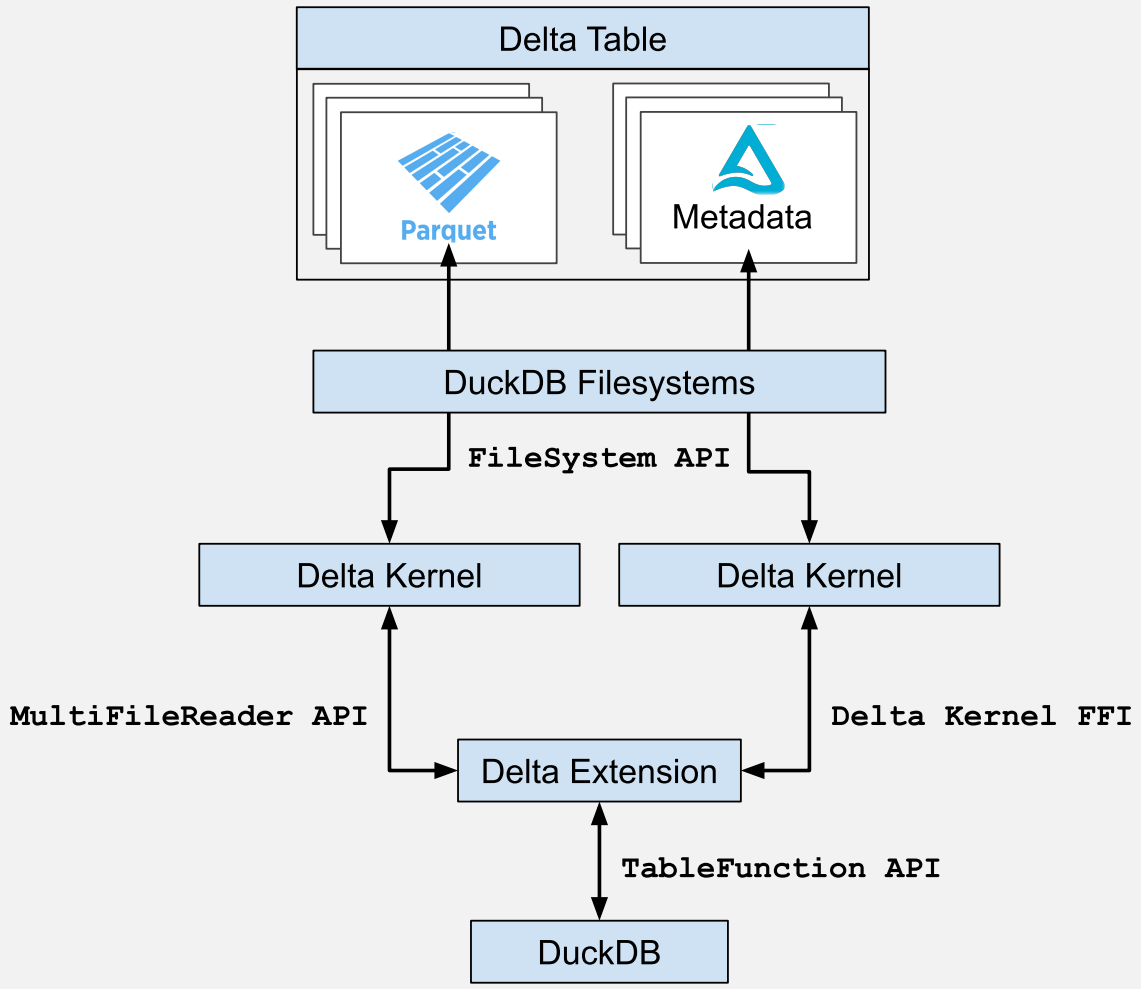
Component	Description
Delta kernel	The delta-kernel-rs library
Delta extension	DuckDB's loadable Delta extension
Parquet extension	DuckDB's loadable Parquet extension
DuckDB	Super cool duck-themed analytical database

Additionally, we need to understand that there are four main APIs involved:

API	Description
`FileSystem`	DuckDB's API for I/O (for local files, Azure, S3, etc.)
`TableFunction`	DuckDB's API for table functions (e.g., `read_parquet`, `read_csv`)
`MultiFileReader`	DuckDB's API for handling multi-file scans
Delta Kernel C/C++ FFI	Delta Kernel FFI for Delta Lake

Now we have all the links, let's tie them all together. When a user runs a query with a `delta_scan` table function, DuckDB will call into the `delta_scan` function from the Delta extension using the `TableFunction` API. The `delta_scan` table function, however, is actually just an exact copy of the regular `read_parquet` function. To change the `read_parquet` into a `delta_scan`, it will replace the regular `MultiFileReader` of the `parquet_scan` (which simply scans a list or glob of files), with a custom `DeltaMultiFileReader` that

will generate a list of files based on the Delta Table metadata. Finally, whenever the Parquet extension requires any IO, it will call into DuckDB using the `FileSystem` API to handle the I/O. This entire interaction is captured in the diagram below.



In this Diagram, we can see all four components involved in the processing of query containing a `delta_scan` table function. The arrows represent the communication that occurs between the components across the four APIs. Now when reading a Delta Table, we can see that the Metadata is handled on the right side going through the Delta Kernel. On the left side we can see how the Parquet data flows through the Parquet extension.

While there are obviously some important details missing here, such as the handling of deletion vectors and column mappings, we have now covered the basic concept of the DuckDB Delta extension. Also, we have demonstrated how the current implementation achieves a very natural logical separation, with component internals being abstracted away by connecting through clearly defined APIs. In doing so, the implementation achieves the following key properties:

1. **The details of the Delta protocol remain largely opaque to any DuckDB component.** The only point of contact with the internals of the Delta protocol is the narrow FFI exposed by the Delta kernel. This is fully handled by the Delta extension, whose only job is to translate this into native DuckDB APIs.
2. **Full reuse of existing Parquet scanning logic** of DuckDB, without any code reuse or compile time dependencies between extensions. Because all interaction between the Delta and Parquet extension is done over DuckDB APIs through the running DuckDB instance, the extensions only interface over the `TableFunction` and `MultiFileReader` APIs. This also means that any future optimizations

that are made to the Parquet extension will automatically be available in the Delta extension.

3. **All I/O will go through DuckDB's `FileSystem` API.** This means that all file systems (Azure, S3, etc.) that are available to DuckDB, are available to scan with. This means that any DuckDB file system that can read and list files can be used for Delta. This is also useful in DuckDB-Wasm where custom filesystem implementations are used. *Warning*, two small notes need to be made here. Firstly, currently the DuckDB Delta extension still lets a small part of IO be handled by the Delta kernel through internal filesystem libraries, this is due to the FFI not yet exposing the `FileSystem` APIs, but this will change very soon. Secondly, while the architectural design of the Delta Extension is made with DuckDB-Wasm in mind, the Wasm version of the extension is not yet available.

How to Use Delta in DuckDB

Using the Delta extension in DuckDB is very simple, as it is distributed as one of the core DuckDB extensions, and available for autoloading. What this means is that you can simply start DuckDB (using v0.10.3 or higher) and run:

```
SELECT * FROM delta_scan('./my_delta_table');
```

DuckDB will automatically install and load the Delta Extension. Then it will query the local Delta table `./my_delta_table`.

In case your Delta table lives on S3, there are probably some S3 credentials that you want to set. If these credentials are already in one of the default places, such as an environment variable, or in the `~/.aws/credentials` file? Simply run:

```
CREATE SECRET delta_s1 (
    TYPE s3,
    PROVIDER credential_chain
)
SELECT * FROM delta_scan('s3://<some-bucket>/<path/to/a/delta/table>');
```

Do you prefer remembering your AWS tokens by heart, and would like to type them out? Go with:

```
CREATE SECRET delta_s2 (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<eu-west-1>'
)
SELECT * FROM delta_scan('s3://<some-bucket>/<path/to/a/delta/table>');
```

Do you have multiple Delta tables, with different credentials? No problem, you can use scoped secrets:

```
CREATE SECRET delta_s3 (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE1>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY1>',
    REGION '<eu-west-1>',
    SCOPE 's3://<some-bucket-1>'
)
CREATE SECRET delta_s4 (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE2>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY2>',
    REGION '<us-west-1>',
    SCOPE 's3://<some-bucket-2>'
)
SELECT * FROM delta_scan('s3://<some-bucket-1>/<table1>');
SELECT * FROM delta_scan('s3://<some-bucket-2>/<table2>');
```

Finally, is your table public, but outside the default AWS region? Make sure you set the region using an empty S3 Secret:

```
CREATE SECRET delta_s5 (
    TYPE s3,
    REGION '<eu-west-2>'
)
SELECT * FROM delta_scan('s3://<some-public-bucket>/<table1>');
```

Current State of the Delta Extension

Currently, the Delta Extension is still considered **experimental**. This is partly because the Delta extension itself is still very new, but also because the `delta-kernel-rs` project it relies on is still experimental. Nevertheless, core Delta scanning features are already supported by the current version of the Delta extension, such as:

- All data types
- Filter and projection pushdown
- File skipping based on filter pushdown
- Deletion vectors
- Partitioned tables
- Fully parallel scanning

The Delta extension is available on the platforms `linux_amd64`, `linux_arm64`, `osx_amd64` and `osx_arm64`. Support for the remaining platforms is coming soon. Additionally, we will continue to work together with Databricks on further improving the Delta Extension to add more features such as:

- Write support
- Column mapping
- Time travel
- Variant, RowIds
- Wasm support

For details and info on newly added features, keep an eye on the Delta extension docs and repository.

Conclusion

In this blog post, we presented DuckDB's new Delta extension, enabling easy interaction with Delta Lake directly from the comfort of your own DuckDB environment. To do so, we demonstrated what the Delta Lake format looks like by creating a Delta table and analyzing it using DuckDB.

We want to emphasize the fact that by implementing the Delta extension with the `delta-kernel-rs` library, both DuckDB and the Delta extension have been kept relatively simple and largely agnostic to the internals of the Delta protocol.

We hope you give the Delta extension a try and look forward to any feedback from the community! Also, if you're attending the 2024 Databricks Data + AI Summit be sure to check out DuckDB co-founder Hannes Mühleisen's talk on Thursday during the keynote and the in-depth breakout session, also on Thursday, for more details on the DuckDB–Delta integration.

Command Line Data Processing: Using DuckDB as a Unix Tool

Publication date: 2024-06-20

Author: Gábor Szárnyas

TL;DR: DuckDB's CLI client is portable to many platforms and architectures. It handles CSV files conveniently and offers users the same rich SQL syntax everywhere. These characteristics make DuckDB an ideal tool to complement traditional Unix tools for data processing in the command line.

In this blog post, we dive into the terminal to compare DuckDB with traditional tools used in Unix shells (Bash, Zsh, etc.). We solve several problems requiring operations such as projection and filtering to demonstrate the differences between using SQL queries in DuckDB versus specialized command line tools. In the process, we will show off some cool features such as DuckDB's powerful CSV reader and the positional join operator. Let's get started!

The Unix Philosophy

To set the stage, let's recall the Unix philosophy. This states that programs should:

- do one thing and do it well,
- work together, and
- handle text streams.

Unix-like systems such as macOS, Linux and WSL in Windows have embraced this philosophy. Tools such as `grep`, `sed`, and `sort` are ubiquitous and widely used in shell scripts.

As a purpose-built data processing tool, DuckDB fits the Unix philosophy quite well. First, it was designed to be a fast in-process analytical SQL database system *(do one thing and do it well).* Second, it has a standalone command line client, which can consume and produce CSV files *(work together),* and also supports reading and writing text streams *(handle text streams)*. Thanks to these, DuckDB works well in the ecosystem of Unix CLI tools, as shown in several posts.

Portability and Usability

While Unix CLI tools are fast, robust, and available on all major platforms, they often have cumbersome syntax that's difficult to remember. To make matters worse, these tools often come with slight differences between systems – think of the differences between GNU `sed` and macOS's `sed` or the differences between regex syntax among programs, which is aptly captured by Donald Knuth's quip *"I define Unix as 30 definitions of regular expressions living under one roof."*

While there are shells specialized specifically for dataframe processing, such as the Nushell project, older Unix shells (e.g., the Bourne shell `sh` and Bash) are still the most wide-spread, especially on servers.

At the same time, we have DuckDB, an extremely portable database system which uses the same SQL syntax on all platforms. With version 1.0.0 released recently, DuckDB's syntax – based on the proven and widely used PostgreSQL dialect – is now in a stable state. Another attractive feature of DuckDB is that it offers an interactive shell, which aids quick debugging. Moreover, DuckDB is available in several host languages as well as in the browser via WebAssembly, so if you ever decide to use your SQL scripts outside of the shell, DuckDB SQL scripts can be ported to a wide variety of environments without any changes.

Data Processing with Unix Tools and DuckDB

In the following, we give examples for implementing simple data processing tasks using the CLI tools provided in most Unix shells and using DuckDB SQL queries. We use DuckDB v1.0.0 and run it in in-memory mode. This mode makes sense for the problems we are tackling, as we do not create any tables and the operations are not memory-intensive, so there is no data to persist or to spill on disk.

Datasets

We use the four input files capturing information on cities and airports in the Netherlands.

You can download all input files as a single zip file.

Projecting Columns

Projecting columns is a very common data processing step. Let's take the `pop.csv` file and project the first and last columns, `city` and `population`.

Unix Shell: `cut` In the Unix shell, we use the `cut` command and specify the file's delimiter (`-d`) and the columns to be projected (`-f`).

```
cut -d , -f 1,3 pop.csv
```

This produces the following output:

```
city,population
Amsterdam,905234
Rotterdam,656050
The Hague,552995
Utrecht,361924
Eindhoven,238478
Groningen,234649
Tilburg,224702
Almere,218096
Breda,184716
Nijmegen,179073
```

DuckDB: `SELECT` In DuckDB, we can use the CSV reader to load the data, then use the `SELECT` clause with column indexes (`#i`) to designate the columns to be projected:

```
SELECT #1, #3 FROM 'pop.csv';
```

Note that we did not have to define any schema or load the data to a table. Instead, we simply used `'pop.csv'` in the `FROM` clause as we would do with a regular table. DuckDB detects that this is a CSV file and invokes the `read_csv` function, which automatically infers the CSV file's dialect (delimiter, presence of quotes, etc.) as well as the schema of the table. This allows us to simply project columns using `SELECT #1, #3`. We could also use the more readable syntax `SELECT city, population`.

To make the output of the solutions using Unix tools and DuckDB equivalent, we wrap the query into a `COPY ... TO` statement:

```
COPY (
    SELECT #1, #3 FROM 'pop.csv'
  ) TO '/dev/stdout/';
```

This query produces the same result as the Unix command's output shown above.

To turn this into a standalone CLI command, we can invoke the DuckDB command line client with the `-c <query>{:.language-sql .highlight}` argument, which runs the SQL query and exits once it's finished. Using this technique, the query above can be turned into the following one-liner:

```
duckdb -c "COPY (SELECT #1, #3 FROM 'pop.csv') TO '/dev/stdout/'"
```

In the following, we'll omit the code blocks using the standalone `duckdb{:.language-sql .highlight}` command: all solutions can be executed in the `duckdb -c <query>{:.language-sql .highlight}` template and yield the same result as the solutions using Unix tools.

Sorting Files

Another common task is to sort files based on given columns. Let's rank the cities within provinces based on their populations. To do so, we need to sort the `pop.csv` file first based on the name of the `province` using an ascending order, then on the `population` using a descending order. We then return the `province` column first, followed by the `city` and the `population` columns.

Unix Shell: `sort` In the Unix shell, we rely on the `sort` tool. We specify the CSV file's separator with the `-t` argument and set the keys to sort on using `-k` arguments. We first sort on the second column (`province`) with `-k 2,2`. Then, we sort on the third column (`population`), setting the ordering to be reversed (`r`) and numeric (`n`) with `-k 3rn`. Note that we need to handle the header of the file separately: we take the first row with `head -n 1` and the rest of the rows with `tail -n +2`, sort the latter, and glue them back together with the header. Finally, we perform a projection to reorder the columns. Unfortunately, the `cut` command cannot reorder the columns, so we use awk instead:

```
(head -n 1 pop.csv; tail -n +2 pop.csv \
    | sort -t , -k 2,2 -k 3rn) \
    | awk -F , '{ print $2 "," $1 "," $3 }'
```

The result is the following:

```
province,city,population
Flevoland,Almere,218096
Gelderland,Nijmegen,179073
Groningen,Groningen,234649
North Brabant,Eindhoven,238478
North Brabant,Tilburg,224702
North Brabant,Breda,184716
North Holland,Amsterdam,905234
South Holland,Rotterdam,656050
South Holland,The Hague,552995
Utrecht,Utrecht,361924
```

DuckDB: `ORDER BY` In DuckDB, we simply load the CSV and specify the column ordering via `SELECT province, city, population`, then set the sorting criteria on the selected columns (`province` ASC and `population` DESC). The CSV reader automatically detects types, so the sorting is numeric by default. Finally, we surround the query with a COPY statement to print the results to the standard output.

```
COPY (
    SELECT province, city, population
    FROM 'pop.csv'
    ORDER BY province ASC, population DESC
  ) TO '/dev/stdout/';
```

Intersecting Columns

A common task is to calculate the intersection of two columns, i.e., to find entities that are present in both. Let's find the cities that are both in the top-10 most populous cities and have their own airports.

Unix Shell: `comm` The Unix solution for intersection uses the `comm` tool, intended to compare two *sorted* files line-by-line. We first `cut` the relevant column from both files. Due to the sorting requirement, we apply `sort` on both inputs before performing the intersection. The intersection is performed using `comm -12` where the argument `-12` means that we only want to keep lines that are in both files. We again rely on `head` and `tail` to treat the headers and the rest of the files separately during processing and glue them together at the end.

```
head -n 1 pop.csv | cut -d , -f 1; \
    comm -12 \
        <(tail -n +2 pop.csv | cut -d , -f 1 | sort) \
        <(tail -n +2 cities-airports.csv | cut -d , -f 1 | sort)
```

The script produces the following output:

```
city
Amsterdam
Eindhoven
Groningen
Rotterdam
The Hague
```

DuckDB: `INTERSECT ALL` The DuckDB solution reads the CSV files, projects the `city` fields and applies the `INTERSECT ALL` clause to calculate the intersection:

```
COPY (
    SELECT city FROM 'pop.csv'
    INTERSECT ALL
    SELECT city FROM 'cities-airports.csv'
  ) TO '/dev/stdout/';
```

Pasting Rows Together

Pasting rows together line-by-line is a recurring task. In our example, we know that the `pop.csv` and the `area.csv` files have an equal number of rows, so we can produce a single file that contains both the population and the area of every city in the dataset.

Unix Shell: `paste` In the Unix shell, we use the `paste` command and remove the duplicate `city` field using `cut`:

```
paste -d , pop.csv area.csv | cut -d , -f 1,2,3,5
```

The output is the following:

```
city,province,population,area
Amsterdam,North Holland,905234,219.32
Rotterdam,South Holland,656050,324.14
The Hague,South Holland,552995,98.13
Utrecht,Utrecht,361924,99.21
Eindhoven,North Brabant,238478,88.92
Groningen,Groningen,234649,197.96
Tilburg,North Brabant,224702,118.13
Almere,Flevoland,218096,248.77
Breda,North Brabant,184716,128.68
Nijmegen,Gelderland,179073,57.63
```

DuckDB: `POSITIONAL JOIN` In DuckDB, we can use a `POSITIONAL JOIN`. This join type is one of DuckDB's SQL extensions and it provides a concise syntax to combine tables row-by-row based on each row's position in the table. Joining the two tables together using `POSITIONAL JOIN` results in two `city` columns – we use the `EXCLUDE` clause to remove the duplicate column:

```
COPY (
    SELECT pop.*, area.* EXCLUDE city
    FROM 'pop.csv'
    POSITIONAL JOIN 'area.csv'
  ) TO '/dev/stdout/';
```

Filtering

Filtering is another very common operation. For this, we'll use `cities-airports.csv` file. For each airport, this file contains its `IATA` code and the main cities that it serves:

```
city,IATA
Amsterdam,AMS
Haarlemmermeer,AMS
Eindhoven,EIN
...
```

Let's try to formulate two queries:

1. Find all cities whose name ends in `dam`.
2. Find all airports whose IATA code is equivalent to the first three letters of a served city's name, but the city's name does *not* end in `dam`.

Unix Shell: `grep` To answer the first question in the Unix shell, we use `grep` and the regular expression `^[^,]*dam,`:

```
grep "^[^,]*dam," cities-airports.csv
```

In this expression, `^` denotes the start of the line, `[^,]*` searches for a string that does not contain the comma character (the separator). The expression `dam,` ensures that the end of the string in the first field is `dam`. The output is:

```
Amsterdam,AMS
Rotterdam,RTM
```

Let's try to answer the second question. For this, we need to match the first three characters in the `city` field to the `IATA` field but we need to do so in a case-insensitive manner. We also need to use a negative condition to exclude the lines where the city's name ends in `dam`. Both of these requirements are difficult to achieve with a single `grep` or `egrep` command as they lack support for two features. First, they do not support case-insensitive matching *using a backreference* (`grep -i` alone is not sufficient to ensure this). Second, they do not support negative lookbehinds. Therefore, we use `pcregrep`, and formulate our question as follows:

```
pcregrep -i '^([a-z]{3}).*?(?<!dam),\1$' cities-airports.csv
```

Here, we call `pcregrep` with the case-insensitive flag (`-i`), which in `pcregrep` also affects backreferences such as `\1`. We capture the first three letters with `([a-z]{3})` (e.g., `Ams`) and match it to the second field with the backreference: `,\1$`. We use a non-greedy `.*?` to seek to the end of the first field, then apply a negative lookbehind with the `(?<!dam)` expression to ensure that the field does not end in `dam`. The result is a single line:

```
Eindhoven,EIN
```

DuckDB: `WHERE ... LIKE` Let's answer the questions now in DuckDB. To answer the first question, we can use `LIKE` for pattern matching. The header should not be part of the output, so we disable it with `HEADER false`. The complete query looks like follows:

```
COPY (
    FROM 'cities-airports.csv'
    WHERE city LIKE '%dam'
  ) TO '/dev/stdout/' (HEADER false);
```

For the second question, we use string slicing to extract the first three characters, `upper` to ensure case-insensitivity, and `NOT LIKE` for the negative condition:

```
COPY (
    FROM 'cities-airports.csv'
    WHERE upper(city[1:3]) = IATA
      AND city NOT LIKE '%dam'
  ) TO '/dev/stdout/' (HEADER false);
```

These queries return exactly the same results as the solutions using `grep` and `pcregrep`.

In both of these queries, we used the FROM-first syntax. If the `SELECT` clause is omitted, the query is executed as if `SELECT *` was used, i.e., it returns all columns.

Joining Files

Joining tables is an essential task in data processing. Our next example is going to use a join to return city name–airport name combinations. This is achieved by joining the `cities-airports.csv` and the `airport-names.csv` files on their IATA code fields.

Unix Shell: `join` Unix tools support joining files via the `join` command, which joins lines of two *sorted* inputs on a common field. To make this work, we sort the files based on their `IATA` fields, then perform the join on the first file's 2nd column (`-1 2`) and the second file's 1st column (`-2 1`). We have to omit the header for the `join` command to work, so we do just that and construct a new header with an `echo` command:

```
echo "IATA,city,airport name"; \
    join -t , -1 2 -2 1 \
        <(tail -n +2 cities-airports.csv | sort -t , -k 2,2) \
        <(tail -n +2 airport-names.csv   | sort -t , -k 1,1)
```

The result is the following:

```
IATA,city,airport name
AMS,Amsterdam,Amsterdam Airport Schiphol
AMS,Haarlemmermeer,Amsterdam Airport Schiphol
EIN,Eindhoven,Eindhoven Airport
GRQ,Eelde,Groningen Airport Eelde
GRQ,Groningen,Groningen Airport Eelde
MST,Beek,Maastricht Aachen Airport
MST,Maastricht,Maastricht Aachen Airport
RTM,Rotterdam,Rotterdam The Hague Airport
RTM,The Hague,Rotterdam The Hague Airport
```

DuckDB In DuckDB, we load the CSV files and connect them using the `NATURAL JOIN` clause, which joins on column(s) with the same name. To ensure that the result matches with that of the Unix solution, we use the `ORDER BY ALL` clause, which sorts the result on all columns, starting from the first one, and stepping through them for tie-breaking to the last column.

```
COPY (
    SELECT "IATA", "city", "airport name"
    FROM 'cities-airports.csv'
    NATURAL JOIN 'airport-names.csv'
    ORDER BY ALL
  ) TO '/dev/stdout/';
```

Replacing Strings

You may have noticed that we are using very clean datasets. This is of course very unrealistic, so in an evil twist, let's reduce the data quality a bit:

- Replace the space in the province's name with an underscore, e.g., turning `North Holland` to `North_Holland`.
- Add thousand separating commas, e.g., turning `905234` to `905,234`.
- Change the CSV's separator to the semicolon character (`;`).

And while we're at it, also fetch the data set via HTTPS this time, using the URL `https://duckdb.org/data/cli/pop.csv`.

Unix Shell: `curl` and `sed` In Unix, remote data sets are typically fetched via `curl`. The output of `curl` is piped into the subsequent processing steps, in this case, a bunch of `sed` commands.

```
curl -s https://duckdb.org/data/cli/pop.csv \
    | sed 's/\([^,]*,.*\) \(.*,[^,]*\)/\1_\2/g' \
    | sed 's/,/;/g' \
    | sed 's/\([0-9][0-9][0-9]\)$/,\1/'
```

This results in the following output:

```
city;province;population
Amsterdam;North_Holland;905,234
Rotterdam;South_Holland;656,050
The Hague;South_Holland;552,995
Utrecht;Utrecht;361,924
Eindhoven;North_Brabant;238,478
```

```
Groningen;Groningen;234,649
Tilburg;North_Brabant;224,702
Almere;Flevoland;218,096
Breda;North_Brabant;184,716
Nijmegen;Gelderland;179,073
```

DuckDB: `httpfs` and `regexp_replace` In DuckDB, we use the following query:

```
COPY (
    SELECT
        city,
        replace(province, ' ', '_') AS province,
        regexp_replace(population::VARCHAR, '([0-9][0-9][0-9])$', ',\1')
            AS population
    FROM 'https://duckdb.org/data/cli/pop.csv'
  ) TO '/dev/stdout/' (DELIMITER ';');
```

Note that the FROM clause now has an HTTPS URL instead of a simple CSV file. The presence of the `https://` prefix triggers DuckDB to load the `httpfs` extension and use it to fetch the JSON document. We use the `replace` function to substitute the spaces with underscores, and the `regexp_replace` function for the replacement using a regular expression. (We could have also used string formatting functions such as `format` and `printf`). To change the separator to a semicolon, we serialize the file using the COPY statement with the `DELIMITER ';'` option.

Reading JSON

As a final exercise, let's query the number of stars given to the duckdb/duckdb repository on GitHub.

Unix Shell: `curl` and `jq` In Unix tools, we can use `curl` to get the JSON file from `https://api.github.com` and pipe its output to `jq` to query the JSON object.

```
curl -s https://api.github.com/repos/duckdb/duckdb \
    | jq ".stargazers_count"
```

DuckDB: `read_json` In DuckDB, we use the `read_json` function, invoking it with the remote HTTPS endpoint's URL. The schema of the JSON file is detected automatically, so we can simply use SELECT to return the required field.

```
SELECT stargazers_count
  FROM read_json('https://api.github.com/repos/duckdb/duckdb');
```

Output Both of these commands return the current number of stars of the repository.

Performance

At this point, you might be wondering about the performance of the DuckDB solutions. After all, all of our prior examples have only consisted of a few lines, so benchmarking them against each other will not result in any measurable performance differences. So, let's switch to the Dutch railway services dataset that we used in a previous blog post and formulate a different problem.

We'll use the 2023 railway services file (`services-2023.csv.gz`) and count the number of Intercity services that operated in that year.

In Unix, we can use the `gzcat` command to decompress the `csv.gz` file into a pipeline. Then, we can use `grep` or `pcregrep` (which is more performant), and top it off with the `wc` command to count the number of lines (`-l`). In DuckDB, the built-in CSV reader also supports compressed CSV files, so we can use that without any extra configuration.

```
gzcat services-2023.csv.gz | grep '^[^,]*,[^,]*,Intercity,' | wc -l
gzcat services-2023.csv.gz | pcregrep '^[^,]*,[^,]*,Intercity,' | wc -l
duckdb -c "SELECT count(*) FROM 'services-2023.csv.gz' WHERE \"Service:Type\" = 'Intercity';"
```

We also test the tools on uncompressed input:

```
gunzip -k services-2023.csv.gz
grep '^[^,]*,[^,]*,Intercity,' services-2023.csv | wc -l
pcregrep '^[^,]*,[^,]*,Intercity,' services-2023.csv | wc -l
duckdb -c "SELECT count(*) FROM 'services-2023.csv' WHERE \"Service:Type\" = 'Intercity';"
```

To reduce the noise in the measurements, we used the `hyperfine` benchmarking tool and took the mean execution time of 10 runs. The experiments were carried out on a MacBook Pro with a 12-core M2 Pro CPU and 32 GB RAM, running macOS Sonoma 14.5. To reproduce them, run the `grep-vs-duckdb-microbenchmark.sh` script. The following table shows the runtimes of the solutions on both compressed and uncompressed inputs:

Tool	Runtime (compressed)	Runtime (uncompressed)
grep 2.6.0-FreeBSD	20.9 s	20.5 s
pcregrep 8.45	3.1 s	2.9 s
DuckDB 1.0.0	4.2 s	1.2 s

The results show that on compressed input, `grep` was the slowest, while DuckDB is slightly edged out by `gzcat+pcregrep`, which ran in 3.1 seconds compared to DuckDB's 4.2 seconds. On uncompressed input, DuckDB can utilize all CPU cores from the get-go (instead of starting with a single-threaded decompression step), allowing it to outperform both `grep` and `pcregrep` by a significant margin: 2.5× faster than `pcregrep` and more than 15× faster than `grep`.

While this example is quite simple, as queries get more complex, there are more opportunities for optimization and larger intermediate dataset may be produced. While both of these can be tackled within a shell script (by manually implementing optimizations and writing the intermediate datasets to disk), these will likely be less efficient than what a DBMS can come up with. Shell scripts implementing complex pipelines can also be very brittle and need to be rethought even for small changes, making the performance advantage of using a database even more significant for more complex problems.

Summary

In this post, we used DuckDB as a standalone CLI application, and explored its abilities to complement or substitute existing command line tools (`sort`, `grep`, `comm`, `join`, etc.). While we obviously like DuckDB a lot and prefer to use it in many cases, we also believe Unix tools have their place: on most systems, they are already pre-installed and a well-chosen toolchain of Unix commands *can* be fast, efficient, and portable (thanks to POSIX-compliance). Additionally, they can be very concise for certain problems. However, to reap their benefits, you will need to learn the syntax and quirks of each tool such as `grep` variants, awk as well as advanced ones such as `xargs` and `parallel`. In the meantime, DuckDB's SQL is easy-to-learn (you likely know quite a bit of it already) and DuckDB handles most of the optimization for you.

If you have a favorite CLI use case for DuckDB, let us know on social media or submit it to DuckDB snippets. Happy hacking!

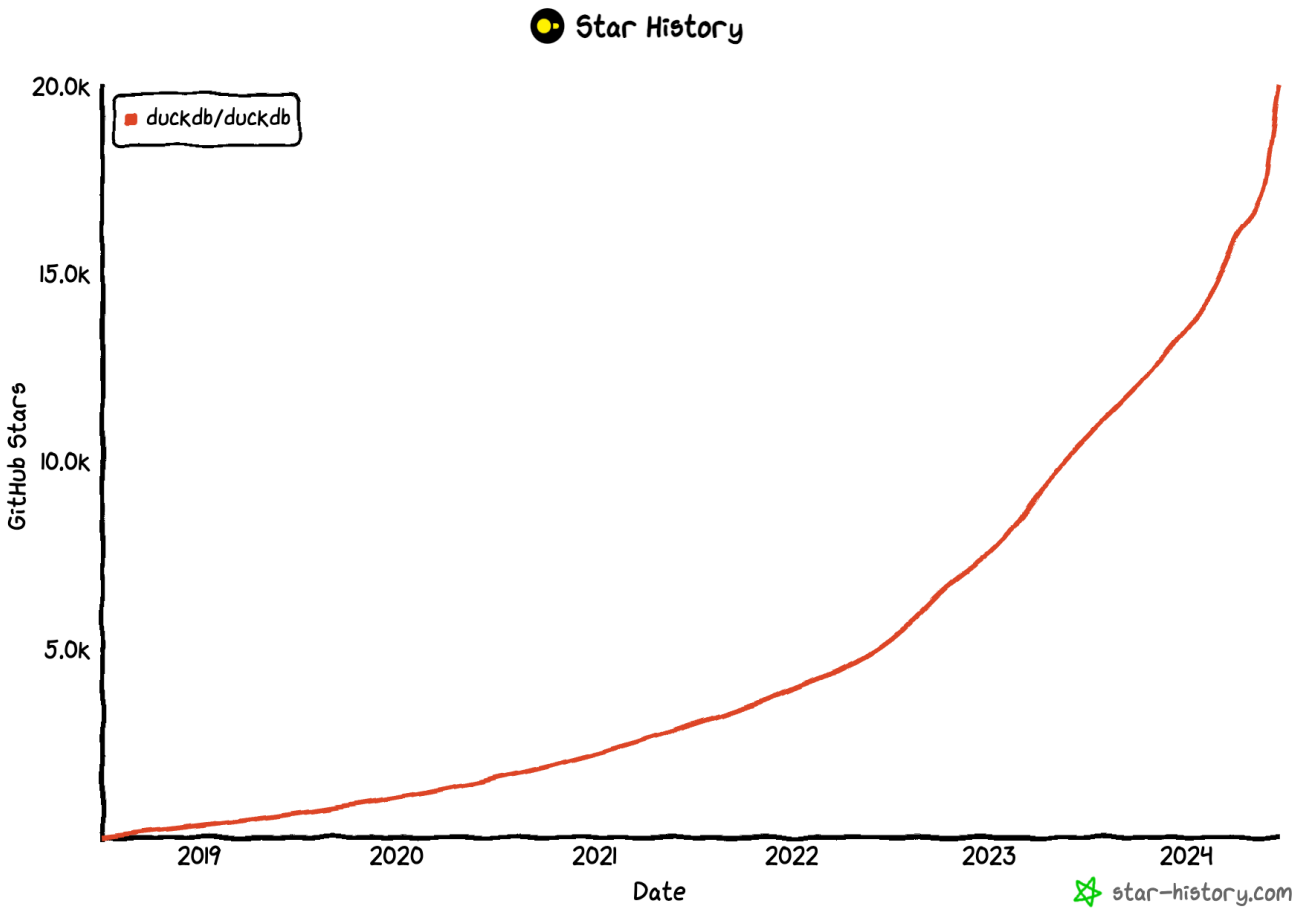
20 000 Stars on GitHub

Publication date: 2024-06-22

Author: The DuckDB team

DuckDB reached 20 000 stars today on GitHub. We would like to thank our amazing community of users and contributors.

To this day, we continue to be amazed by the adoption of DuckDB. We hit the previous milestone, 10 000 stars just a little over a year ago. Since then, the growth of stars has been slowly increasing, until the release of version 1.0.0 in early June gave the boost that propelled the star count to 20 000.



(image source: star-history.com)

What Else Happened in June?

The last few weeks since the release were quite eventful:

1. MotherDuck, a DuckDB-based cloud warehouse, just reached General Availability last week. Congratulations to the team on the successful release!

2. We added support to DuckDB for Delta Lake, an open-source lakehouse framework. This feature was described in Sam Ansmink's blog post and Hannes Mühleisen's keynote segment at the DATA+AI summit.

   With extensions for both Delta Lake and Iceberg, DuckDB can now read the two most popular data lake formats.

3. We ran a poster campaign for DuckDB in Amsterdam:

4. DuckLabs sponsored the Hack4Her event, a female-focused student hackathon in the Netherlands. During the DuckDB Challenge of the event, teams built a community-driven app providing safe walking routes in Amsterdam using DuckDB and its geospatial library.

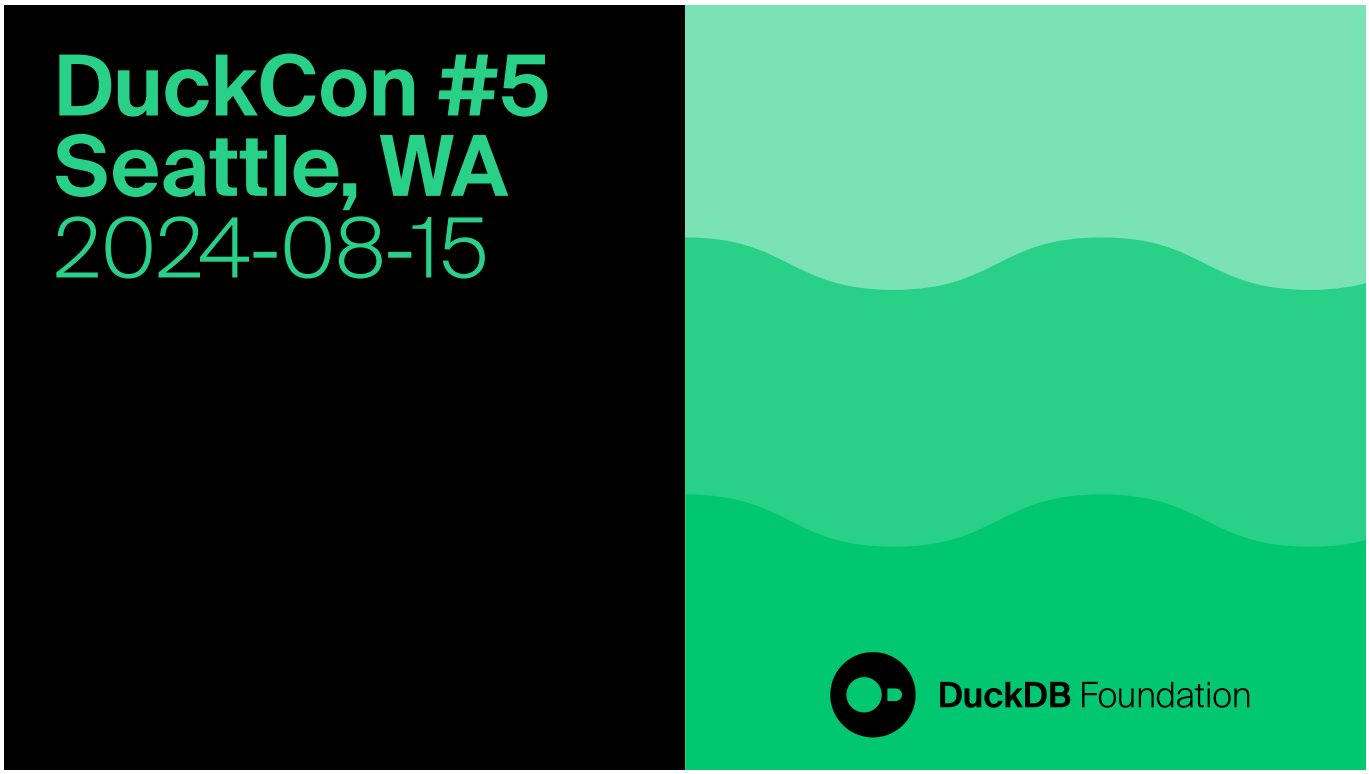
Looking Ahead

There are several interesting events lined up for the summer.

First, two books about DuckDB are expected to be released:

- **Getting Started with DuckDB**, authored by Simon Aubury and Ned Letcher, and published by Packt Publishing
- **DuckDB in Action**, authored by Mark Needham, Michael Hunger and Michael Simons, and published by Manning Publications

Second, we are holding our next user community meeting, DuckCon #5 in Seattle on August 15 with the regular “State of the Duck” update as well as three regular talks and several lightning talks.



Third, we will improve DuckDB's extension ecosystem and streamline the publication process for community extensions.

Finally, we have a series of blog posts lined up for publication. These will discuss DuckDB's performance over time, the results of the user survey we conducted during the spring, DuckDB's storage format, and many more. Stay tuned!

We are looking forward to the next part of our journey and, of course, the next 10 000 stars on GitHub.

Benchmarking Ourselves over Time at DuckDB

Publication date: 2024-06-26

Author: Alex Monahan

TL;DR: In the last 3 years, DuckDB has become 3-25× faster and can analyze ~10× larger datasets all on the same hardware.

A big part of DuckDB's focus is on the developer experience of working with data. However, performance is an important consideration when investigating data management systems. Fairly comparing data processing systems using benchmarks is very difficult. Whoever creates the benchmark is likely to know one system better than the rest, influencing benchmark selection, how much time is spent tuning parameters, and more.

Instead, this post focuses on benchmarking *our own* performance over time.

This approach avoids many comparison pitfalls, and also provides several valuable data points to consider when selecting a system.

- **How fast is it improving?** Learning a new tool is an investment. Picking a vibrant, rapidly improving database ensures your choice pays dividends for years to come. Plus, if you haven't experimented with a tool in a while, you can see how much faster it has become since you last checked!
- **What is it especially good at?** The choice of benchmark is an indicator of what types of workloads a tool is useful for. The higher the variety of analyses in the benchmark, the more broadly useful the tool can be.
- **What scale of data can it handle?** Many benchmarks are deliberately smaller than typical workloads. This allows the benchmark to complete in a reasonable amount of time when run with many configurations. However, an important question to answer when selecting a system is whether the size of your data can be handled within the size of your compute resources.

There are some limitations when looking at the performance of a system over time. If a feature is brand new, there is no prior performance to compare to! As a result, this post focuses on fundamental workloads rather than DuckDB's ever-increasing set of integrations with different lakehouse data formats, cloud services, and more.

The code used to run the benchmark also avoids many of DuckDB's Friendlier SQL additions, as those have also been added more recently. (When writing these queries, it felt like going back in time!)

Benchmark Design Summary

This post measures DuckDB's performance over time using the H2O.ai benchmark, plus some new benchmarks added for importing, exporting, and using window functions. Please see our previous blog posts for details on why we believe the H2O.ai benchmark is a good approach! The full details of the benchmark design are in the appendix.

- H2O.ai, plus import/export and window function tests
- Python instead of R
- 5 GB scale for everything, plus 50 GB scale for group bys and joins
- Median of 3 runs
- Using a MacBook Pro M1 with 16 GB RAM
- DuckDB versions 0.2.7 through 1.0.0
  - Nearly 3 years, from 2021-06-14 to 2024-06-03
- Default settings
- Pandas pre-version 0.5.1, Apache Arrow 0.5.1+

Overall Benchmark Results

The latest DuckDB can complete one run of the full benchmark suite in under 35 seconds, while version 0.2.7 required nearly 500 seconds for the same task in June 2021. **That is 14 times faster, in only 3 years!**

Performance over Time

> **Note.** These graphs are interactive, thanks to Plotly.js! Feel free to filter the various series (single click to hide, double click to show only that series) and click-and-drag to zoom in. Individual benchmark results are visible on hover.

The above plot shows the median runtime in seconds for all tests. Due to the variety of uses for window functions, and their relative algorithmic complexity, the 16 window function tests require the most time of any category.

This plot normalizes performance to the latest version of DuckDB to show relative improvements over time. If you look at the point in time when you most recently measured DuckDB performance, that number will show you how many times faster DuckDB is now!

A portion of the overall improvement is DuckDB's addition of multi-threading, which became the default in November 2021 with version 0.3.1. DuckDB also moved to a push-based execution model in that version for additional gains. Parallel data loading boosted performance in December 2022 with version 0.6.1, as did improvements to the core `JOIN` algorithm. We will explore other improvements in detail later in the post.

However, we see that all aspects of the system have seen improvements, not just raw query performance! DuckDB focuses on the entire data analysis workflow, not just aggregate or join performance. CSV parsing has seen significant gains, import and export have improved significantly, and window functions have improved the most of all.

What was the slight regression from December 2022 to June 2023? Window functions received additional capabilities and experienced a slight performance degradation in the process. However, from June 2023 onward we see substantial performance improvement across the board for window functions. If window functions are filtered out of the chart, we see a smoother trend.

You may also notice that starting with version 0.9 in September 2023, the performance appears to plateau. What is happening here? First, don't forget to zoom in! Over the last year, DuckDB has still improved over 3×! More recently, the DuckLabs team focused on scalability by developing algorithms that support larger-than-memory calculations. We will see the fruits of those labors in the scale section later on! In addition, DuckDB focused exclusively on bug fixes in versions 0.10.1, 0.10.2, and 0.10.3 in preparation for an especially robust DuckDB 1.0. Now that those two major milestones (larger than memory calculations and DuckDB 1.0) have been accomplished, performance improvements will resume! It is worth noting that the boost from moving to multi-threading will only occur once, but there are still many opportunities moving forward.

Performance by Version

We can also recreate the overall plot by version rather than by time. This demonstrates that DuckDB has been doing more frequent releases recently. See DuckDB's release calendar for the full version history.

If you remember the version that you last tested, you can compare how much faster things are now with 1.0!

Results by Workload

CSV Reader

DuckDB has invested substantially in building a fast and robust CSV parser. This is often the first task in a data analysis workload, and it tends to be undervalued and underbenchmarked. DuckDB has **improved CSV reader performance by nearly 3×**, while adding the ability to handle many more CSV dialects automatically.

Group By

Group by or aggregation operations are critical steps in OLAP workloads, and have therefore received substantial focus in DuckDB, **improving over 12× in the last 3 years**.

In November 2021, version 0.3.1 enabled multithreaded aggregation by default, providing a significant speedup.

In December 2022, data loads into tables were parallelized with the release of version 0.6.1. This is another example of improving the entire data workflow, as this group by benchmark actually stressed the insertion performance substantially. Inserting the results was taking the majority of the time!

Enums were also used in place of strings for categorical columns in version 0.6.1. This means that DuckDB was able to use integers rather than strings when operating on those columns, further boosting performance.

Despite what appears at first glance to be a performance plateau, zooming in to 2023 and 2024 reveals a ~20% improvement. In addition, aggregations have received significant attention in the most recent versions to enable larger-than-memory aggregations. You can see that this was achieved while continuing to improve performance for the smaller-than-memory case.

Join

Join operations are another area of focus for analytical databases, and DuckDB in particular. Join speeds have **improved by 4× in the last 3 years**!

Version 0.6.1 in December 2022 introduced improvements to the out-of-core hash join that actually improved the smaller-than-memory case as well. Parallel data loading from 0.6.1 also helps in this benchmark as well, as some results are the same size as the input table.

In recent versions, joins have also been upgraded to support larger-than-memory capabilities. This focus has also benefitted the smaller-than-memory case and has led to the improvements in 0.10, launched in February 2024.

Window Functions

Over the time horizon studied, window functions have **improved a dramatic 25×**!

Window function performance was improved substantially with the 0.9.0 release in September 2023. 14 different performance optimizations contributed. Aggregate computation was vectorized (with special focus on the segment tree data structure). Work stealing enabled multi-threaded processing and sorting was adapted to run in parallel. Care was also taken to pre-allocate memory in larger batches.

DuckDB's window functions are also capable of processing larger-than-memory datasets. We leave benchmarking that feature for future work!

Export

Often DuckDB is not the final step in a workflow, so export performance has an impact. Exports are **10× faster now!** Until recently, the DuckDB format was not backward compatible, so the recommended long term persistence format was Parquet. Parquet is also critical to interoperability with many other systems, especially data lakes. DuckDB works well as a workflow engine, so exporting to other in-memory formats is quite common as well.

In the September 2022 release (version 0.5.1) we see significant improvements driven by switching from Pandas to Apache Arrow as the recommended in-memory export format. DuckDB's underlying data types share many similarities with Arrow, so data transfer is quite quick.

Parquet export performance has improved by 4–5× over the course of the benchmark, with dramatic improvements in versions 0.8.1 (June 2023) and 0.10.2 (April 2024). Version 0.8.1 added parallel Parquet writing while continuing to preserve insertion order.

The change driving the improvement in 0.10.2 was more subtle. When exporting strings with high cardinality, DuckDB decides whether or not to do dictionary compression depending on if it reduces file size. From 0.10.2 onward, the compression ratio is tested after a sample of the values are inserted into the dictionary, rather than after all values are added. This prevents substantial unnecessary processing for high-cardinality columns where dictionary compression is unhelpful.

Exporting Apache Arrow vs. Pandas vs. Parquet This plot shows the performance of all three export formats over the entire time horizon (rather than picking the winner between Pandas and Arrow). It allows us to see at what point Apache Arrow passes Pandas in performance.

Pandas export performance has improved substantially over the course of the benchmark. However, Apache Arrow has proven to be the more efficient data format, so Arrow is now preferred for in-memory exports. Interestingly, DuckDB's Parquet export is now so efficient that it is faster to write a persistent Parquet file than it is to write to an in-memory Pandas dataframe! It is even competitive with Apache Arrow.

Scan Other Formats

In some use cases, DuckDB does not need to store the raw data, but instead should simply read and analyze it. This allows DuckDB to fit seamlessly into other workflows. This benchmark measures how fast DuckDB can scan and aggregate various data formats.

To enable comparisons over time, we switch from Pandas to Arrow at version 0.5.1 as mentioned. DuckDB is **over 8× faster in this workload**, and the absolute time required is very short. DuckDB is a great fit for this type of work!

Scanning Apache Arrow vs. Pandas vs. Parquet Once again, we examine all three formats over the entire time horizon.

When scanning data, Apache Arrow and Pandas are more comparable in performance. As a result, while Arrow is clearly preferable for exports, DuckDB will happily read Pandas with similar speed. However, in this case, the in-memory nature of both Arrow and Pandas allow them to perform 2–3× faster than Parquet. In absolute terms, the time required to complete this operation is a very small fraction of the benchmark, so other operations should be the deciding factor.

Scale tests

Analyzing larger-than-memory data is a superpower for DuckDB, allowing it to be used for substantially larger data analysis tasks than were previously possible.

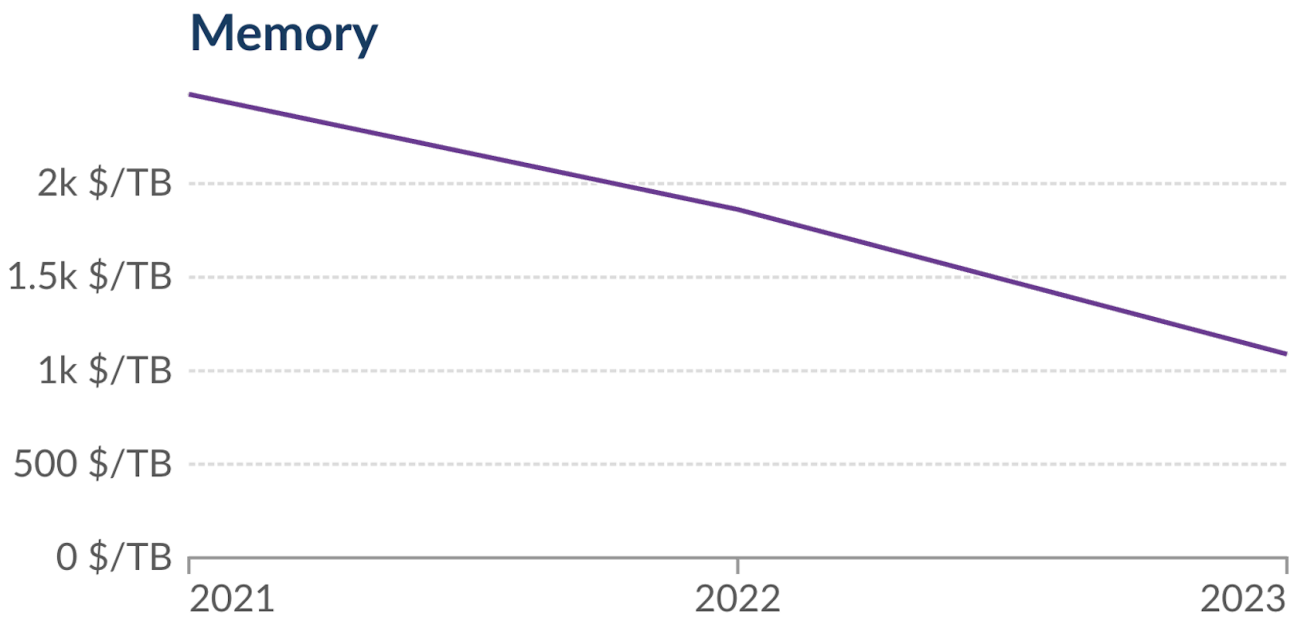
In version 0.9.0, launched in September 2023, DuckDB's hash aggregate was enhanced to handle out-of-core (larger than memory) intermediates. The details of the algorithm, along with some benchmarks, are available in this blog post. This allows for DuckDB to aggregate one billion rows of data (50 GB in size) on a MacBook Pro with only 16 GB of RAM, even when the number of unique groups in the group by is large. This represents at least a 10× improvement in aggregate processing scale over the course of the 3 years of the benchmark.

DuckDB's hash join operator has supported larger-than-memory joins since version 0.6.1 in December 2022. However, the scale of this benchmark (coupled with the limited RAM of the benchmarking hardware), meant that this benchmark could still not complete successfully. In version 0.10.0, launched in February 2024, DuckDB's memory management received a significant upgrade to handle multiple concurrent operators all requiring significant memory. The 0.10.0 release blog post shares additional details about this feature.

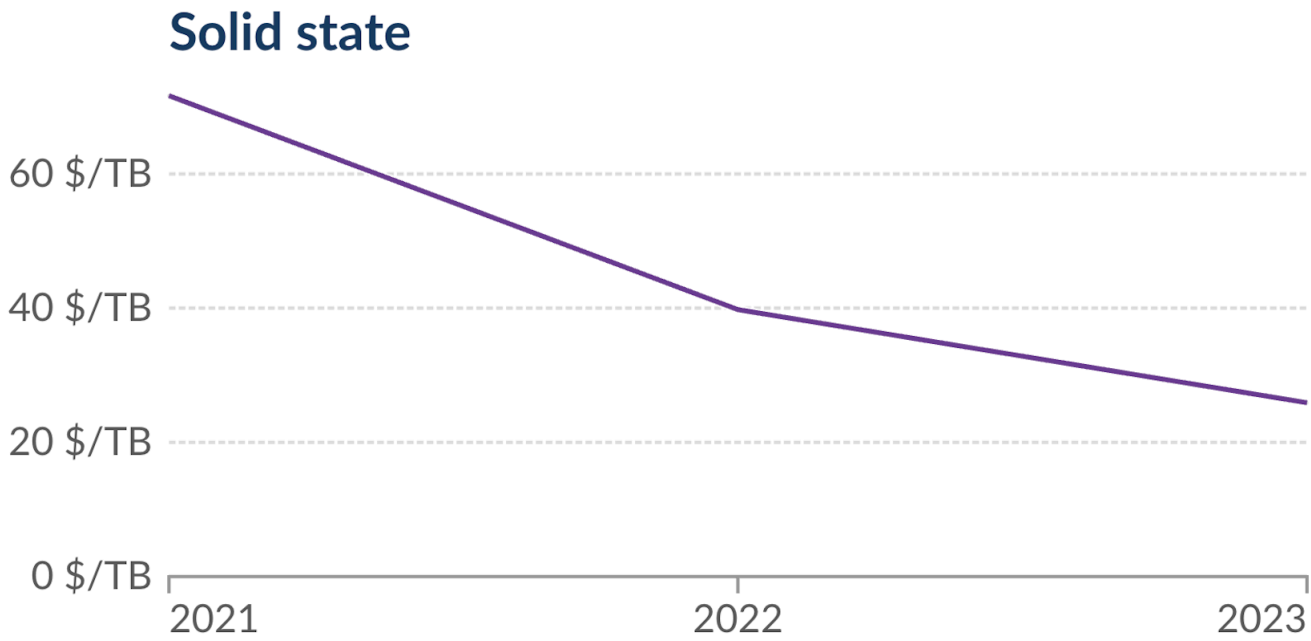
As a result, by version 0.10.0 DuckDB was able to handle calculations on data that is significantly larger than memory, even if the intermediate calculations are large in size. All operators are supported, including sorting, aggregating, joining, and windowing. Future work can further test the boundaries of what is possible with DuckDB's out-of-core support, including window functions and even larger data sizes.

Hardware Capabilities over Time

DuckDB's performance on the same hardware has improved dramatically, and at the same time, the capabilities of hardware are increasing rapidly as well.



{:



width="450" }
width="450" } Source: Our World in Data

The price of RAM has declined by 2.2× and the price of SSD storage has decreased by 2.7× from 2021 to 2023 alone. Thanks to the combination of DuckDB enhancements and hardware prices, the scale of analysis possible on a single node has increased by substantially more than an order of magnitude in just 3 years!

Analyzing the Results Yourself

A DuckDB 1.0 database containing the results of these benchmarks is available at https://blobs.duckdb.org/data/duckdb_perf_over_time.duckdb. Any DuckDB client with the `httpfs` extension can read that file.

You can even use the DuckDB Wasm web shell to **query the file directly from your browser** (with the queries pre-populated and automatically executed!):

```
LOAD httpfs;
ATTACH 'https://blobs.duckdb.org/data/duckdb_perf_over_time.duckdb' AS performance_results;
USE performance_results;
```

The file contains two tables: `benchmark_results` and `scale_benchmark_results`. Please let us know if you uncover any interesting findings!

Conclusion

In summary, not only is DuckDB's feature set growing substantially with each release, DuckDB is getting faster very fast! Overall, performance has **improved by 14× in only 3 years!**

Yet query performance is only part of the story! The variety of workloads that DuckDB can handle is wide and growing wider thanks to a full-featured SQL dialect, including high performance window functions. Additionally, critical workloads like data import, CSV parsing, and data export have improved dramatically over time. The complete developer experience is critical for DuckDB!

Finally, DuckDB now supports larger-than-memory calculations across all operators: sorting, aggregating, joining, and windowing. The size of problem that you can handle on your current compute resources just got 10× bigger, or more!

If you have made it this far, welcome to the flock! 🦆 Join us on Discord, we value your feedback!

Appendix

Benchmark Design

H2O.ai as the Foundation This post measures DuckDB's performance over time on the H2O.ai benchmark for both joins and group by queries.

The result of each H2O.ai query is written to a table in a persistent DuckDB file. This does require additional work when compared with an in-memory workflow (especially the burden on the SSD rather than RAM), but improves scalability and is a common approach for larger analyses.

As in the current H2O.ai benchmark, categorical-type columns (`VARCHAR` columns with low cardinality) were converted to the `ENUM` type as a part of the benchmark. The time for converting into `ENUM` columns was included in the benchmark time, and resulted in a lower total amount of time (so the upfront conversion was worthwhile). However, the `ENUM` data type was not fully operational in DuckDB until version 0.6.1 (December 2022), so earlier versions skip this step.

Python Client To measure interoperability with other dataframe formats, we have used Python rather than R (used by H2O.ai) for this analysis. We do continue to use R for the data generation step for consistency with the benchmark. Python is DuckDB's most popular client, great for data science, and also the author's favorite language for this type of work.

Export and Replacement Scans We now extend this benchmark in several important ways. In addition to considering raw query performance, we measure import and export performance with several formats: Pandas, Apache Arrow, and Apache Parquet. The results of both the join and group by benchmarks are exported to each format.

When exporting to dataframes, we measured the performance in both cases. However, when summarizing the total performance, we chose the best performing format at the time. This likely mirrors the behavior of performance-sensitive users (as they would likely not write to both formats!). In version 0.5.1, released September 2022, DuckDB's performance when writing to and reading from the Apache Arrow format surpassed Pandas. As a result, versions 0.2.7 to 0.4.0 use Pandas, and 0.5.1 onward uses Arrow.

On the import side, replacement scans allow DuckDB to read those same formats without a prior import step. In the replacement scan benchmark, the data that is scanned is the output of the final H2O.ai group by benchmark query. At the 5 GB scale it is a 100 million row dataset. Only one column is read, and a single aggregate is calculated. This focuses the benchmark on the speed of scanning the data rather than DuckDB's aggregation algorithms or speed of outputting results. The query used follows the format:

```
SELECT
    sum(v3) AS v3
FROM <dataframe_or_Parquet_file>;
```

Window Functions We also added an entire series of window function benchmarks. Window functions are a critical workload in real world data analysis scenarios, and can stress test a system in other ways. DuckDB has implemented state of the art algorithms to quickly process even the most complex window functions. We use the largest table from the join benchmark as the raw data for these new tests to help with comparability to the rest of the benchmark.

Window function benchmarks are much less common than more traditional joins and aggregations, and we were unable to find a suitable suite off the shelf. These queries were designed to showcase the variety of uses for window functions, but there are certainly more that could be added. We are open to your suggestions for queries to add, and hope these queries could prove useful for other systems!

Since the window functions benchmark is new, the window functions from each of the queries included are shown in the appendix at the end of the post.

Workload Size We test only the middle 5 GB dataset size for the workloads mentioned thus far, primarily because some import and export operations to external formats like Pandas must fit in memory (and we used a MacBook Pro M1 with only 16 GB of RAM). Additionally, running the tests for 21 DuckDB versions was time-intensive even at that scale, due to the performance of older versions.

Scale Tests Using only 5 GB of data does not answer our second key question: "What scale of data can it handle?"! We also ran only the group by and join related operations (avoiding in-memory imports and exports) at the 5 GB and the 50 GB scale. Older versions of DuckDB could not handle the 50 GB dataset when joining or aggregating, but modern versions can handle both, even on a memory-constrained 16 GB RAM laptop. Instead of measuring performance, we measure the size of the benchmark that was able to complete on a given version.

Summary Metrics With the exception of the scale tests, each benchmark was run 3 times and the median time was used for reporting results. The scale tests were run once and produced a binary metric, success or failure, at each data size tested. As older versions would not fail gracefully, the scale metrics were accumulated across multiple partial runs.

Computing Resources All tests use a MacBook Pro M1 with 16 GB of RAM. In 2024, this is far from state of the art! If you have more powerful hardware, you will see both improved performance and scalability.

DuckDB Versions Version 0.2.7, published in June 2021, was the first version to include a Python client compiled for ARM64, so it was the first version that could easily run on the benchmarking compute resources. Version 1.0.0 is the latest available at the time of publication (June 2024), although we also provide a sneak preview of an in-development feature branch.

Default Settings All versions were run with the default settings. As a result, improvements from a new feature only appear in these results once that feature became the default and was therefore ready for production workloads.

Window Functions Benchmark

Each benchmark query follows the format below, but with different sets of window functions in the `<window_function(s)>{:.language-sql .highlight}` placeholder. The table in use is the largest table from the H2O.ai join benchmark, and in this case the 5 GB scale was used.

```
DROP TABLE IF EXISTS windowing_results;

CREATE TABLE windowing_results AS
    SELECT
        id1,
        id2,
        id3,
```

```
        v2,
        <window_function(s)>
    FROM join_benchmark_largest_table;
```

The various window functions that replace the placeholder are below and are labelled to match the result graphs. These were selected to showcase the variety of use cases for window functions, as well as the variety of algorithms required to support the full range of the syntax. The DuckDB documentation contains a full railroad diagram of the available syntax. If there are common use cases for window functions that are not well-covered in this benchmark, please let us know!

```
/* 302 Basic Window */
sum(v2) OVER () AS window_basic

/* 303 Sorted Window */
first(v2) OVER (ORDER BY id3) AS first_order_by,
row_number() OVER (ORDER BY id3) AS row_number_order_by

/* 304 Quantiles Entire Dataset */
quantile_cont(v2, [0, 0.25, 0.50, 0.75, 1]) OVER ()
    AS quantile_entire_dataset

/* 305 PARTITION BY */
sum(v2) OVER (PARTITION BY id1) AS sum_by_id1,
sum(v2) OVER (PARTITION BY id2) AS sum_by_id2,
sum(v2) OVER (PARTITION BY id3) AS sum_by_id3

/* 306 PARTITION BY ORDER BY */
first(v2) OVER
    (PARTITION BY id2 ORDER BY id3) AS first_by_id2_ordered_by_id3

/* 307 Lead and Lag */
first(v2) OVER
    (ORDER BY id3 ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
    AS my_lag,
first(v2) OVER
    (ORDER BY id3 ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
    AS my_lead

/* 308 Moving Averages */
avg(v2) OVER
    (ORDER BY id3 ROWS BETWEEN 100 PRECEDING AND CURRENT ROW)
    AS my_moving_average,
avg(v2) OVER
    (ORDER BY id3 ROWS BETWEEN id1 PRECEDING AND CURRENT ROW)
    AS my_dynamic_moving_average

/* 309 Rolling Sum */
sum(v2) OVER
    (ORDER BY id3 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS my_rolling_sum

/* 310 RANGE BETWEEN */
sum(v2) OVER
    (ORDER BY v2 RANGE BETWEEN 3 PRECEDING AND CURRENT ROW)
    AS my_range_between,
sum(v2) OVER
    (ORDER BY v2 RANGE BETWEEN id1 PRECEDING AND CURRENT ROW)
    AS my_dynamic_range_between

/* 311 Quantiles PARTITION BY */
quantile_cont(v2, [0, 0.25, 0.50, 0.75, 1])
    OVER (PARTITION BY id2)
    AS my_quantiles_by_id2

/* 312 Quantiles PARTITION BY ROWS BETWEEN */
first(v2) OVER
```

```
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
    AS my_lag_by_id2,
first(v2) OVER
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
    AS my_lead_by_id2

/* 313 Moving Averages PARTITION BY */
avg(v2) OVER
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN 100 PRECEDING AND CURRENT ROW)
    AS my_moving_average_by_id2,
avg(v2) OVER
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN id1 PRECEDING AND CURRENT ROW)
    AS my_dynamic_moving_average_by_id2

/* 314 Rolling Sum PARTITION BY */
sum(v2) OVER
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS my_rolling_sum_by_id2

/* 315 RANGE BETWEEN PARTITION BY */
sum(v2) OVER
    (PARTITION BY id2 ORDER BY v2 RANGE BETWEEN 3 PRECEDING AND CURRENT ROW)
    AS my_range_between_by_id2,
sum(v2) OVER
    (PARTITION BY id2 ORDER BY v2 RANGE BETWEEN id1 PRECEDING AND CURRENT ROW)
    AS my_dynamic_range_between_by_id2

/* 316 Quantiles PARTITION BY ROWS BETWEEN */
quantile_cont(v2, [0, 0.25, 0.50, 0.75, 1]) OVER
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN 100 PRECEDING AND CURRENT ROW)
    AS my_quantiles_by_id2_rows_between
```

DuckDB Community Extensions

Publication date: 2024-07-05

Author: The DuckDB team

TL;DR: DuckDB extensions can now be published as DuckDB Community Extensions. The repository makes it easier for users to install extensions using the `INSTALL extension_name FROM community` syntax. Extension developers avoid the burdens of compilation and distribution.

> To browse existing community extensions, visit the DuckDB Community Extensions documentation page.

DuckDB Extensions

Design Philosophy

One of the main design goals of DuckDB is *simplicity*, which – to us – implies that the system should be rather nimble, very light on dependencies, and generally small enough to run on constrained platforms like WebAssembly. This goal is in direct conflict with very reasonable user requests to support advanced features like spatial data analysis, vector indexes, connectivity to various other databases, support for data formats, etc. Baking all those features into a monolithic binary is certainly possible and the route some systems take. But we want to preserve DuckDB's simplicity. Also, shipping all possible features would be quite excessive for most users because no use cases require *all* extensions at the same time (the "Microsoft Word paradox", where even power users only use a few features of the system, but the exact set of features vary between users).

To achieve this, DuckDB has a powerful extension mechanism, which allows users to add new functionalities to DuckDB. This mechanism allows for registering new functions, supporting new file formats and compression methods, handling new network protocols, etc. In fact, many of DuckDB's popular features are implemented as extensions: the Parquet reader, the JSON reader, and the HTTPS/S3 connector all use the extension mechanism.

Using Extensions

Since version 0.3.2, we have already greatly simplified the discovery and installation by hosting them on a centralized extension repository. So, for example, to install the spatial extension, one can just run the following commands using DuckDB's SQL interface:

```
INSTALL spatial; -- once
LOAD spatial;    -- on each use
```

What happens behind the scenes is that DuckDB downloads an extension binary suitable to the current operating system and processor architecture (e.g., macOS on ARM64) and stores it in the `~/.duckdb` folder. On each `LOAD`, this file is loaded into the running DuckDB instance, and things happily continue from there. Of course, for this to work, we compile, sign and host the extensions for a rather large and growing list of processor architecture – operating system combinations. This mechanism is already heavily used, currently, we see around six million extension downloads *each week* with a corresponding data transfer volume of around 40 terabytes!

Until now, publishing third-party extensions has been a *difficult process* which required the extension developer to build the extensions in their repositories for a host of platforms. Moreover, they were unable to sign the extensions using official keys, forcing users to use the `allow_unsigned_extensions` option that disables signature checks which is problematic in itself.

DuckDB Community Extensions

Distributing software in a safe way has never been easier, allowing us to reach a wide base of users across pip, conda, cran, npm, brew, etc. We want to provide a similar experience both to users who can easily grab the extension they will want to use, and developers who should not be burdened with distribution details. We are also interested in lowering the bar to package utilities and scripts as a DuckDB extension, empowering users to package useful functionality connected to their area of expertise (or pain points).

We believe that fostering a community extension ecosystem is the next logical step for DuckDB. That's why we're very excited about launching our Community Extensions repository which was announced at the Data + AI Summit.

For users, this repository allows for easy discovery, installation and maintenance of community extensions directly from the DuckDB SQL prompt. For developers, it greatly streamlines the publication process of extensions. In the following, we'll discuss how the new extension repository enhances the experiences of these groups.

User Experience

We are going to use the h3 extension as our example. This extension implements hierarchical hexagonal indexing for geospatial data.

Using the DuckDB Community Extensions repository, you can now install and load the h3 extension as follows:

```
INSTALL h3 FROM community;
LOAD h3;
```

Then, you can instantly start using it. Note that the sample data is 500 MB:

```
SELECT
    h3_latlng_to_cell(pickup_latitude, pickup_longitude, 9) AS cell_id,
    h3_cell_to_boundary_wkt(cell_id) AS boundary,
    count() AS cnt
FROM read_parquet('https://blobs.duckdb.org/data/yellow_tripdata_2010-01.parquet')
GROUP BY cell_id
HAVING cnt > 10;
```

On load, the extension's signature is checked, both to ensure platform and versions are compatible, and to verify that the source of the binary is the Community Extensions repository. Extensions are built, signed and distributed for Linux, macOS, Windows, and WebAssembly. This allows extensions to be available to any DuckDB client using version 1.0.0 and upcoming versions.

The h3 extension's documentation is available at https://duckdb.org/community_extensions/extensions/h3.

Developer Experience

From the developer's perspective, the Community Extensions repository performs the steps required for publishing extensions, including building the extensions for all relevant platforms, signing the extension binaries and serving them from the repository.

For the maintainer of h3, the publication process required performing the following steps:

1. Sending a PR with a metadata file `description.yml` contains the description of the extension:

```
extension:
  name: h3
  description: Hierarchical hexagonal indexing for geospatial data
  version: 1.0.0
  language: C++
  build: cmake
  license: Apache-2.0
  maintainers:
    - isaacbrodsky

repo:
  github: isaacbrodsky/h3-duckdb
  ref: 3c8a5358e42ab8d11e0253c70f7cc7d37781b2ef
```

2. The CI will build and test the extension. The checks performed by the CI are aligned with the `extension-template` repository, so iterations can be done independently.
3. Wait for approval from the DuckDB Community Extensions repository's maintainers and for the build process to complete.

Published Extensions

To show that it's feasible to publish extensions, we reached out to a few developers of key extensions. At the time of the publication of this blog post, the DuckDB Community Extensions repository already contains the following extensions.

Name	Description
crypto	Adds cryptographic hash functions and HMAC.
h3	Implements hierarchical hexagonal indexing for geospatial data.
lindel	Implements linearization/delinearization, Z-Order, Hilbert and Morton curves.
prql	Allows running PRQL commands directly within DuckDB.
scrooge	Supports a set of aggregation functions and data scanners for financial data.
shellfs	Allows shell commands to be used for input and output.

DuckLabs and the DuckDB Foundation do not vet the code within community extensions and, therefore, cannot guarantee that DuckDB community extensions are safe to use. The loading of community extensions can be explicitly disabled with the following one-way configuration option:

```
SET allow_community_extensions = false;
```

For more details, see the documentation's Securing DuckDB page.

Summary and Looking Ahead

In this blog post, we introduced the DuckDB Community Extensions repository, which allows easy installation of third-party DuckDB extensions.

We are looking forward to continuously extending this repository. If you have an idea for creating an extension, take a look at the already published extension source codes, which provide good examples of how to package community extensions, and join the `#extensions` channel on our Discord. Once you have an extension, please contribute it via a pull request.

Finally, we would like to thank the early adopters of DuckDB's extension mechanism and Community Extensions repository. Thanks for iterating with us and providing feedback to us.

Memory Management in DuckDB

Publication date: 2024-07-09

Author: Mark Raasveldt

Memory is an important resource when processing large amounts of data. Memory is a fast caching layer that can provide immense speed-ups to query processing. However, memory is finite and expensive, and when working with large data sets there is generally not enough memory available to keep all necessary data structures cached. Managing memory effectively is critical for a high-performance query engine – as memory must be utilized in order to provide that high performance, but we must be careful so that we do not use excessive memory which can cause out-of-memory errors or can cause the ominous OOM killer to zap the process out of existence.

DuckDB is built to effectively utilize available memory while avoiding running out of memory:

- The streaming execution engine allows small chunks of data to flow through the system without requiring entire data sets to be materialized in memory.
- Data from intermediates can be spilled to disk temporarily in order to free up space in memory, allowing computation of complex queries that would otherwise exceed the available memory.
- The buffer manager caches as many pages as possible from any attached databases without exceeding the pre-defined memory limits.

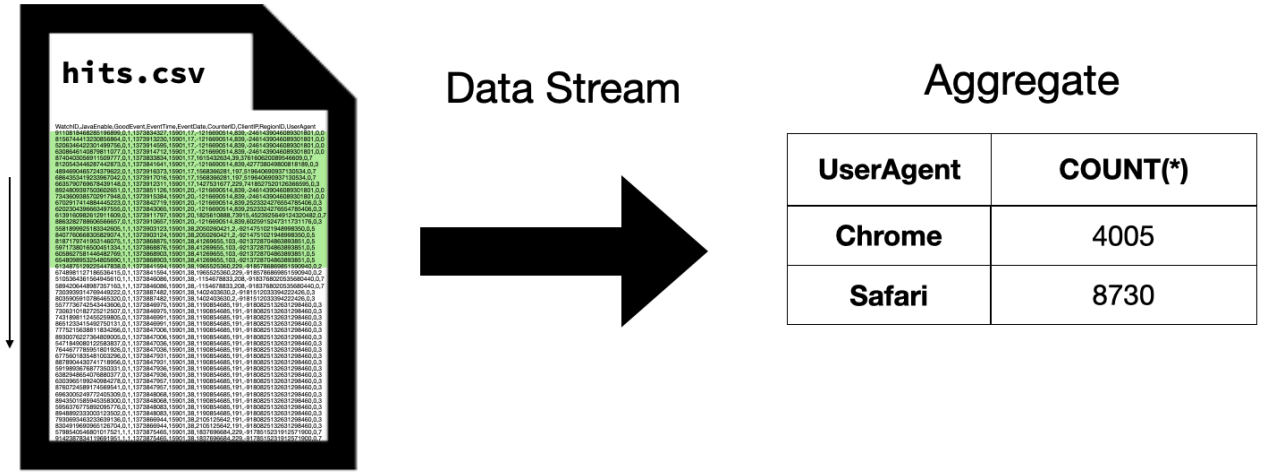
In this blog post we will cover these aspects of memory management within DuckDB – and provide examples of where they are utilized.

Streaming Execution

DuckDB uses a streaming execution engine to process queries. Data sources, such as tables, CSV files or Parquet files, are never fully materialized in memory. Instead, data is read and processed one chunk at a time. For example, consider the execution of the following query:

```
SELECT
    UserAgent,
    count(*)
FROM 'hits.csv'
GROUP BY UserAgent;
```

Instead of reading the entire CSV file at once, DuckDB reads data from the CSV file in pieces, and computes the aggregation incrementally using the data read from those pieces. This happens continuously until the entire CSV file is read, at which point the entire aggregation result is computed.



In the above example we are only showing a single data stream. In practice, DuckDB uses multiple data streams to enable multi-threaded execution – each thread executes its own data stream. The aggregation results of the different threads are combined to compute the final result.

While streaming execution is conceptually simple, it is powerful, and is sufficient to provide larger-than-memory support for many simple use cases. For example, streaming execution enables larger-than-memory support for:

- Computing aggregations where the total number of groups is small
- Reading data from one file and writing to another (e.g., reading from CSV and writing to Parquet)
- Computing a Top-N over the data (where N is small)

Note that nothing needs to be done to enable streaming execution – DuckDB always processes queries in this manner.

Intermediate Spilling

While streaming execution enables larger-than-memory processing for simple queries, there are many cases where streaming execution alone is not sufficient.

In the previous example, streaming execution enabled larger-than-memory processing because the computed aggregate result was very small – as there are very few unique user agents in comparison to the total number of web requests. As a result, the aggregate hash table would always remain small, and never exceed the amount of available memory.

Streaming execution is not sufficient if the intermediates required to process a query are larger than memory. For example, suppose we group by the source IP in the previous example:

```
SELECT
    IPNetworkID,
    count(*)
FROM 'hits.csv'
GROUP BY IPNetworkID;
```

Since there are many more unique source IPs, the hash table we need to maintain is significantly larger. If the size of the aggregate hash table exceeds memory, the streaming execution engine is not sufficient to prevent out-of-memory issues.

Larger-than-memory intermediates can happen in many scenarios, in particular when executing more complex queries. For example, the following scenarios can lead to larger-than-memory intermediates:

- Computing an aggregation with many unique groups
- Computing an exact distinct count of a column with many distinct values
- Joining two tables together that are both larger than memory

- Sorting a larger-than-memory dataset
- Computing a complex window over a larger-than-memory table

DuckDB deals with these scenarios by disk spilling. Larger-than-memory intermediates are (partially) written to disk in the temporary directory when required. While powerful, disk spilling reduces performance – as additional I/O must be performed. For that reason, DuckDB tries to minimize disk spilling. Disk spilling is adaptively used only when the size of the intermediates increases past the memory limit. Even in those scenarios, as much data is kept in memory as possible to maximize performance. The exact way this is done depends on the operators and is detailed in other blog posts (aggregation, sorting).

The `memory_limit` setting controls how much data DuckDB is allowed to keep in memory. By default, this is set to 80% of the physical RAM of your system (e.g., if your system has 16 GB RAM, this defaults to 12.8 GB). The memory limit can be changed using the following command:

```
SET memory_limit = '4GB';
```

The location of the temporary directory can be chosen using the `temp_directory` setting, and is by default the connected database with a `.tmp` suffix (e.g., `database.db.tmp`), or only `.tmp` if connecting to an in-memory database. The maximum size of the temporary directory can be limited using the `max_temp_directory_size` setting, which defaults to 90% of the remaining disk space on the drive where the temporary files are stored. These settings can be adjusted as follows:

```
SET temp_directory = '/tmp/duckdb_swap';
SET max_temp_directory_size = '100GB';
```

If the memory limit is exceeded and disk spilling cannot be used, either because disk spilling is explicitly disabled, the temporary directory size exceeds the provided limit, or a system limitation means that disk spilling cannot be used for a given query – an out-of-memory error is reported and the query is canceled.

Buffer Manager

Another core component of memory management in DuckDB is the buffer manager. The buffer manager is responsible for caching pages from DuckDB's own persistent storage. Conceptually the buffer manager works in a similar fashion to the intermediate spilling. Pages are kept in memory as much as possible, and evicted from memory when space is required for other data structures. The buffer manager abides by the same memory limit as any intermediate data structures. Pages in the buffer manager can be freed up to make space for intermediate data structures, or vice versa.

There are two main differences between the buffer manager and intermediate data structures:

- As the buffer manager caches pages that already exist on disk (in DuckDB's persistent storage) – they do not need to be written to the temporary directory when evicted. Instead, when they are required again, they can be re-read from the attached storage file directly.
- Query intermediates have a natural life-cycle, namely when the query is finished processing the intermediates are no longer required. Pages that are buffer managed from the persistent storage are useful across queries. As such, the pages kept by the buffer manager are kept cached until either the persistent database is closed, or until space must be freed up for other operations.

The performance boost of the buffer manager depends on the speed of the underlying storage medium. When data is stored on a very fast disk, reading data is fast and the speed-up is minimal. When data is stored on a network drive or read over http/S3, reading requires performing network requests, and the speed-up can be very large.

Profiling Memory Usage

DuckDB contains a number of tools that can be used to profile memory usage.

The `duckdb_memory()` function can be used to inspect which components of the system are using memory. Memory used by the buffer manager is labeled as `BASE_TABLE`, while query intermediates are divided into separate groups.

```
FROM duckdb_memory();
```

tag varchar	memory_usage_bytes int64	temporary_storage_bytes int64
BASE_TABLE	168558592	0
HASH_TABLE	0	0
PARQUET_READER	0	0
CSV_READER	0	0
ORDER_BY	0	0
ART_INDEX	0	0
COLUMN_DATA	0	0
METADATA	0	0
OVERFLOW_STRINGS	0	0
IN_MEMORY_TABLE	0	0
ALLOCATOR	0	0
EXTENSION	0	0
12 rows		3 columns

The `duckdb_temporary_files` function can be used to examine the current contents of the temporary directory.

```
FROM duckdb_temporary_files();
```

path varchar	size int64
.tmp/duckdb_temp_storage-0.tmp	967049216

Conclusion

Memory management is critical for a high-performance analytics engine. DuckDB is built to take advantage of any available memory to speed up query processing, while gracefully dealing with larger-than-memory datasets using intermediate spilling. Memory management is still an active area of development and has continuously improved across DuckDB versions. Amongst others, we are working on improving memory management for complex queries that involve multiple operators with larger-than-memory intermediates.

Friendly Lists and Their Buddies, the Lambdas

Publication date: 2024-08-08

Authors: Tania Bogatsch, Maia de Graaf

Introduction

Nested data types, such as lists and structs, are widespread in analytics. Several popular formats, such as Parquet and JSON, support nested types. Traditionally, working with nested types requires normalizing steps before any analysis. Then, to return nested results, systems need to (re-)aggregate their data. Normalization and aggregation are undesirable from both a usability and performance perspective. To streamline the operation on nested data, analytical systems, including DuckDB, provide native functionality on these nested types.

In this blog post, we'll first cover the basics of lists and lambdas. Then, we dive into their technical details. Finally, we'll show some examples from the community. Feel free to skip ahead if you're already familiar with lists and lambdas and are just here for our out-of-the-box examples!

Lists

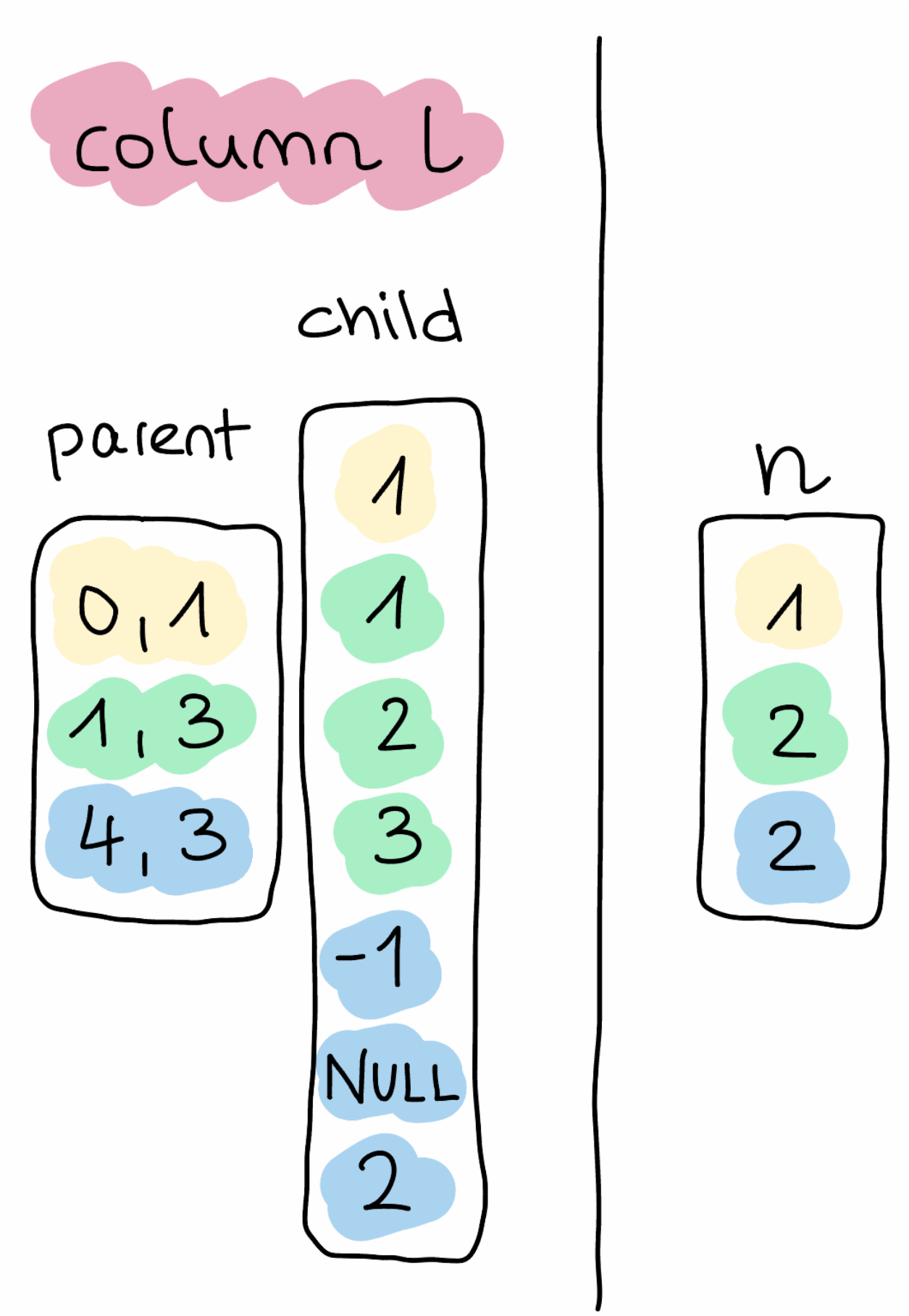
Before jumping into lambdas, let's take a quick detour into DuckDB's `LIST` type. A list contains any number of elements with the same data type. Below is a table containing two columns, `l` and `n`. `l` contains lists of integers, and `n` contains integers.

```
CREATE OR REPLACE TABLE my_lists (l INTEGER[], n INTEGER);
INSERT INTO my_lists VALUES ([1], 1), ([1, 2, 3], 2), ([-1, NULL, 2], 2);
FROM my_lists;
```

l int32[]	n int32
[1]	1
[1, 2, 3]	2
[-1, NULL, 2]	2

Internally, all data moves through DuckDB's execution engine in `Vectors`. For more details on `Vectors` and vectorized execution, please refer to the documentation and respective research papers (1 and 2). In this case, we get two vectors, as depicted below. This representation is mostly similar to Arrow's physical list representation.

When examined closely, we can observe that the nested child vector of `l` looks suspiciously similar to the vector `n`. These nested vector representations enable our execution engine to reuse existing components on nested types. We'll elaborate more on why this is relevant later.


column L
child
parent
n
1
0, 1
1
1, 3
2
4, 3
3
-1
NULL
2
1
2
2

Lambdas

A **lambda function** is an anonymous function, i.e., a function without a name. In DuckDB, a lambda function's syntax is `lambda param1, param2, ...: expression`. The parameters can have any name, and the `expression` can be any SQL expression.

Currently, DuckDB has three scalar functions for working with lambdas: `list_transform`, `list_filter`, and `list_reduce`, along with their aliases. Each accepts a `LIST` as its first argument and a lambda function as its second argument.

Lambdas were the guest star in our SQL Gymnastics: Bending SQL into Flexible New Shapes blog post. This time, we want to put them in the spotlight.

Zooming In: List Transformations

To return to our previous example, let's say we want to add `n` to each element of the corresponding list `l`.

Pure Relational Solution

Using pure relational operators, i.e., avoiding list-native functions, we would need to perform the following steps:

1. Unnest the lists while keeping the connection to their respective rows. We can achieve this by inventing a temporary unique identifier, such as a `rowid` or a `UUID`.
2. Transform each element by adding `n`.
3. Using our temporary identifier `rowid`, we can reaggregate the transformed elements by grouping them into lists.

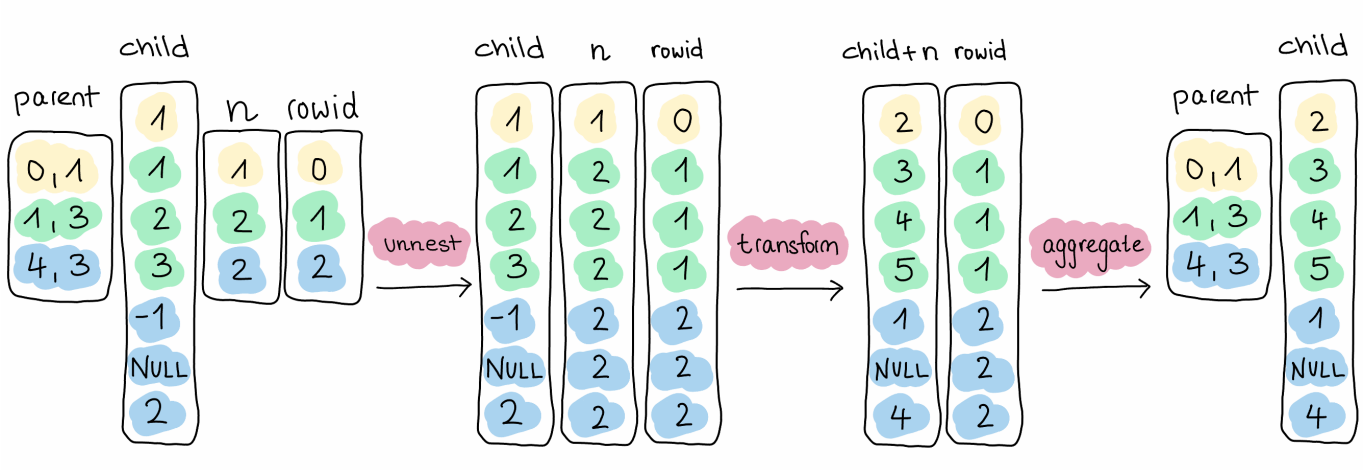
In SQL, it would look like this:

```
WITH flattened_tbl AS (
    SELECT unnest(l) AS elements, n, rowid
    FROM my_lists
)
SELECT array_agg(elements + n) AS result
FROM flattened_tbl
GROUP BY rowid
ORDER BY rowid;
```

```
┌──────────────┐
│    result    │
│   int32[]    │
├──────────────┤
│ [2]          │
│ [3, 4, 5]    │
│ [1, NULL, 4] │
└──────────────┘
```

While the above example is reasonably readable, more complex transformations can become lengthy queries, which are difficult to compose and maintain. More importantly, this query adds an `unnest` operation and an aggregation (`array_agg`) with a `GROUP BY`. Adding a `GROUP BY` can be costly, especially for large datasets.

We have to dive into the technical implications to fully understand why the above query yields suboptimal performance. Internally, the query execution performs the steps depicted in the diagram below. We can directly emit the child vector for the `unnest` operation, i.e., without copying any data. For the correlated columns `rowid` and `n`, we use selection vectors, which again prevents the copying of data. This way, we can fire our expression execution on the child vector, another nested vector, and the expanded vector `n`.



The heavy-hitting operation is the last one, reaggregating the transformed elements into their respective lists. As we don't propagate the parent vector, we have no information about the resulting element's correlation to the initial lists. Recreating these lists requires a full copy of the data and partitioning, which impacts performance even with DuckDB's high-performance aggregation operator.

As a consequence, the normalized approach is both cumbersome to write and it is inefficient as it produces a significant (and unnecessary) overhead despite the relative simplicity of the query. This is yet another example of how shaping nested data into relational forms or forcing it through rectangles can have a significant negative performance impact.

Native List Functions

With support for native list functions, DuckDB mitigates these drawbacks by operating directly on the `LIST` data structure. Since, as we've seen, lists are essentially nested columns, we can reshape these functions into concepts already understood by our execution engine and leverage their full potential.

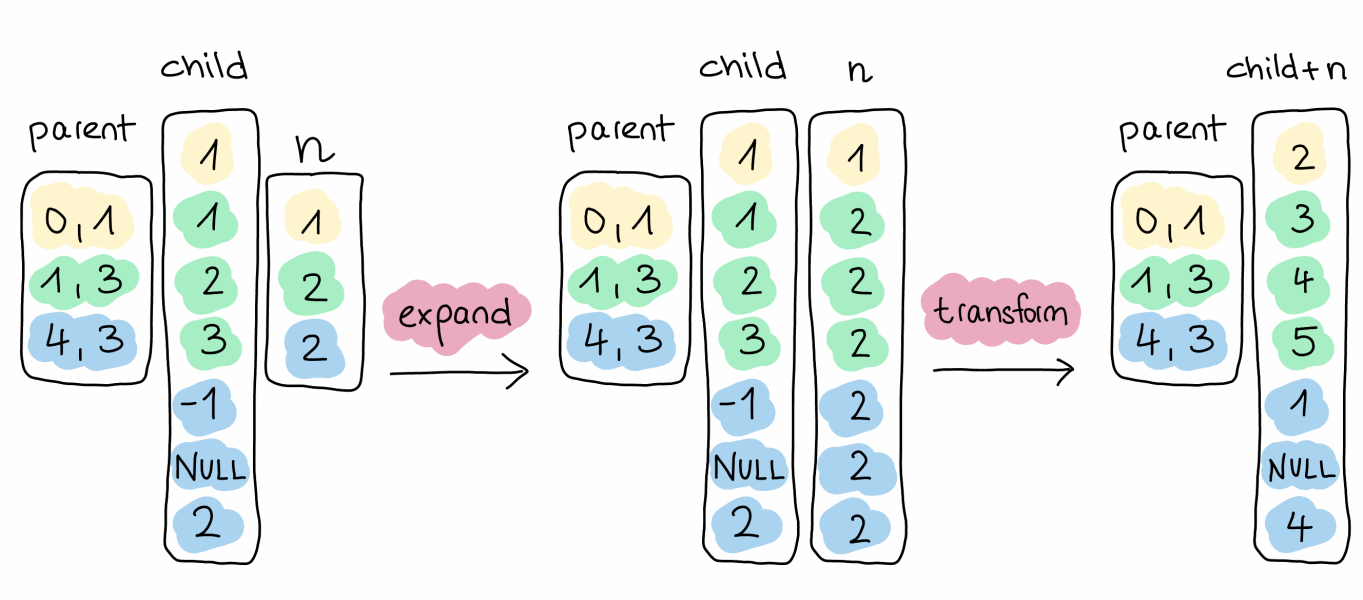
In the case of transformations, the corresponding list-native function is `list_transform`. Here is the rewritten query:

```
SELECT list_transform(l, lambda x: x + n) AS result
FROM my_lists;
```

Alternatively, with Python's list comprehension syntax:

```
SELECT [x + n FOR x IN l] AS result
FROM my_lists;
```

Internally, this query expands all related vectors, which is just n in this case. Just like before, we employ selection vectors to avoid any data copies. Then, we use the lambda function `x: x + n` to fire our expression execution on the child vector and the expanded vector n. As this is a list-native function, we're aware of the existence of a parent vector and keep it alive. So, once we get the result from the transformation, we can completely omit the reaggregation step.



To see the efficiency of `list_transform` in action, we executed a simple benchmark. Firstly, we added 1M rows to our table `my_lists`, each containing five elements.

```
INSERT INTO my_lists
    SELECT [r, r % 10, r + 5, r + 11, r % 2], r
    FROM range(1_000_000) AS tbl(r);
```

Then, we ran both our normalized and list-native queries on this data. Both queries were run in the CLI with DuckDB v1.0.0 on a MacBook Pro 2021 with a M1 Max chip.

Normalized	Native
0.522 s	0.027 s

As we can see, the native query is more than 10× faster. Amazing! If we look at the execution plan using `EXPLAIN ANALYZE` (not shown in this blog post), we can see that DuckDB spends most of its time in the `HASH_GROUP_BY` and `UNNEST` operators. In comparison, these operators no longer exist in the list-native query plan.

Lists and Lambdas in the Community

To better present what's possible by combining our `LIST` type and lambda functions, we've scoured the community Discord and GitHub, as well as some far corners of the internet, for exciting use cases.

`list_transform`

As established earlier, `list_transform` applies a lambda function to each element of the input list and returns a new list with the transformed elements. Here, one of our users implemented a `list_shuffle` function by nesting different `LIST` native functions.

```
CREATE OR REPLACE MACRO list_shuffle(l) AS (
    list_select(l, list_grade_up([random() FOR _ IN l]))
);
```

Another user investigated querying remote Parquet files using DuckDB. In their query, they first use `list_transform` to generate a list of URLs for Parquet files. This is followed by the `read_parquet` function, which reads the Parquet files and calculates the total size of the data. The query looks like this:

```
SELECT
    sum(size) AS size
FROM read_parquet(
    ['https://huggingface.co/datasets/vivym/midjourney-messages/resolve/main/data/' ||
        format('{:06d}', n) || '.parquet'
        FOR n IN generate_series(0, 55)
    ]
);
```

`list_filter`

The `list_filter` function filters all elements of the input list for which the lambda function returns `true`.

Here is an example using `list_filter` from a discussion on our Discord where the user wanted to remove the element at index `idx` from each list.

```
CREATE OR REPLACE MACRO remove_idx(l, idx) AS (
    list_filter(l, lambda _, i: i != idx)
);
```

So far, we've primarily focused on showcasing our lambda function support in this blog post. Yet, there are often many possible paths with SQL and its rich dialects. We couldn't help but show how we can achieve the same functionality with some of our other native list functions. In this case, we used `list_slice` and `list_concat`.

```
CREATE OR REPLACE MACRO remove_idx(l, idx) AS (
    l[:idx - 1] || l[idx + 1:]
);
```

`list_reduce`

Most recently, we've added `list_reduce`, which applies a lambda function to an accumulator value. The accumulator is the result of the previous lambda function and is also what the function ultimately returns.

We took the following example from a discussion on GitHub. The user wanted to use a lambda to validate BSN numbers, the Dutch equivalent of social security numbers. A BSN must be 8 or 9 digits, but to limit our scope we'll just focus on BSNs that are 9 digits long. After multiplying each digit by its index, from 9 down to 2, and the last digit by -1, the sum must be divisible by 11 to be valid.

Setup For our example, we assume that input BSNs are of type `INTEGER[]`.

```
CREATE OR REPLACE TABLE bsn_tbl AS
    FROM VALUES
        ([2, 4, 6, 7, 4, 7, 5, 9, 6]),
        ([1, 2, 3, 4, 5, 6, 7, 8, 9]),
        ([7, 6, 7, 4, 4, 5, 2, 1, 1]),
        ([8, 7, 9, 0, 2, 3, 4, 1, 7]),
        ([1, 2, 3, 4, 5, 6, 7, 8, 9, 0])
        tbl(bsn);
```

Solution

```
CREATE OR REPLACE MACRO valid_bsn(bsn) AS (
    bsn.list_reverse().list_reduce(
        lambda x, y, i: IF (i = 2, -x, x) + y * i
    ) % 11 = 0
    AND len(bsn) = 9
);
```

Using our macro with the example table we get the following result:

```
SELECT bsn, valid_bsn(bsn) AS valid
FROM bsn_tbl;
```

bsn int32[]	valid boolean
[2, 4, 6, 7, 4, 7, 5, 9, 6]	true
[1, 2, 3, 4, 5, 6, 7, 8, 9]	false
[7, 6, 7, 4, 4, 5, 2, 1, 1]	true
[8, 7, 9, 0, 2, 3, 4, 1, 7]	true
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]	false

Conclusion

Native nested type support is critical for analytical systems. As such, DuckDB offers native nested type support and many functions to work with these types directly. These functions make working with nested types easier and substantially faster. In this blog post, we looked at the technical details of working with nested types by diving into our `list_transform` function. Additionally, we highlighted various use cases that we came across in our community.

DuckDB Tricks – Part 1

Publication date: 2024-08-19

Author: Gábor Szárnyas

TL;DR: We use a simple example data set to present a few tricks that are useful when using DuckDB.

In this blog post, we present five simple DuckDB operations that we found particularly useful for interactive use cases. The operations are summarized in the following table:

Operation	Snippet
Pretty-printing floats	`SELECT (10 / 9)::DECIMAL(15, 3)`{:.language-sql .highlight}
Copying the schema	`CREATE TABLE tbl AS FROM example LIMIT 0`{:.language-sql .highlight}
Shuffling data	`FROM example ORDER BY hash(rowid + 42)`{:.language-sql .highlight}
Specifying types when reading CSVs	`FROM read_csv('example.csv', types = {'x': 'DECIMAL(15, 3)'})`{:.language-sql .highlight}
Updating CSV files in-place	`COPY (SELECT s FROM 'example.csv') TO 'example.csv'`{:.language-sql .highlight}

Creating the Example Data Set

We start by creating a data set that we'll use in the rest of the blog post. To this end, we define a table, populate it with some data and export it to a CSV file.

```
CREATE TABLE example (s STRING, x DOUBLE);
INSERT INTO example VALUES ('foo', 10/9), ('bar', 50/7), ('qux', 9/4);
COPY example TO 'example.csv';
```

Wait a bit, that's way too verbose! DuckDB's syntax has several SQL shorthands including the "friendly SQL" clauses. Here, we combine the VALUES clause with the FROM-first syntax, which makes the SELECT clause optional. With these, we can compress the data creation script to ~60% of its original size. The new formulation omits the schema definition and creates the CSV with a single command:

```
COPY (FROM VALUES ('foo', 10/9), ('bar', 50/7), ('qux', 9/4) t(s, x))
TO 'example.csv';
```

Regardless of which script we run, the resulting CSV file will look like this:

```
s,x
foo,1.1111111111111112
bar,7.142857142857143
qux,2.25
```

Let's continue with the code snippets and their explanations.

Pretty-Printing Floating-Point Numbers

When printing a floating-point number to the output, the fractional parts can be difficult to read and compare. For example, the following query returns three numbers between 1 and 8 but their printed widths are very different due to their fractional parts.

```
SELECT x
FROM 'example.csv';
```

```
┌────────────────────┐
│         x          │
│       double       │
├────────────────────┤
│ 1.1111111111111112 │
│  7.142857142857143 │
│               2.25 │
└────────────────────┘
```

By casting a column to a `DECIMAL` with a fixed number of digits after the decimal point, we can pretty-print it as follows:

```
SELECT x::DECIMAL(15, 3) AS x
FROM 'example.csv';
```

```
┌─────────────────┐
│        x        │
│ decimal(15,3)   │
├─────────────────┤
│           1.111 │
│           7.143 │
│           2.250 │
└─────────────────┘
```

A typical alternative solution is to use the `printf` or `format` functions, e.g.:

```
SELECT printf('%.3f', x)
FROM 'example.csv';
```

However, these approaches require us to specify a formatting string that's easy to forget. What's worse, the statement above returns string values, which makes subsequent operations (e.g., sorting) more difficult. Therefore, unless keeping the full precision of the floating-point numbers is a concern, casting to `DECIMAL` values should be the preferred solution for most use cases.

Copying the Schema of a Table

To copy the schema from a table without copying its data, we can use `LIMIT 0`.

```
CREATE TABLE example AS
    FROM 'example.csv';
CREATE TABLE tbl AS
    FROM example
    LIMIT 0;
```

This will result in an empty table with the same schema as the source table:

```
DESCRIBE tbl;
```

column_name varchar	column_type varchar	null varchar	key varchar	default varchar	extra varchar
s	VARCHAR	YES			
x	DOUBLE	YES			

Alternatively, in the CLI client, we can run the `.schema` dot command:

```
.schema
```

This will return the schema of the table.

```
CREATE TABLE example (s VARCHAR, x DOUBLE);
```

After editing the table's name (e.g., `example` to `tbl`), this query can be used to create a new table with the same schema.

Shuffling Data

Sometimes, we need to introduce some entropy into the ordering of the data by shuffling it. To shuffle *non-deterministically*, we can simply sort on a random value provided the `random()` function:

```
FROM 'example.csv' ORDER BY random();
```

Shuffling *deterministically* is a bit more tricky. To achieve this, we can order on the hash, of the `rowid` pseudocolumn. Note that this column is only available in physical tables, so we first have to load the CSV in a table, then perform the shuffle operation as follows:

```
CREATE OR REPLACE TABLE example AS FROM 'example.csv';
FROM example ORDER BY hash(rowid + 42);
```

The result of this shuffle operation is deterministic – if we run the script repeatedly, it will always return the following table:

s varchar	x double
bar	7.142857142857143
qux	2.25
foo	1.1111111111111112

Note that the `+ 42` is only necessary to nudge the first row from its position – as `hash(0)` returns `0`, the smallest possible value, using it for ordering leaves the first row in its place.

Specifying Types in the CSV Loader

DuckDB's CSV loader auto-detects types from a short list of `BOOLEAN`, `BIGINT`, `DOUBLE`, `TIME`, `DATE`, `TIMESTAMP` and `VARCHAR`. In some cases, it's desirable to override the detected type of a given column with a type outside of this list. For example, we may want to treat column `x` as a `DECIMAL` value from the get-go. We can do this on a per-column basis with the `types` argument of the `read_csv` function:

```
CREATE OR REPLACE TABLE example AS
    FROM read_csv('example.csv', types = {'x': 'DECIMAL(15, 3)'});
```

Then, we can simply query the table to see the result:

```
FROM example;
```

s varchar	x decimal(15,3)
foo	1.111
bar	7.143
qux	2.250

Updating CSV Files In-Place

In DuckDB, it is possible to read, process and write CSV files in-place. For example, to project the column `s` into the same file, we can simply run:

```
COPY (SELECT s FROM 'example.csv') TO 'example.csv';
```

The resulting `example.csv` file will have the following content:

```
s
foo
bar
qux
```

Note that this trick is not possible in Unix shells without a workaround. One might be tempted to run the following command on the `example.csv` file and expect the same result:

```
cut -d, -f1 example.csv > example.csv
```

However, due to the intricacies of Unix pipelines, executing this command leaves us with an empty `example.csv` file. The solution is to use different file names, then perform a rename operation:

```
cut -d, -f1 example.csv > tmp.csv && mv tmp.csv example.csv
```

Closing Thoughts

That's it for today. The tricks shown in this post are available on duckdbsnippets.com. If you have a trick that would like to share, please submit it there, or send it to us via social media or Discord. Happy hacking!

Announcing DuckDB 1.1.0

Publication date: 2024-09-09

Author: The DuckDB team

TL;DR: The DuckDB team is happy to announce that today we're releasing DuckDB version 1.1.0, codenamed “Eatoni”.

To install the new version, please visit the installation guide. For the release notes, see the release page.

> Some packages (R, Java) take a few extra days to release due to the reviews required in the release pipelines.

We are proud to release DuckDB 1.1.0, our first release since we released version 1.0.0 three months ago. This release is codenamed “Eatoni” after the Eaton's pintail (Anas eatoni), a dabbling duck that occurs only on two very remote island groups in the southern Indian Ocean.

What's New in 1.1.0

There have been far too many changes to discuss them each in detail, but we would like to highlight several particularly exciting features! Below is a summary of those new features with examples.

Breaking SQL Changes

IEEE-754 semantics for division by zero. The IEEE-754 floating point standard states that division by zero returns `inf`. Previously, DuckDB would return `NULL` when dividing by zero, also for floating point division. Starting with this release, DuckDB will return `inf` instead.

```
SELECT 1 / 0 AS division_by_zero;
```

```
┌─────────────────┐
│ division_by_zero │
│     double      │
├─────────────────┤
│             inf │
└─────────────────┘
```

The `ieee_floating_point_ops` can be set to `false` to revert this behavior:

```
SET ieee_floating_point_ops = false;
SELECT 1 / 0 AS division_by_zero;
```

```
┌─────────────────┐
│ division_by_zero │
│     double      │
├─────────────────┤
│            NULL │
└─────────────────┘
```

Error when scalar subquery returns multiple values. Scalar subqueries can only return a single value per input row. Previously, DuckDB would match SQLite's behavior and select an arbitrary row to return when multiple rows were returned. In practice this behavior often led to confusion. Starting with this release, an error is returned instead, matching the behavior of Postgres. The subquery can be wrapped with `array` to collect all of the results of the subquery in a list.

```
SELECT (SELECT unnest(range(10)));
```

```
Invalid Input Error:
More than one row returned by a subquery used as an expression -
scalar subqueries can only return a single row.
```

```
SELECT array(SELECT unnest(range(10))) AS subquery_result;
```

```
┌──────────────────────────────┐
│       subquery_result        │
│           int64[]            │
├──────────────────────────────┤
│ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] │
└──────────────────────────────┘
```

The `scalar_subquery_error_on_multiple_rows` setting can be set to `false` to revert this behavior.

```
SET scalar_subquery_error_on_multiple_rows = false;
SELECT (SELECT unnest(range(10))) AS result;
```

```
┌────────┐
│ result │
│ int64  │
├────────┤
│      0 │
└────────┘
```

Community Extensions

Recently we introduced Community Extensions. Community extensions allow anyone to build extensions for DuckDB, that are then built and distributed by us. The list of community extensions has been growing since then.

In this release, we have been working towards making community extensions easier to build and produce. This release includes a new method of registering extensions using the C API in addition to a lot of extensions to the C API allowing scalar functions, aggregate functions and custom types to be defined. These changes will enable building extensions against a stable API, that are smaller in size, that will work across different DuckDB versions. In addition, these changes will enable building extensions in other programming languages in the future.

Friendly SQL

Histogram. This version introduces the `histogram` function that can be used to compute histograms over columns of a dataset. The histogram function works for columns of any type, and allows for various different binning strategies and a custom amount of bins.

```
FROM histogram(
    'https://blobs.duckdb.org/data/ontime.parquet',
    UniqueCarrier,
    bin_count := 5
);
```

```
┌──────────────┬─────────┬──────────────────────────────────────────────────────────────────────────────┐
│     bin      │  count  │                                     bar                                      │
│   varchar    │  uint64 │                                   varchar                                    │
├──────────────┼─────────┼──────────────────────────────────────────────────────────────────────────────┤
│ AA           │  677215 │ ████████████████████████████████████████████████                             │
│ DL           │  696931 │ ████████████████████████████████████████████████                             │
│ OO           │  521956 │ ████████████████████████████████████                                         │
│ UA           │  435757 │ ██████████████████████████████                                               │
│ WN           │  999114 │ ████████████████████████████████████████████████████████████████████████     │
│ (other values) │  945484 │ ████████████████████████████████████████████████████████████████             │
└──────────────┴─────────┴──────────────────────────────────────────────────────────────────────────────┘
```

SQL variables. This release introduces support for variables that can be defined in SQL. Variables can hold a single value of any type – including nested types like lists or structs. Variables can be set as literals, or from scalar subqueries.

The value stored within variables can be read using `getvariable`. When used in a query, `getvariable` is treated as a literal during query planning and optimization. This allows variables to be used in places where we normally cannot read values from within tables, for example, when specifying which CSV files to read:

```
SET VARIABLE list_of_files = (SELECT LIST(file) FROM csv_files);
SELECT * FROM read_csv(getvariable('list_of_files'), filename := true);
```

a	filename
int64	varchar
42	test.csv
84	test2.csv

Unpacked Columns

The COLUMNS expression allows users to write dynamic SQL over a set of columns without needing to explicitly list the columns in the SQL string. Instead, the columns can be selected through either a regex or computed with a lambda function.

This release expands this capability by allowing the COLUMNS expression to be *unpacked* into a function. This is especially useful when combined with nested functions like `struct_pack` or `list_value`.

```
CREATE TABLE many_measurements (
    id INTEGER, m1 INTEGER, m2 INTEGER, m3 INTEGER
);
INSERT INTO many_measurements VALUES (1, 10, 100, 20);

SELECT id, struct_pack(*COLUMNS('m\d')) AS measurements
FROM many_measurements;
```

id	measurements
int32	struct(m1 integer, m2 integer, m3 integer)
1	{'m1': 10, 'm2': 100, 'm3': 20}

`query` and `query_table` Functions

The `query` and `query_table` functions take a string literal, and convert it into a SELECT subquery or a table reference. Note that these functions can only take literal strings. As such, they are not as powerful (or dangerous) as a generic `eval`.

These functions are conceptually simple, but enable powerful and more dynamic SQL. For example, they allow passing in a table name as a prepared statement parameter:

```
CREATE TABLE my_table (i INTEGER);
INSERT INTO my_table VALUES (42);

PREPARE select_from_table AS SELECT * FROM query_table($1);
EXECUTE select_from_table('my_table');
```

i
int32
42

When combined with the COLUMNS expression, we can write very generic SQL-only macros. For example, below is a custom version of SUMMARIZE that computes the min and max of every column in a table:

```
CREATE OR REPLACE MACRO my_summarize(table_name) AS TABLE
SELECT
    unnest([*COLUMNS('alias_.*')]) AS column_name,
    unnest([*COLUMNS('min_.*')]) AS min_value,
    unnest([*COLUMNS('max_.*')]) AS max_value
FROM (
    SELECT
        any_value(alias(COLUMNS(*))) AS "alias_\0",
        min(COLUMNS(*))::VARCHAR AS "min_\0",
        max(COLUMNS(*))::VARCHAR AS "max_\0"
    FROM query_table(table_name::VARCHAR)
);

SELECT *
FROM my_summarize('https://blobs.duckdb.org/data/ontime.parquet')
LIMIT 3;
```

column_name varchar	min_value varchar	max_value varchar
year	2017	2017
quarter	1	3
month	1	9

Performance

Dynamic Filter Pushdown from Joins

This release adds a *very cool* optimization for joins: DuckDB now automatically creates filters for the larger table in the join during execution. Say we are joining two tables A and B. A has 100 rows, and B has one million rows. We are joining on a shared key `i`. If there were any filter on `i`, DuckDB would already push that filter into the scan, greatly reducing the cost to complete the query. But we are now filtering on another column from A, namely `j`:

```
CREATE TABLE A AS
    SELECT range AS i, range AS j
    FROM range(100);

CREATE TABLE B AS
    SELECT t1.range AS i
    FROM range(100) t1, range(10_000) t2;

SELECT count(*)
FROM A
JOIN B
USING (i) WHERE j > 90;
```

DuckDB will execute this join by building a hash table on the smaller table A, and then probe said hash table with the contents of B. DuckDB will now observe the values of `i` during construction of the hash table on A. It will then create a min-max range filter of those values of `i` and then *automatically* apply that filter to the values of `i` in B! That way, we early remove (in this case) 90% of data from the large table before even looking at the hash table. In this example, this leads to a roughly 10× improvement in query performance. The optimization can also be observed in the output of `EXPLAIN ANALYZE`.

Automatic CTE Materialization

Common Table Expressions (CTE) are a convenient way to break up complex queries into manageable pieces without endless nesting of subqueries. Here is a small example for a CTE:

```
WITH my_cte AS (SELECT range AS i FROM range(10))
SELECT i FROM my_cte WHERE i > 5;
```

Sometimes, the same CTE is referenced multiple times in the same query. Previously, the CTE would be "copied" wherever it appeared. This creates a potential performance problem: if computing the CTE is computationally expensive, it would be better to cache ("materialize") its results instead of computing the result multiple times in different places within the same query. However, different filter conditions might apply for different instantiations of the CTE, which could drastically reduce their computation cost. A classical no-win scenario in databases. It was already possible to explicitly mark a CTE as materialized using the `MATERIALIZED` keyword, but that required manual intervention.

This release adds a feature where DuckDB automatically decides whether a CTE result should be materialized or not using a heuristic. The heuristic currently is that if the CTE performs aggregation and is queried more than once, it should be materialized. We plan to expand that heuristic in the future.

Parallel Streaming Queries

DuckDB has two different methods for fetching results: *materialized* results and *streaming* results. Materialized results fetch all of the data that is present in a result at once, and return it. Streaming results instead allow iterating over the data in incremental steps. Streaming results are critical when working with large result sets as they do not require the entire result set to fit in memory. However, in previous releases, the final streaming phase was limited to a single thread.

Parallelism is critical for obtaining good query performance on modern hardware, and this release adds support for parallel streaming of query results. The system will use all available threads to fill up a query result buffer of a limited size (a few megabytes). When data is consumed from the result buffer, the threads will restart and start filling up the buffer again. The size of the buffer can be configured through the `streaming_buffer_size` parameter.

Below is a small benchmark using `ontime.parquet` to illustrate the performance benefits that can be obtained using the Python streaming result interface:

```
import duckdb
duckdb.sql("SELECT * FROM 'ontime.parquet' WHERE flightnum = 6805;").fetchone()
```

v1.0	v1.1
1.17 s	0.12 s

Parallel `union_by_name`

The `union_by_name` parameter allows combination of – for example – CSV files that have the same columns in them but not in the same order. This release adds support for parallelism when using `union_by_name`. This greatly improves reading performance when using the union by name feature on multiple files.

Nested ART Rework (Foreign Key Load Speed-Up)

We have greatly improved index insertion and deletion performance for foreign keys. Normally, we directly inline row identifiers into the tree structure. However, this is impossible for indexes that contain a lot of duplicates, as is the case with foreign keys. Instead, we now actually create another index entry for each key that is itself another "recursive" index tree in its own right. That way, we can achieve good insertion and deletion performance inside index entries. The performance results of this change are drastic, consider the following example where a has 100 rows and b has one million rows that all reference a:

```
CREATE TABLE a (i INTEGER, PRIMARY KEY (i));
CREATE TABLE b (i INTEGER, FOREIGN KEY (i) REFERENCES a(i));

INSERT INTO a FROM range(100);
INSERT INTO b SELECT a.range FROM range(100) a, range(10_000) b;
```

On the previous version, this would take ca. 10 seconds on a MacBook to complete. It now takes 0.2 seconds thanks to the new index structure, a ca. 50× improvement!

Window Function Improvements

Window functions see a lot of use in DuckDB, which is why we are continuously improving performance of executing Window functions over large datasets.

The `DISTINCT` and `FILTER` window function modifiers can now be executed in streaming mode. Streaming mode means that the input data for the operator does not need to be completely collected and buffered before the operator can execute. For large intermediate results, this can have a very large performance impact. For example, the following query will now use the streaming window operator:

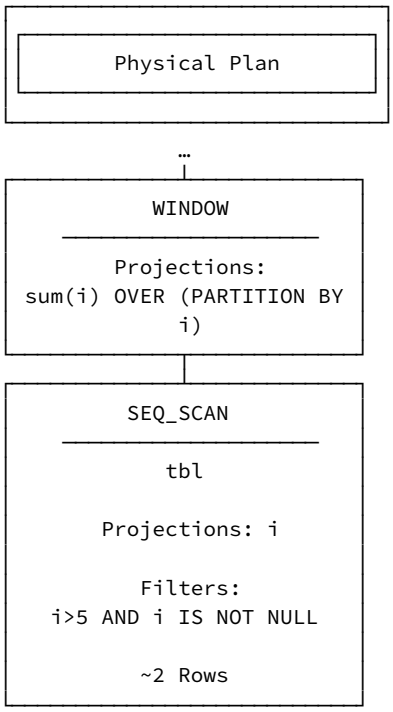
```
SELECT
    sum(DISTINCT i)
        FILTER (i % 3 = 0)
        OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM range(10) tbl(i);
```

We have also implemented streaming mode for positive `lead` offsets.

We can now push filters on columns through window functions that are partitioned by the same column. For example, consider the following scenario:

```
CREATE TABLE tbl2 AS SELECT range i FROM range(10);
SELECT i
FROM (SELECT i, SUM(i) OVER (PARTITION BY i) FROM tbl2)
WHERE i > 5;
```

Previously, the filter on `i` could not be pushed into the scan on `tbl`. But we now recognize that pushing this filter "through" the window is safe and the optimizer will do so. This can be verified through `EXPLAIN`:



The blocking (non-streaming) version of the window operator now processes input data in parallel. This greatly reduces the footprint of the window operator.

See also Richard's talk on the topic at DuckCon #5 in Seattle a few weeks ago.

Spatial Features

GeoParquet

GeoParquet is an extension format of the ubiquitous Parquet format that standardizes how to encode vector geometries and their metadata in Parquet files. This can be used to store geographic data sets in Parquet files efficiently. When the `spatial` extension is installed and loaded, reading from a GeoParquet file through DuckDB's normal Parquet reader will now automatically convert geometry columns to the `GEOMETRY` type, for example:

```
INSTALL spatial;
LOAD spatial;

FROM 'https://blobs.duckdb.org/data/geoparquet-example.parquet'
SELECT GEOMETRY g
LIMIT 10;
```

```
┌──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│                                                                                                                              g │
│                                                                                                                       geometry │
├──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┤
│ MULTIPOLYGON (((180 -16.067132663642447, 180 -16.555216566639196, 179.36414266196414 -16.801354076946883,
17…  │
│ POLYGON ((33.90371119710453 -0.95, 34.07261999999997 -1.059819999999945, 37.69868999999994
-3.09698999999994…  │
│ POLYGON ((-8.665589565454809 27.656425889592356, -8.665124477564191 27.589479071558227,
-8.684399786809053 2…  │
│ MULTIPOLYGON (((-122.84000000000003 49.000000000000114, -122.97421000000001 49.00253777777778, -124.91024
49…  │
│ MULTIPOLYGON (((-122.84000000000003 49.000000000000114, -120 49.000000000000114, -117.03121 49,
-116.04818 4…  │
└──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

R-Tree

The spatial extension accompanying this release also implements initial support for creating "R-Tree" spatial indexes. An R-Tree index stores the approximate bounding boxes of each geometry in a column into an auxiliary hierarchical tree-like data structure where every "node" contains a bounding box covering all of its child nodes. This makes it really fast to check what geometries intersect a specific region of interest as you can quickly prune out a lot of candidates by recursively moving down the tree.

Support for spatial indexes has been a long-requested feature on the spatial extension roadmap, and now that we have one, a ton of new use cases and directions for further development are opening up. However, as of now they are only used to accelerate simple queries that select from a table with a filter using one out of a hardcoded set of spatial predicate functions applied on an indexed geometry column and a constant geometry. This makes R-Tree indexes useful when you have a very large table of geometries that you repeatedly query, but you don't want to perform a full table scan when you're only interested in the rows whose geometries intersect or fit within a certain region anyway. Here is an example where we can see that the `RTREE_INDEX_SCAN` operator is used:

```
INSTALL spatial;
LOAD spatial;

-- Create a table with 10_000_000 random points
CREATE TABLE t1 AS SELECT point::GEOMETRY AS geom
FROM st_generatepoints(
        {min_x: 0, min_y: 0, max_x: 10_000, max_y: 10_000}::BOX_2D,
        10_000_000,
        1337
    );

-- Create an index on the table
CREATE INDEX my_idx ON t1 USING RTREE (geom);
```

```
-- Perform a query with a "spatial predicate" on the indexed geometry
-- column. Note how the second argument in this case,
-- the ST_MakeEnvelope call is a "constant"
SELECT count(*)
FROM t1
WHERE ST_Within(geom, ST_MakeEnvelope(450, 450, 650, 650));
```

```
3986
```

R-Tree indexes mostly share the same feature-set as DuckDB's built-in ART index. They are buffer-managed, persistent, lazily-loaded from disk and support inserts, updates and deletes to the base table. Although they can not be used to enforce constraints.

Final Thoughts

These were a few highlights – but there are many more features and improvements in this release. The full release notes can be found on GitHub.

We would like to thank again our amazing community for using DuckDB, building cool projects on DuckDB and improving DuckDB by providing us feedback. Your contributions truly mean a lot!

Changing Data with Confidence and ACID

Publication date: 2024-09-25

Authors: Hannes Mühleisen, Mark Raasveldt

TL;DR: Transactions are key features in database management systems and are also beneficial for data analysis workloads. DuckDB supports fully ACID transactions, confirmed by the TPC-H benchmark's test suite.

The great quote “Everything changes and nothing stays the same” from Heraclitus, according to Socrates, according to Plato is not very controversial: change is as old as the universe. Yet somehow, when dealing with data, we often consider change as merely an afterthought.

Static datasets are split-second snapshots of whatever the world looked like at one moment. But very quickly, the world moves on, and the dataset needs to catch up to remain useful. In the world of tables, new rows can be added, old rows may be deleted and sometimes rows have to be changed to reflect a new situation. Often, changes are interconnected. A row in a table that maps orders to customers is not very useful without the corresponding entry in the `orders` table. Most, if not all, datasets eventually get changed. As a data management system, managing change is thus not optional. However, managing changes properly is difficult.

ACID Guarantees

Early data management systems researchers invented a concept called “transactions”, the notions of which were first formalized in the 1980s. In essence, transactionality and the well-known ACID principles describe a set of guarantees that a data management system has to provide in order to be considered safe. ACID is an acronym that stands for Atomicity, Consistency, Isolation and Durability.

The ACID principles are not a theoretical exercise. Much like the rules governing airplanes or trains, they have been “written in blood” – they are hard-won lessons from decades of data management practice. It is very hard for an application to reason correctly when dealing with non-ACID systems. The end result of such problems is often corrupted data or data that no longer reflects reality accurately. For example, rows can be duplicated or missing.

DuckDB provides full ACID guarantees by default without additional configuration. In this blog post, we will describe in detail what that means together with concrete examples, and show how you can take advantage of this functionality.

Atomicity

Atomicity means that *either all changes in a set of updates happen or none of them happen*. Consider the example below, where we insert two rows in two separate tables. The inserts themselves are separate statements, but they can be made atomic by wrapping them in a transaction:

```
CREATE TABLE customer (id INTEGER, name VARCHAR);
CREATE TABLE orders (customer_id INTEGER, item VARCHAR);

BEGIN TRANSACTION;
INSERT INTO customer VALUES (42, 'DuckLabs');
INSERT INTO orders VALUES (42, 'stale bread');
COMMIT;

SELECT * FROM orders;
```

customer_id int32	item varchar
42	stale bread

By wrapping the changes in a transaction, we can be sure that *either both rows are written, or none of them are written*. The `BEGIN TRANSACTION` statement signifies all following statements belong to that transaction. The `COMMIT` signifies the end of the transaction – and will persist the changes to disk.

It is also possible to undo a set of changes by issuing a `ROLLBACK` at the end of a transaction. This will ensure that none of the changes made in the transaction are persisted.

```
BEGIN TRANSACTION;
INSERT INTO orders VALUES (42, 'iceberg lettuce');
INSERT INTO orders VALUES (42, 'dried worms');
ROLLBACK;
SELECT * FROM orders;
```

customer_id int32	item varchar
42	stale bread

As we can see, the two new rows have not been inserted permanently.

Atomicity is great to have because it allows the application to move the database from one consistent state to another consistent state without ever having to worry about intermediate states being visible to an application.

We should note that queries by default run in the so-called "auto-commit" mode, where each query will automatically be run in its own transaction. That said, even for these single-statement queries, transactions are very useful. For example, when bulk loading data into a table using an `INSERT` or `COPY` command, either *all* of the data is loaded, or *none* of the data is loaded. The system will not partially load a CSV file into a table.

We should also note that in DuckDB *schema changes are also transactional*. This means that you can create or delete tables, as well as alter the schema of a table, all within the safety of a transaction. It also means that you can undo any of these operations by issuing a `ROLLBACK`.

Consistency

Consistency means that all of the constraints that are defined in the database must always hold, both before and after a transaction. The constraints can never be violated. Examples of constraints are `PRIMARY KEY` or `FOREIGN KEY` constraints.

```
CREATE TABLE customer (id INTEGER, name VARCHAR, PRIMARY KEY (id));

INSERT INTO customer VALUES (42, 'DuckLabs');
INSERT INTO customer VALUES (42, 'Wilbur the Duck');
```

In the example above, the `customer` table requires the `id` column to be unique for all entries, otherwise multiple customers would be associated with the same orders. We can enforce this constraint by defining a so-called `PRIMARY KEY` on that column. When we insert two entries with the same id, the consistency check fails, and we get an error message:

```
Constraint Error: Duplicate key "id: 42" violates primary key
constraint. (...)
```

Having these kinds of constraints in place is a great way to make sure data *remains* consistent even after many updates have taken place.

Isolation

Isolation means that concurrent transactions are isolated from one another. A database can have many clients interacting with it *at the same time,* causing many transactions to happen all at once. An easy way of isolating these transactions is to execute them one after another. However, that would be prohibitively slow. Thousands of requests might have to wait for one particularly slow one.

To avoid this problem, transactions are typically executed *interleaved*. However, as those transactions change data, one must ensure that each transaction is logically *isolated* – it only ever sees a consistent state of the database and can – for example – never read data from a transaction that has not yet committed.

DuckDB does not have connections in the typical sense – as it is not a client/server database that allows separate applications to connect to it. However, DuckDB has full multi-client support within a single application. The user can create multiple clients that all connect to the same DuckDB instance. The transactions can be run concurrently and they are isolated using Snapshot Isolation.

The way that multiple connections are created differs per client. Below is an example where we showcase the transactionality of the system using the Python client.

```
import duckdb

con1 = duckdb.connect(":memory:mydb")
con1.sql("CREATE TABLE customer (id INTEGER, name VARCHAR)")

con1.sql("INSERT INTO customer VALUES (42, 'DuckLabs')")

con1.begin()
con1.sql("INSERT INTO customer VALUES (43, 'Wilbur the Duck')")
# no commit!

# start a new connection
con2 = duckdb.connect(":memory:mydb")
con2.sql("SELECT name FROM customer").show()

#
#   name
#  varchar
#
#  DuckLabs
#

# commit from the first connection
con1.commit()

# now the changes are visible
con2.sql("SELECT name FROM customer").show()

#
#      name
#     varchar
#
#  DuckLabs
#  Wilbur the Duck
#
```

As you can see, we have two connections to the same database, and the first connection inserts the `Wilbur the Duck` customer but *does not yet commit the change*. Meanwhile, the second connection reads from the customer table. The result does not yet show the new entry, because the two transactions are isolated from each other with regards to uncommitted changes. After the first connection commits, the second connection can read its changes.

Durability

Finally, **durability** is the behavior of a system under failure. This is important as a process might crash or power to a computer may be lost. A database system now needs to ensure that *all committed transactions* are durable, meaning their effects will be visible after restarting the database. Transactions that have not yet completed cannot leave any visible traces behind. Databases typically guarantee this property by keeping close tabs on the various caches, for example by using `fsync` to force changes to disk as transactions complete. Skipping the `fsync` is a common "optimization" that endangers durability.

Here is an example, again using Python:

```
import duckdb
import os
import signal

con = duckdb.connect("mydb.duckdb")
con.sql("CREATE TABLE customer (id INTEGER, name VARCHAR)")
con.sql("INSERT INTO customer VALUES (42, 'DuckLabs')")

# begin a transaction
con.begin()
con.sql("INSERT INTO customer VALUES (43, 'Wilbur the Duck')")
# no commit!

os.kill(os.getpid(), signal.SIGKILL)
```

After restarting, we can check the `customer` table:

```
import duckdb

con = duckdb.connect("mydb.duckdb")
con.sql("SELECT name FROM customer").show()
```

```
┌─────────────┐
│    name     │
│   varchar   │
├─────────────┤
│ DuckLabs    │
└─────────────┘
```

In this example, we first create the customer table in the database file `mydb.duckdb`. We then insert a single row with DuckLabs as a first transaction. Then, we begin but *do not commit* a second transaction that adds the `Wilbur the Duck` entry. If we then kill the process and with it the database, we can see that upon restart only the `DuckLabs` entry has survived. This is because the second transaction was not committed and hence not subject to durability. Of course, this gets more complicated when non-clean exits such as operating system crashes have to be considered. DuckDB also guarantees durability in those circumstances, some more on this below.

Why ACID in OLAP?

There are two main classes of data management systems, transactional systems (OLTP) and analytical systems (OLAP). As the name implies, transactional systems are far more concerned with guaranteeing the ACID properties than analytical ones. Systems like the venerable PostgreSQL deservedly pride themselves on doing the "right thing" with regard to providing transactional guarantees by default. Even NoSQL transactional systems such as MongoDB that swore off guaranteeing the ACID principles "for performance" early on had to eventually "roll back" to offering ACID guarantees with one or two hurdles along the way.

Analytical systems such as DuckDB – in principle – have less of an imperative to provide strong transactional guarantees. They are often not the so-called "system of record", which is the data management system that is considered the source truth. In fact, DuckDB offers various connectors to load data from systems of record, like the PostgreSQL scanner. If an OLAP database would become corrupted, it is often possible to recover from that source of truth. Of course, that first requires that users notice that something has gone wrong, which is not always simple to detect. For example, a common mistake is ingesting data from the same CSV file twice into a database because the first attempt went wrong at some point. This can lead to duplicate rows causing incorrect aggregate results. ACID prevents these kinds of problems. ACID properties enable useful functionality in OLAP systems. For example:

Concurrent Ingestion and Reporting. As change is continuous, we often have data ingestion streams adding new data to a database system. In analytical systems, it is common to have a single connection append new data to a database, while other connections read from the database in order to e.g., generate graphs and reports. If these connections are isolated, then the generated graphs and aggregates will always be executed over a complete and consistent snapshot of the database, ensuring that the generated graphs and aggregates are correct.

Rolling Back Incorrect Transformations. When analyzing data, a common pattern is loading data from data sets stored in flat files followed by performing a number of transformations on that data. For example, we might load a data set from a CSV file, followed by cleaning up

NULL values and then deleting incomplete rows. If we make an incorrect transformation, it is possible we accidentally delete too many rows.

This is not the end of the world, as we can recover by re-reading from the original CSV files. However, we can save ourselves a lot of time by wrapping the transformations in a transaction and rolling back when something goes wrong. For example:

```
CREATE TABLE people AS SELECT * FROM 'people.csv';

BEGIN TRANSACTION;
UPDATE people SET age = NULL WHERE age = -99;
-- oops, we deleted all rows!
DELETE FROM people WHERE name <> 'non-existent name';
-- we can recover our original table by rolling back the delete
ROLLBACK;
```

SQL Assertions. When a (non-syntax) error occurs in a transaction, the transaction is automatically aborted, and the changes cannot be committed. We can use this property of transactions to add assertions to our transactions. When one of these assertions is triggered, an error is raised, and the transaction cannot be committed. We can use the `error` function to define our own `assert` macro:

```
CREATE MACRO assert(condition, message) AS
    CASE WHEN NOT condition THEN error(message) END;
```

We can then use this `assert` macro to assert that the `people` table is not empty:

```
CREATE TABLE people AS SELECT * FROM 'people.csv';

BEGIN TRANSACTION;
UPDATE people SET age = NULL WHERE age = -99;
-- oops, we deleted all rows!
DELETE FROM people WHERE name <> 'non-existent name';
SELECT assert(
           (SELECT count(*) FROM people) > 0,
           'People should not be empty'
       );
COMMIT;
```

When the assertion triggers, the transaction is automatically aborted, and the changes are rolled back.

Full TPC-H Benchmark Implementation

The Transaction Processing Performance Council (TPC) is an industry association of data management systems and hardware vendors. TPC publishes database benchmark specifications and oversees auditing of benchmark results, which it then publishes on its website. There are various benchmarks aimed at different use cases. The TPC-H decision support benchmark is specifically aimed at analytical query processing on large volumes of data. Its famous 22 SQL queries and data generator specifics have been thourougly analyzed by both database vendors and academics ad nauseam.

It is less well known that the official TPC-H benchmark includes *data modification transactions* that require ACID compliance, which is not too-surprising given the name of the organization. For one-off performance shoot-outs, the updates are typically ignored and only the run-times of the 22 queries on a static dataset are reported. Such results are purely informational and cannot be audited or formally published by the TPC. But as we have argued above, change is inevitable, so let's perform the TPC-H experiments *with updates* with DuckDB.

TPC-H generates data for a fictional company selling things. The largest tables are `orders` and `lineitem`, which contains elements of each order. The benchmark can generate data of different sizes, the size is controlled by a so-called "scale factor" (SF). The specification defines two "refresh functions", that modify the database. The first refresh function will add `SF * 1500` new rows into the `orders` table, and randomly between 1 and 7 new entries for each order into the `lineitem` table. The second refresh function will delete `SF * 1500` entries from the `orders` table along with the associated `lineitem` entries. The benchmark data generator `dbgen` can generate an arbitrary amount of refresh function CSV files with new entries for `orders` and `lineitem` along with rows to be deleted.

Metrics

TPC-H's main benchmark metric is combined from both a "power" and a "throughput" test result.

The power test will run the first refresh function and time it, then run the 22 queries, then run the second refresh function, and calculate the geometric mean of all timings. With a scale factor of 100 and DuckDB 1.1.1 on a MacBook Pro with an M3 Max CPU and 64 GB of RAM, we get a *Power@Size value of 650 536*.

The throughput test will run a number of concurrent query "streams" that execute the 22 benchmark queries in shuffled order in parallel. In addition, a single refresh stream will run both refresh functions a number of times. The number of query streams and refresh sets is derived from the scale factor. For SF100, there are 5 query streams and 10 refresh sets. For our experiment, we get a *Throughput@Size of 452 571*. Results are hard to compare, but the result does not look too shabby when compared with the official result list.

ACID Tests

Section 3 of the TPC-H benchmark specification discusses the ACID properties in detail. The specification defines a set of tests to stress the ACID guarantees of a data management system. The spec duly notes that no test can prove that the ACID properties are fully supported, passing them is a "necessary but not sufficient condition" of compliance. Below, we will give an overview of what is tested.

The tests specify an "ACID Transaction", which modifies the `lineitem` and `orders` tables in such a way that an invariant holds: the `orders` table contains a total sum of all the prices of all the lineitems that belong to this order. The transaction picks a random order, and modifies the last lineitem to have a new price. It then re-calculates the order total price and updates the `orders` table with that. Finally, the transaction inserts information about which row was updated when and the price delta used in a `history` table.

To test *atomicity*, the ACID transaction is ran for a random order and then committed. It is verified that the database has been changed accordingly with the specified values. The test is repeated but this time the transaction is aborted. It is verified that the database has not been changed.

For *consistency*, a number of threads run the ACID transaction in parallel 100 times on random orders. Before and after the test, a consistency condition is checked, which essentially makes sure that the sum of all lineitem prices for an order is consistent with the sum in the order.

To test *isolation*, one thread will run the transaction, but not commit or rollback yet. Another connection will make sure the changes are not visible to it. Another set of tests will have two threads running transactions on the same order, and ensure that one of them is aborted by the system due to the conflict.

Finally, to test *durability*, a number of threads run the ACID transaction and log the results. They are allowed to complete at least 100 transactions each. Then, a failure is caused, in our case, we simply killed the process (using `SIGKILL`). Then, the database system is allowed to recover the committed changes from the write-ahead log (WAL). The log is checked to ensure that there are no log entries that are not reflected in the `history` table and there are no history entries that don't have log entries, minus very few that might have been lost in flight (i.e., persisted by the database but not yet logged by the benchmark driver). Finally, the consistency is checked again.

We're happy to report that DuckDB passed all tests.

Our scripts to run the benchmark are available on GitHub. We are planning to perform a formal audit of our results in the future. We will update this post when that happens.

Conclusion

Change in datasets is inevitable, and data management systems need to be able to safely manage change. DuckDB supports strong ACID guarantees that allow for safe and concurrent data modification. We have run extensive experiments with TPC-H's transactional validation tests and found that they pass.

Creating a SQL-Only Extension for Excel-Style Pivoting in DuckDB

Publication date: 2024-09-27

Author: Alex Monahan

TL;DR: Easily create sharable extensions using only SQL macros that can apply to any table and any columns. We demonstrate the power of this capability with the pivot_table extension that provides Excel-style pivoting.

The Power of SQL-Only Extensions

SQL is not a new language. As a result, it has historically been missing some of the modern luxuries we take for granted. With version 1.1, DuckDB has launched community extensions, bringing the incredible power of a package manager to the SQL language. A bold goal of ours is for DuckDB to become a convenient way to wrap any C++ library, much the way that Python does today, but across any language with a DuckDB client.

For extension builders, compilation and distribution are much easier. For the user community, installation is as simple as two commands:

```
INSTALL pivot_table FROM community;
LOAD pivot_table;
```

The extension can then be used in any query through SQL functions.

However, **not all of us are C++ developers**! Can we, as a SQL community, build up a set of SQL helper functions? What would it take to build these extensions with *just SQL?*

Reusability

Traditionally, SQL is highly customized to the schema of the database on which it was written. Can we make it reusable? Some techniques for reusability were discussed in the SQL Gymnasics post, but now we can go even further. With version 1.1, DuckDB's world-class friendly SQL dialect makes it possible to create macros that can be applied:

- To any tables
- On any columns
- Using any functions

The new ability to work **on any tables** is thanks to the `query` and `query_table` functions! The `query` function is a safe way to execute `SELECT` statements defined by SQL strings, while `query_table` is a way to make a `FROM` clause pull from multiple tables at once. They are very powerful when used in combination with other friendly SQL features like the `COLUMNS` expression and `LIST` lambda functions.

Community Extensions as a Central Repository

Traditionally, there has been no central repository for SQL functions across databases, let alone across companies! DuckDB's community extensions can be that knowledge base. DuckDB extensions can be used across all languages with a DuckDB client, including Python, NodeJS, Java, Rust, Go, and even WebAssembly (Wasm)!

If you are a DuckDB fan and a SQL user, you can share your expertise back to the community with an extension. This post will show you how! No C++ knowledge is needed – just a little bit of copy/paste and GitHub Actions handles all the compilation. If I can do it, you can do it!

Powerful SQL

All that said, just how valuable can a SQL MACRO be? Can we do more than make small snippets? I'll make the case that you can do quite complex and powerful operations in DuckDB SQL using the `pivot_table` extension as an example. The `pivot_table` function allows for Excel-style pivots, including `subtotals`, `grand_totals`, and more. It is also very similar to the Pandas `pivot_table` function, but with all the scalability and speed benefits of DuckDB. It contains over **250 tests**, so it is intended to be useful beyond just an example!

To achieve this level of flexibility, the `pivot_table` extension uses many friendly and advanced SQL features:

- The `query` function to execute a SQL string
- The `query_table` function to query a list of tables
- The `COLUMNS` expression to select a dynamic list of columns
- List lambda functions to build up the SQL statement passed into `query`
  - `list_transform` for string manipulation like quoting
  - `list_reduce` to concatenate strings together
  - `list_aggregate` to sum multiple columns and identify subtotal and grand total rows
- Bracket notation for string slicing
- `UNION ALL BY NAME` to stack data by column name for subtotals and grand totals
- `SELECT * REPLACE` to dynamically clean up subtotal columns
- `SELECT * EXCLUDE` to remove internally generated columns from the final result
- `GROUPING SETS` and `ROLLUP` to generate subtotals and grand totals
- `UNNEST` to convert lists into separate rows for `values_axis := 'rows'`
- `MACRO`s to modularize the code
- `ORDER BY ALL` to order the result dynamically
- `ENUM`s to determine what columns to pivot horizontally
- And of course the `PIVOT` function for horizontal pivoting!

DuckDB's innovative syntax makes this extension possible!

So, we now have all 3 ingredients we will need: a central package manager, reusable macros, and enough syntactic flexibility to do valuable work.

Create Your Own SQL Extension

Let's walk through the steps to creating your own SQL-only extension.

Writing the Extension

Extension Setup The first step is to create your own GitHub repo from the DuckDB Extension Template for SQL by clicking *Use this template.*

Then clone your new repository onto your local machine using the terminal:

```
git clone --recurse-submodules \
    https://github.com/<your_github_username>/<your_extension_repo>.git
```

Note that `--recurse-submodules` will ensure DuckDB is pulled which is required to build the extension.

Next, replace the name of the example extension with the name of your extension in all the right places by running the Python script below.

> **Note.** If you don't have Python installed, head to python.org and follow those instructions. This script doesn't require any libraries, so Python is all you need! (No need to set up any environments.)

```
python3 ./scripts/bootstrap-template.py <extension_name_you_want>
```

Initial Extension Test At this point, you can follow the directions in the README to build and test locally if you would like. However, even easier, you can simply commit your changes to git and push them to GitHub, and GitHub Actions can do the compilation for you! GitHub Actions will also run tests on your extension to validate it is working properly.

> **Note.** The instructions are not written for a Windows audience, so we recommend GitHub Actions in that case!

```
git add -A
git commit -m "Initial commit of my SQL extension!"
git push
```

Write Your SQL Macros It is likely a bit faster to iterate if you test your macros directly in DuckDB. After you have written your SQL, we will move it into the extension. The example we will use demonstrates how to pull a dynamic set of columns from a dynamic table name (or a view name!).

```
CREATE OR REPLACE MACRO select_distinct_columns_from_table(table_name, columns_list) AS TABLE (
    SELECT DISTINCT
        COLUMNS(lambda column_name: list_contains(columns_list, column_name))
    FROM query_table(table_name)
    ORDER BY ALL
);

FROM select_distinct_columns_from_table('duckdb_types', ['type_category']);
```

type_category
BOOLEAN
COMPOSITE
DATETIME
NUMERIC
STRING
NULL

Add SQL Macros Technically, this is the C++ part, but we are going to do some copy/paste and use GitHub Actions for compiling so it won't feel that way!

DuckDB supports both scalar and table macros, and they have slightly different syntax. The extension template has an example for each (and code comments too!) inside the file named `<your_extension_name>.cpp`. Let's add a table macro here since it is the more complex one. We will copy the example and modify it!

```
static const DefaultTableMacro <your_extension_name>_table_macros[] = {
    {DEFAULT_SCHEMA, "times_two_table", {"x", nullptr}, {{"two", "2"}, {nullptr, nullptr}}, R"(SELECT x *
    two AS output_column;)"},
    {
        DEFAULT_SCHEMA, // Leave the schema as the default
        "select_distinct_columns_from_table", // Function name
        {"table_name", "columns_list", nullptr}, // Parameters
        {{nullptr, nullptr}}, // Optional parameter names and values (we choose not to have any here)
        // The SQL text inside of your SQL Macro, wrapped in R"( )", which is a raw string in C++
        R"(
            SELECT DISTINCT
                COLUMNS(lambda column_name: list_contains(columns_list, column_name))
            FROM query_table(table_name)
            ORDER BY ALL
        )"
    },
    {nullptr, nullptr, {nullptr}, {{nullptr, nullptr}}, nullptr}
    };
```

That's it! All we had to provide were the name of the function, the names of the parameters, and the text of our SQL macro.

Testing the Extension

We also recommend adding some tests for your extension to the `<your_extension_name>.test` file. This uses sqllogictest to test with just SQL! Let's add the example from above.

> **Note.** In sqllogictest, `query I` indicates that there will be 1 column in the result. We then add `----` and the resultset in tab separated format with no column names.

```
query I
FROM select_distinct_columns_from_table('duckdb_types', ['type_category']);
----
BOOLEAN
COMPOSITE
DATETIME
NUMERIC
STRING
NULL
```

Now, just add, commit, and push your changes to GitHub like before, and GitHub Actions will compile your extension and test it!

If you would like to do further ad-hoc testing of your extension, you can download the extension from your GitHub Actions run's artifacts and then install it locally using these steps.

Uploading to the Community Extensions Repository

Once you are happy with your extension, it's time to share it with the DuckDB community! Follow the steps in the Community Extensions post. A summary of those steps is:

1. Send a PR with a metadata file `description.yml` that contains the description of the extension. For example, the h3 community extension uses the following YAML configuration:

   ```
   extension:
     name: h3
     description: Hierarchical hexagonal indexing for geospatial data
     version: 1.0.0
     language: C++
     build: cmake
     license: Apache-2.0
     maintainers:
       - isaacbrodsky

   repo:
     github: isaacbrodsky/h3-duckdb
     ref: 3c8a5358e42ab8d11e0253c70f7cc7d37781b2ef
   ```

2. Wait for approval from the maintainers.

And there you have it! You have created a shareable DuckDB community extension. Now let's have a look at the `pivot_table` extension as an example of just how powerful a SQL-only extension can be.

Capabilities of the `pivot_table` Extension

The `pivot_table` extension supports advanced pivoting functionality that was previously only available in spreadsheets, dataframe libraries, or custom host language functions. It uses the Excel pivoting API: `values`, `rows`, `columns`, and `filters` – handling 0 or more of each of those parameters. However, not only that, but it supports `subtotals` and `grand_totals`. If multiple `values` are passed in, the `values_axis` parameter allows the user to choose if each value should get its own column or its own row.

Why is this a good example of how DuckDB moves beyond traditional SQL? The Excel pivoting API requires dramatically different SQL syntax depending on which parameters are in use. If no `columns` are pivoted outward, a `GROUP BY` is all that is needed. However, once `columns` are involved, a `PIVOT` is required.

This function can operate on one or more `table_names` that are passed in as a parameter. Any set of tables (or views!) will first be vertically stacked and then pivoted.

Example Using `pivot_table`

Check out a live example using the extension in the DuckDB Wasm shell here!

product_line	product	year	quarter	revenue	cost
Waterfowl watercraft	Duck boats	2022	Q1	100	100
Waterfowl watercraft	Duck boats	2022	Q2	200	100
Waterfowl watercraft	Duck boats	2022	Q3	300	100
Waterfowl watercraft	Duck boats	2022	Q4	400	100
Waterfowl watercraft	Duck boats	2023	Q1	500	100
Waterfowl watercraft	Duck boats	2023	Q2	600	100
Waterfowl watercraft	Duck boats	2023	Q3	700	100
Waterfowl watercraft	Duck boats	2023	Q4	800	100
Duck Duds	Duck suits	2022	Q1	10	10
Duck Duds	Duck suits	2022	Q2	20	10
Duck Duds	Duck suits	2022	Q3	30	10
Duck Duds	Duck suits	2022	Q4	40	10
Duck Duds	Duck suits	2023	Q1	50	10
Duck Duds	Duck suits	2023	Q2	60	10
Duck Duds	Duck suits	2023	Q3	70	10
Duck Duds	Duck suits	2023	Q4	80	10
Duck Duds	Duck neckties	2022	Q1	1	1
Duck Duds	Duck neckties	2022	Q2	2	1
Duck Duds	Duck neckties	2022	Q3	3	1
Duck Duds	Duck neckties	2022	Q4	4	1
Duck Duds	Duck neckties	2023	Q1	5	1
Duck Duds	Duck neckties	2023	Q2	6	1
Duck Duds	Duck neckties	2023	Q3	7	1
Duck Duds	Duck neckties	2023	Q4	8	1

Next, we install the extension from the community repository:

```
INSTALL pivot_table FROM community;
LOAD pivot_table;
```

Now we can build pivot tables like the one below. There is a little bit of boilerplate required, and the details of how this works will be explained shortly.

```
DROP TYPE IF EXISTS columns_parameter_enum;

CREATE TYPE columns_parameter_enum AS ENUM (
    FROM build_my_enum(['business_metrics'],     -- table_names
                       ['year', 'quarter'],      -- columns
                       [])                       -- filters
);

FROM pivot_table(['business_metrics'],           -- table_names
                 ['sum(revenue)', 'sum(cost)'], -- values
                 ['product_line', 'product'],    -- rows
                 ['year', 'quarter'],            -- columns
                 [],                             -- filters
                 subtotals := 1,
                 grand_totals := 1,
                 values_axis := 'rows'
                 );
```

product_line	product	value_names	2022_Q1	2022_Q2	2022_Q3	2022_Q4	2023_Q1	2023_Q2	2023_Q3	2023_Q4
Duck Duds	Duck neckties	sum(cost)	1	1	1	1	1	1	1	1
Duck Duds	Duck neckties	sum(revenue)	2	3	4	5	6	7	8	
Duck Duds	Duck suits	sum(cost)	10	10	10	10	10	10	10	10
Duck Duds	Duck suits	sum(revenue)	20	30	40	50	60	70	80	
Duck Duds	Subtotal	sum(cost)	11	11	11	11	11	11	11	11
Duck Duds	Subtotal	sum(revenue)	22	33	44	55	66	77	88	
Waterfowl watercraft	Duck boats	sum(cost)	100	100	100	100	100	100	100	100
Waterfowl watercraft	Duck boats	sum(revenue)	200	300	400	500	600	700	800	
Waterfowl watercraft	Subtotal	sum(cost)	100	100	100	100	100	100	100	100
Waterfowl watercraft	Subtotal	sum(revenue)	200	300	400	500	600	700	800	
Grand Total	Grand Total	sum(cost)	111	111	111	111	111	111	111	111
Grand Total	Grand Total	sum(revenue)	222	333	444	555	666	777	888	

How the `pivot_table` Extension Works

The `pivot_table` extension is a collection of multiple scalar and table SQL macros. This allows the logic to be modularized. You can see below that the functions are used as building blocks to create more complex functions. This is typically difficult to do in SQL, but it is easy in DuckDB!

The functions and a brief description of each follows.

Building Block Scalar Functions

- `nq`: "No quotes" – Escape semicolons in a string to prevent SQL injection
- `sq`: "Single quotes" – Wrap a string in single quotes and escape embedded single quotes
- `dq`: "Double quotes" – Wrap in double quotes and escape embedded double quotes
- `nq_list`: Escape semicolons for each string in a list. Uses `nq`.
- `sq_list`: Wrap each string in a list in single quotes. Uses `sq`.
- `dq_list`: Wrap each string in a list in double quotes. Uses `dq`.
- `nq_concat`: Concatenate a list of strings together with semicolon escaping. Uses `nq_list`.
- `sq_concat`: Concatenate a list of strings together, wrapping each in single quotes. Uses `sq_list`.

- `dq_concat`: Concatenate a list of strings together, wrapping each in double quotes. Uses `dq_list`.

Functions Creating During Refactoring for Modularity

- `totals_list`: Build up a list as a part of enabling `subtotals` and `grand_totals`.
- `replace_zzz`: Rename `subtotal` and `grand_total` indicators after sorting so they are more friendly.

Core Pivoting Logic Functions

- `build_my_enum`: Determine which new columns to create when pivoting horizontally. Returns a table. See below for details.
- `pivot_table`: Based on inputs, decide whether to call `no_columns`, `columns_values_axis_columns` or `columns_values_axis_rows`. Execute `query` on the SQL string that is generated. Returns a table. See below for details.
  - `no_columns`: Build up the SQL string for `query` to execute when no `columns` are pivoted out.
  - `columns_values_axis_columns`: Build up the SQL string for `query` to execute when pivoting horizontally with each entry in `values` receiving a separate column.
  - `columns_values_axis_rows`: Build up the SQL string for `query` to execute when pivoting horizontally with each entry in `values` receiving a separate row.
- `pivot_table_show_sql`: Return the SQL string that would have been executed by `query` for debugging purposes.

The `build_my_enum` Function

The first step in using the `pivot_table` extension's capabilities is to define an `ENUM` (a user-defined type) containing all of the new column names to create when pivoting horizontally called `columns_parameter_enum`. DuckDB's automatic `PIVOT` syntax can automatically define this, but in our case, we need 2 explicit steps. The reason for this is that automatic pivoting runs 2 statements behind the scenes, but a `MACRO` must only be a single statement. If the `columns` parameter is not in use, this step is essentially a no-op, so it can be omitted or included for consistency (recommended).

The `query` and `query_table` functions only support `SELECT` statements (for security reasons), so the dynamic portion of the `ENUM` creation occurs in the function `build_my_enum`. If this type of usage becomes common, features could be added to DuckDB to enable a `CREATE OR REPLACE` syntax for `ENUM` types, or possibly even temporary enums. That would reduce this pattern from 3 statements down to 2. Please let us know!

The `build_my_enum` function uses a combination of `query_table` to pull from multiple input tables, and the `query` function so that double quotes (and correct character escaping) can be completed prior to passing in the list of table names. It uses a similar pattern to the core `pivot_table` function: build up a SQL query as a string, then call it with `query`. The SQL string is constructed using list lambda functions and the building block functions for quoting.

The `pivot_table` Function

At its core, the `pivot_table` function determines the SQL required to generate the desired pivot based on which parameters are in use.

Since this SQL statement is a string at the end of the day, we can use a hierarchy of scalar SQL macros rather than a single large macro. This is a common traditional issue with SQL – it tends to not be very modular or reusable, but we are able to compartmentalize our logic with DuckDB's syntax.

> **Note.** If a non-optional parameter is not in use, an empty list (`[]`) should be passed in.

- `table_names`: A list of table or view names to aggregate or pivot. Multiple tables are combined with `UNION ALL BY NAME` prior to any other processing.
- `values`: A list of aggregation metrics in the format `['agg_fn_1(col_1)', 'agg_fn_2(col_2)', ...]`.
- `rows`: A list of column names to `SELECT` and `GROUP BY`.

- `columns`: A list of column names to `PIVOT` horizontally into a separate column per value in the original column. If multiple column names are passed in, only unique combinations of data that appear in the dataset are pivoted.
  - Ex: If passing in a `columns` parameter like `['continent', 'country']`, only valid `continent` / `country` pairs will be included.
  - (no `Europe_Canada` column would be generated).
- `filters`: A list of `WHERE` clause expressions to be applied to the raw dataset prior to aggregating in the format `['col_1 = 123', 'col_2 LIKE ''woot%''', ...]`.
  - The `filters` are combined with `AND`.
- `values_axis` (Optional): If multiple `values` are passed in, determine whether to create a separate row or column for each value. Either `rows` or `columns`, defaulting to `columns`.
- `subtotals` (Optional): If enabled, calculate the aggregate metric at multiple levels of detail based on the `rows` parameter. Either 0 or 1, defaulting to 0.
- `grand_totals` (Optional): If enabled, calculate the aggregate metric across all rows in the raw data in addition to at the granularity defined by `rows`. Either 0 or 1, defaulting to 0.

No Horizontal Pivoting (No `columns` in Use) If not using the `columns` parameter, no columns need to be pivoted horizontally. As a result, a `GROUP BY` statement is used. If `subtotals` are in use, the `ROLLUP` expression is used to calculate the `values` at the different levels of granularity. If `grand_totals` are in use, but not `subtotals`, the `GROUPING SETS` expression is used instead of `ROLLUP` to evaluate across all rows.

In this example, we build a summary of the `revenue` and `cost` of each `product_line` and `product`.

```
FROM pivot_table(['business_metrics'],
                 ['sum(revenue)', 'sum(cost)'],
                 ['product_line', 'product'],
                 [],
                 [],
                 subtotals := 1,
                 grand_totals := 1,
                 values_axis := 'columns'
                 );
```

product_line	product	sum(revenue)	sum("cost")
Duck Duds	Duck neckties	36	8
Duck Duds	Duck suits	360	80
Duck Duds	Subtotal	396	88
Waterfowl watercraft	Duck boats	3600	800
Waterfowl watercraft	Subtotal	3600	800
Grand Total	Grand Total	3996	888

Pivot Horizontally, One Column per Metric in `values` Build up a `PIVOT` statement that will pivot out all valid combinations of raw data values within the `columns` parameter. If `subtotals` or `grand_totals` are in use, make multiple copies of the input data, but replace appropriate column names in the `rows` parameter with a string constant. Pass all expressions in `values` to the `PIVOT` statement's `USING` clause so they each receive their own column.

We enhance our previous example to pivot out a separate column for each `year` / `value` combination:

```
DROP TYPE IF EXISTS columns_parameter_enum;

CREATE TYPE columns_parameter_enum AS ENUM (
    FROM build_my_enum(['business_metrics'],
```

```
                      ['year'],
                      [])
);

FROM pivot_table(['business_metrics'],
                 ['sum(revenue)', 'sum(cost)'],
                 ['product_line', 'product'],
                 ['year'],
                 [],
                 subtotals := 1,
                 grand_totals := 1,
                 values_axis := 'columns'
                 );
```

product_line	product	2022_sum(revenue)	2022_sum("cost")	2023_sum(revenue)	2023_sum("cost")
Duck Duds	Duck neckties	10	4	26	4
Duck Duds	Duck suits	100	40	260	40
Duck Duds	Subtotal	110	44	286	44
Waterfowl watercraft	Duck boats	1000	400	2600	400
Waterfowl watercraft	Subtotal	1000	400	2600	400
Grand Total	Grand Total	1110	444	2886	444

Pivot Horizontally, One Row per Metric in `values` Build up a separate `PIVOT` statement for each metric in `values` and combine them with `UNION ALL BY NAME`. If `subtotals` or `grand_totals` are in use, make multiple copies of the input data, but replace appropriate column names in the `rows` parameter with a string constant.

To simplify the appearance slightly, we adjust one parameter in our previous query and set `values_axis := 'rows'`:

```
DROP TYPE IF EXISTS columns_parameter_enum;

CREATE TYPE columns_parameter_enum AS ENUM (
    FROM build_my_enum(['business_metrics'],
                       ['year'],
                       [])
);

FROM pivot_table(['business_metrics'],
                 ['sum(revenue)', 'sum(cost)'],
                 ['product_line', 'product'],
                 ['year'],
                 [],
                 subtotals := 1,
                 grand_totals := 1,
                 values_axis := 'rows'
                 );
```

product_line	product	value_names	2022	2023
Duck Duds	Duck neckties	sum(cost)	4	4
Duck Duds	Duck neckties	sum(revenue)	10	26
Duck Duds	Duck suits	sum(cost)	40	40
Duck Duds	Duck suits	sum(revenue)	100	260
Duck Duds	Subtotal	sum(cost)	44	44
Duck Duds	Subtotal	sum(revenue)	110	286

product_line	product	value_names	2022	2023
Waterfowl watercraft	Duck boats	sum(cost)	400	400
Waterfowl watercraft	Duck boats	sum(revenue)	1000	2600
Waterfowl watercraft	Subtotal	sum(cost)	400	400
Waterfowl watercraft	Subtotal	sum(revenue)	1000	2600
Grand Total	Grand Total	sum(cost)	444	444
Grand Total	Grand Total	sum(revenue)	1110	2886

Conclusion

With DuckDB 1.1, sharing your SQL knowledge with the community has never been easier! DuckDB's community extension repository is truly a package manager for the SQL language. Macros in DuckDB are now highly reusable (thanks to `query` and `query_table`), and DuckDB's SQL syntax provides plenty of power to accomplish complex tasks.

Please let us know if the `pivot_table` extension is helpful to you – we are open to both contributions and feature requests! Together we can write the ultimate pivoting capability just once and use it everywhere.

In the future, we have plans to further simplify the creation of SQL extensions. Of course, we would love your feedback! Join us on Discord in the `community-extensions` channel.

Happy analyzing!

DuckDB in Python in the Browser with Pyodide, PyScript, and JupyterLite

Publication date: 2024-10-02

Author: Alex Monahan

TL;DR: Run DuckDB in an in-browser Python environment to enable simple querying on remote files, interactive documentation, and easy to use training materials.

{:/nomarkdown}

Time to "Hello World"

The first time that you are using a new library, the most important thing is how quickly you can get to "Hello World".

> **Note.** Want to see "Hello World"? Jump to the **fully interactive** examples!

Likewise, if someone is visiting any documentation you have written, you want them to quickly and easily get your tool up and running. When you are giving a demo, you want to avoid "demo hell" and have it work the first try!

If you want to try "expert mode", try leading an entire conference room of people through those setup steps! The classroom or conference workshop environment makes it far more critical that installation be bulletproof.

Python is one of our favorite ways to use DuckDB, but Python is notoriously difficult to set up – doubly so for a novice programmer. What the heck is a virtual environment? Are you on Windows, Linux, or Mac? Pip or Conda? The new kid on the block uv?

Experienced Pythonistas are not immune either! Many, like me, have been forced to celebrate the time honored and xkcd-chronicled tradition of just wiping everything related to Python and starting from scratch.

How can we make it as easy and as fast as possible to test out DuckDB in Python?

Difficulties of Server-Side Python

One response to this challenge is to host a Python environment on a server for each of your users. This has a number of issues.

Hosting Python on a server yourself is not free. If you have many users, it can be far from free.

If you want to use a free solution like Google Colab, each visitor will need a Google account, and you'll need to be comfortable with Google accessing your data. Plus, it is hard to embed within an existing web page for a seamless experience.

Enter Pyodide

Pyodide is a way to run Python directly in your browser with no installation and no setup, thanks to the power of WebAssembly. That makes it the easiest and fastest way to get a Python environment up and running – just load a web page! All computation happens locally, so it can be served like any static website with tools like GitHub Pages. No server-side Python required!

Another benefit is that Pyodide is nicely sandboxed in the browser environment. Each user gets their own workspace, and since it is all local, it is nice and secure.

Part of what sets Pyodide apart from other in-browser-Python approaches is that it can even run libraries that are written in C, C++, or even Fortran, including much of the Python data science stack. This means that now you can use DuckDB in Pyodide as well! You can even combine it with NumPy, SciPy, and Pandas (in addition to many pure-Python libraries). PyArrow and Ibis have experimental support also.

Use Cases for Pyodide DuckDB

Want to quickly analyze some remote data using either Python or DuckDB?

Pyodide is the fastest way to get your questions answered using Python.

Want to quickly analyze some local data?

Pyodide can also query local files!

Want to make your documentation interactive?

Let your users test out your DuckDB-powered library with ease. We will see an example below that demonstrates the `magic-duckdb` Jupyter plugin to enable SQL cells.

Leading a training session with DuckDB and Python?

Skip the hassles of local installation. There is no need to work 1:1 with the 15% of folks in the audience with some quirky setup! Everyone will get this to work on the first try, in seconds, so you can get to the content you want to teach. Plus, it is free, with no signup required of any kind!

Pyodide Examples

We will cover multiple ways to embed Pyodide-powered-Python directly into your site, so your users can try out your new DuckDB-backed tool with a single click!

- **PyScript Editor:** An editor with nice syntax highlighting
- **JupyterLite Notebook:** A classic notebook environment
- **JupyterLite Lab IDE:** A full development environment

PyScript Editor

This HTML snippet will embed a runnable PyScript editor into any page!

```
<script type="module" src="https://pyscript.net/releases/2024.8.2/core.js"></script>
<script type="py-editor" config='{"packages":["duckdb"]}'>
    import duckdb
    print(duckdb.sql("SELECT '42 in an editor' AS s").fetchall())
</script>
```

Just click the play button and you can execute a DuckDB query directly in the browser. You can edit the code, add new lines, etc. Try it out!

{:/nomarkdown}

JupyterLite Notebook

Here is an example of using an `iframe` that points to a JupyterLite environment that was deployed to GitHub Pages!

This is a fully interactive Python notebook environment, with DuckDB running inside. Feel free to give it a run!

{:/nomarkdown}

Configuring a full JupyterLite environment is only a few steps! The JupyterLite folks have built a demo page that serves as a template and have some great documentation. The main steps are to:

1. Use the JupyterLite Demo Template to create your own repo
2. Enable GitHub Pages for that repo

3. Add and commit a .ipynb file in the `content` folder
4. Visit `https://<your_github_username>.github.io/<YOUR_REPOSITORY_NAME>/notebooks/index.html?path=<your_notebook_name>.ipynb{:.language-sql .highlight}`

Note that it can take a couple of minutes for GitHub Pages to deploy. You can monitor the progress on GitHub's Actions tab.

JupyterLite Lab IDE

After following the steps in the JupterLite Notebook setup, if you change your URL from `/notebooks/` to `/lab/`, you can have a full IDE experience instead! This form factor is a bit harder to embed in another page, but great for interactive use.

This example uses the `magic-duckdb` Jupyter extension that allows us to create SQL cells using `%%dql`.

Follow this link to see the Lab IDE interface, or experiment with the Notebook-style version below.

{:/nomarkdown}

Architecture of DuckDB in Pyodide

So how does DuckDB work in Pyodide exactly? The DuckDB Python client is compiled to WebAssembly (Wasm) in its entirety. This is different than the existing DuckDB Wasm approach, since that is compiling the C++ side of the library only and wrapping it with a JavaScript API. Both approaches use the Emscripten toolchain to do the Wasm compilation. It is DuckDB's design decision to avoid dependencies and the prior investments in DuckDB-Wasm that made this feasible to build in such a short period of time!

The Pyodide team has added DuckDB to their hosted repository of libraries, and even set up DuckDB to run as a part of their CI/CD workflow. That is what enables JupyterLite to simply run `%pip install duckdb`, and PyScript to specify DuckDB as a package in the `py-editor config` parameter or in the `<py-config>` tag. Pyodide then downloads the Wasm-compiled version of the DuckDB library from Pyodide's repository. We want to send a big thank you to the Pyodide team, including Hood Chatham and Gyeongjae Choi, as well as the Voltron Data team including Phillip Cloud for leading the effort to get this to work.

Limitations

Running in the browser is a more restrictive environment (for security purposes), so there are some limitations when using DuckDB in Pyodide. There is no free lunch!

- Single-threaded
  - Pyodide currently limits execution to a single thread
- A few extra steps to query remote files
  - Remote files can't be accessed by DuckDB directly
  - Instead, pull the files locally with Pyodide first
  - DuckDB-Wasm has custom enhancements to make this possible, but these are not present in DuckDB's Python client
- No runtime-loaded extensions
  - Several extensions are automatically included: `parquet`, `json`, `icu`, `tpcds`, and `tpch`.
- Release cadence aligned with Pyodide
  - At the time of writing, duckdb-pyodide is at 1.0.0 rather than 1.1.1

Conclusion

Pyodide is now the fastest way to use Python and DuckDB together! It is also an approach that scales to an arbitrary number of users because Pyodide's computations happen entirely locally.

We have seen how to embed Pyodide in a static site in multiple ways, as well as how to read remote files.

If you are excited about DuckDB in Pyodide, feel free to join us on Discord. We have a `#show-and-tell` channel where you can share what you build with the community. You are also welcome to explore the duckdb-pyodide repo and report any issues you find. We would also really love some help with enabling runtime-loaded extensions – please reach out if you can help!

Happy quacking about!

DuckDB User Survey Analysis

Publication date: 2024-10-04

Author: Gábor Szárnyas

TL;DR: We share the findings from a survey of 500+ DuckDB users.

Earlier this year, we conducted a survey in the DuckDB community. We were mostly curious about the following topics:

1. How do people use DuckDB?
2. Where do people use DuckDB?
3. What do they like about DuckDB?
4. What improvements would they like to see in future releases?

The survey was open for about three weeks. More than 500 people submitted their answers, and we raffled 20 t-shirts and hoodies among the participants.

Summary

We summarize the key findings of the survey below:

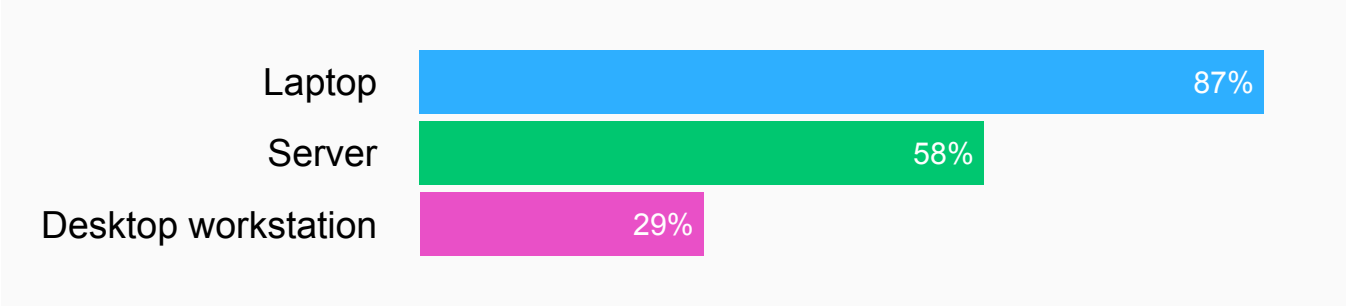
- Users run DuckDB most often on laptops but servers are also very popular.
- The most popular clients are the Python API and the standalone CLI client.
- Most users don't have huge data sets but they appreciate high performance very much.
- Users would like to see performance optimizations related to time series and partitioned data.
- DuckDB is popular among data engineers, analysts and scientists, and also among software engineers.

Let's dive into the details!

Using DuckDB

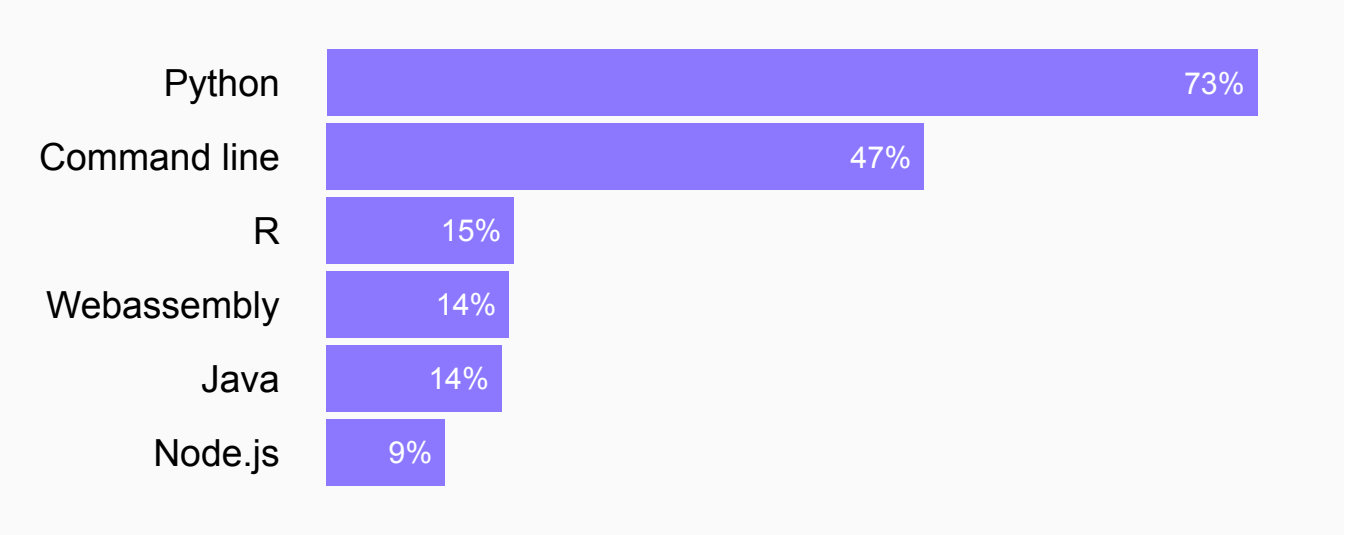
Environments

We asked users about the environment where DuckDB is deployed and found that most of them, 87%, run DuckDB on their laptops. This is in line with the vision that originally drove the creation of DuckDB: creating a system that harnesses the power of hardware available in modern end-user devices. 29% run DuckDB on desktop workstations, and 58% run it on servers (see the breakdown later in the "Server Types" section).



Clients

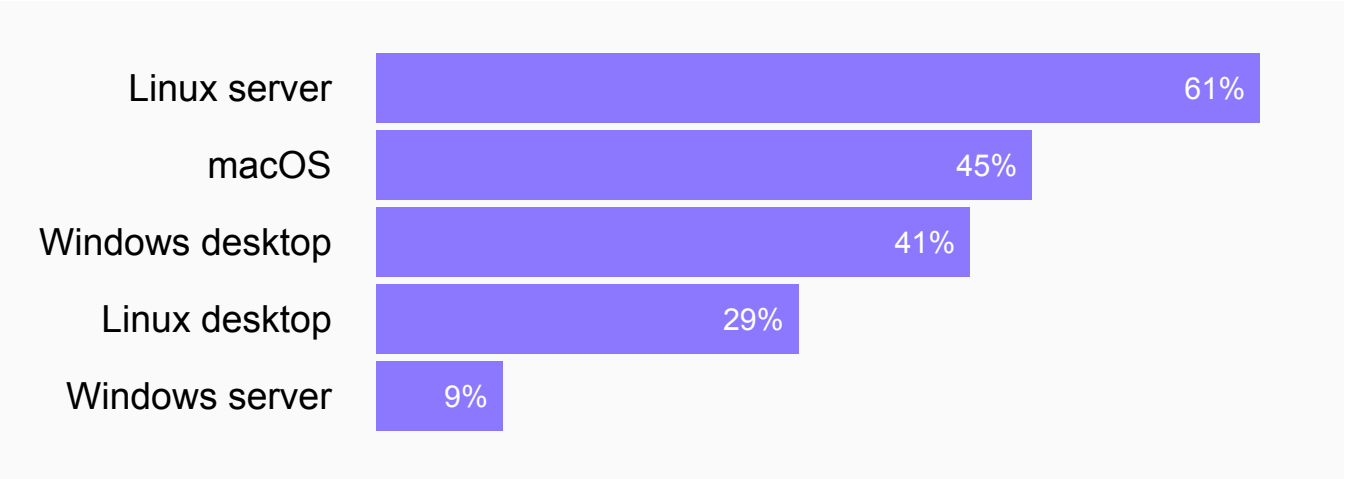
Unsurprisingly, DuckDB is most often used from Python (73%), followed by the standalone command-line application (47%). The third spot is hotly contested with R, WebAssembly (!) and Java all achieving around 14%, followed by Node.js (Javascript) at 9%.



The next few places, with 6-7% each, are occupied by ODBC, Rust, and Go. Finally, Arrow (ADBC) rounds off the top 10 with 5%.

Operating Systems

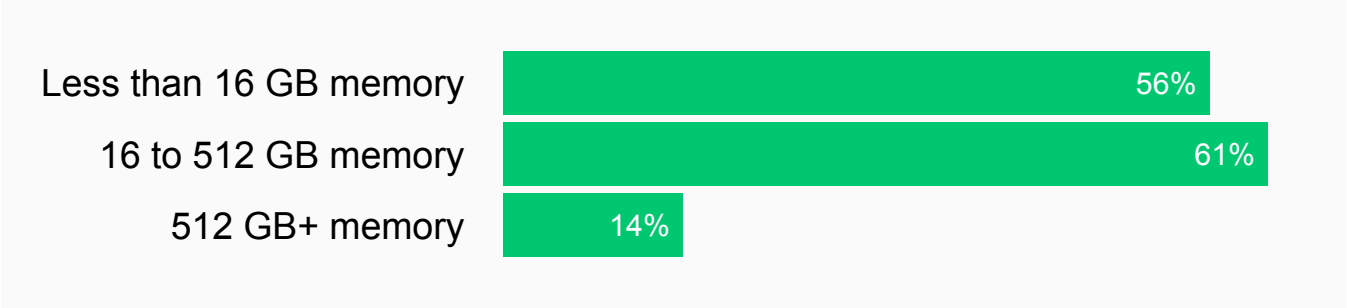
We found that most users, 61%, run DuckDB on Linux servers. These deployments include cloud instances, on-premises installations, and continuous integration (CI) runners. Windows desktop and macOS have a similar share of users, 41–45%. A further 9% run DuckDB on Windows servers.



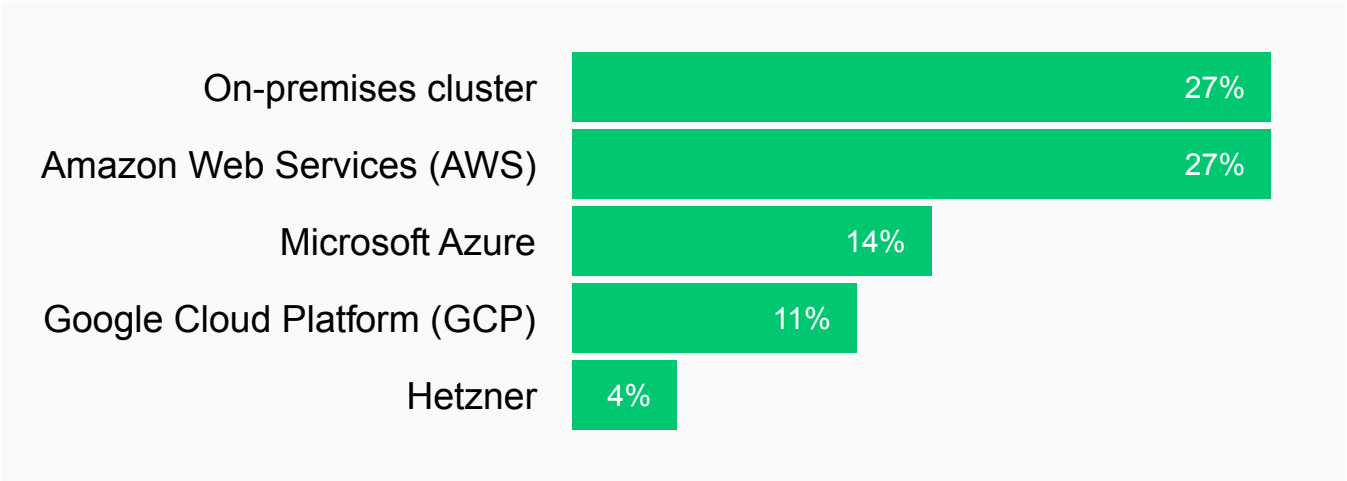
We found the number of Linux desktop users quite striking. While the overall market share of Linux desktop is around 4.5%, *29% of respondents indicated that they run DuckDB on Linux desktop!* We suspect that this is thanks to DuckDB's popularity among data engineers, who often use Linux desktop due to its customizability and similarity to the Linux server-based deployment environments.

Server Types

As we discussed in the "Environments" section, DuckDB is often run on servers. But how big are these servers, and where are they operated? Both small servers (less than 16 GB of memory) and medium-sized servers (16-512 GB of memory) are popular, with 56% and 61% of users reporting that they run DuckDB on these. About 14% of respondents run DuckDB on servers with more than 0.5 TB of memory.



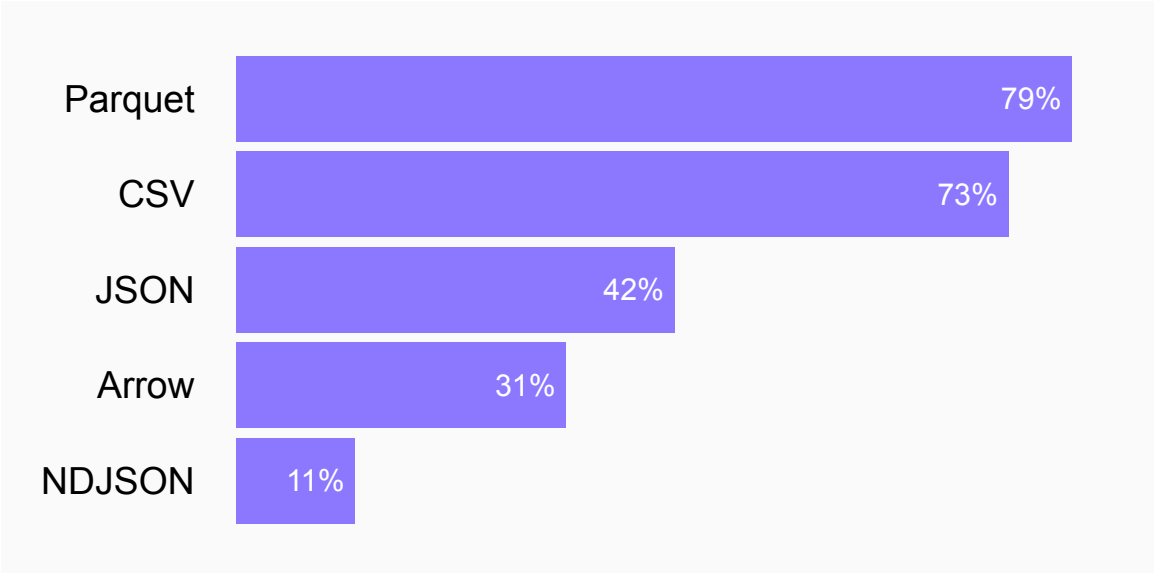
Regarding *where* the servers run, on-premises deployments and AWS are neck-and-neck with 27%. They are followed by two other clouds, Microsoft Azure and the Google Cloud Platform. Finally, about 4% of users run DuckDB on Hetzner servers.



Data

Data Formats

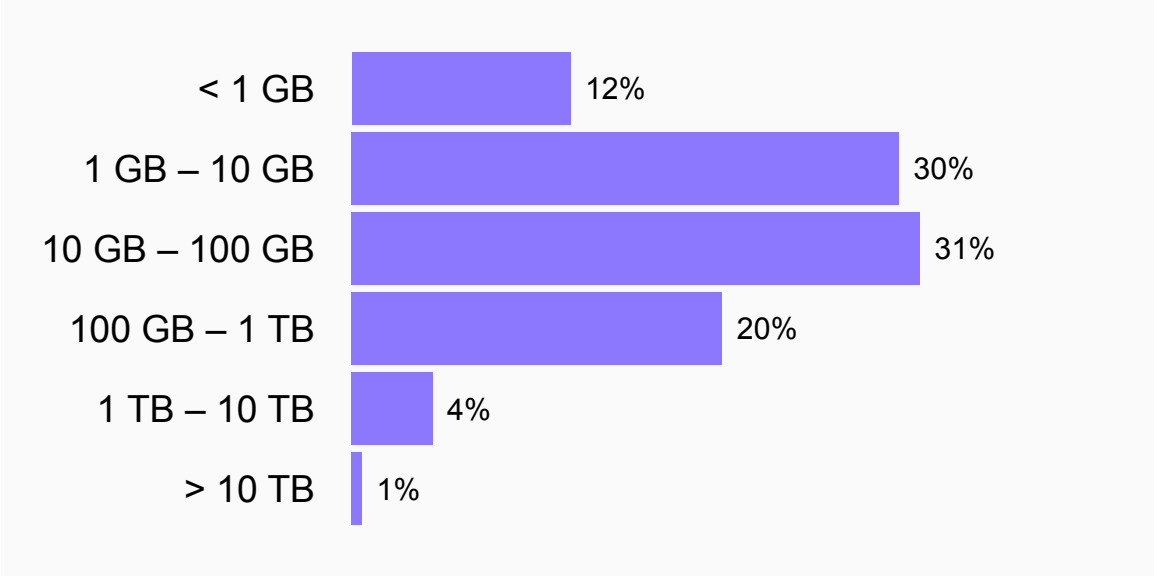
We inquired about the data formats used when working with DuckDB. Parquet is the most popular format: 79% of users reporting to use it. CSV is a close second with 73%. JSON is also popular with vanilla JSON achieving 42% and NDJSON 11%. About ⅓ reported to use Arrow.



Dataset Sizes

We asked users about the size of the largest dataset they processed with DuckDB. We defined *dataset size* as the size of the data when stored in uncompressed CSV format. For Parquet files and DuckDB database files, we asked users to approximate the CSV size by multiplying their file sizes by 5.

The responses showed that only a few respondents use DuckDB to process Big Data. For ¾ of users, their largest dataset size was less than 100 GB data, 20% of users processed a dataset between 100 GB and 1 TB, and approximately 5% of the users ventured into the 1 TB+ territory. About 1% processed 10 TB+ datasets. These findings are in line with statistics derived from a recent RedShift usage dataset by Jordan Tigani of MotherDuck, and the recent analysis of the Snowflake and RedShift datasets by George Fraser of Fivetran.

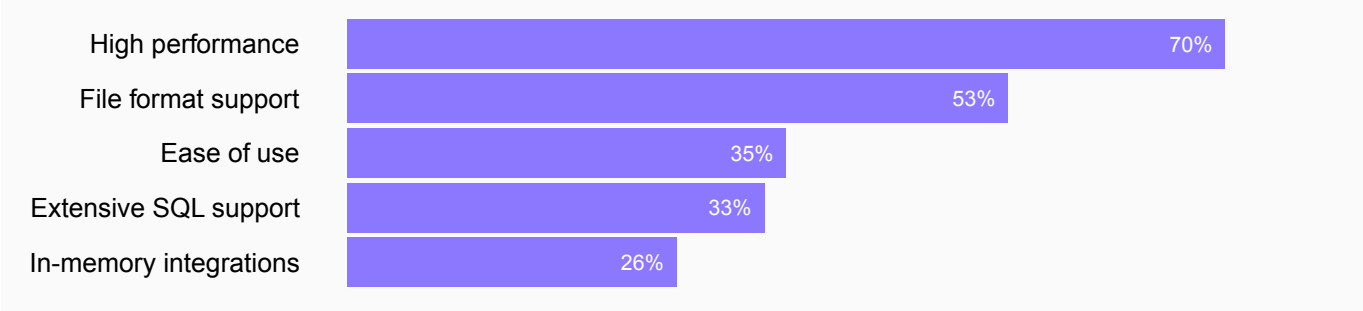


While these results obviously are somewhat biased – users who need to crunch through huge datasets may not work with DuckDB (yet!) –, the skew towards smaller datasets is quite significant and shows that many real-world use cases can be tackled using small to medium-sized datasets. The results also show that DuckDB *can* solve many problems on datasets larger than 1 TB.

Features

Most Liked Features

We were curious: what do users like most about DuckDB? The plot shows the most frequent responses:



The most liked feature is **high performance**. Users also enjoy **file format support** (CSV, Parquet, JSON, etc.), **ease of use**, **extensive SQL support** (including friendly SQL) and **in-memory integrations** such as support for Pandas, Arrow and NumPy. Finally, users mentioned low memory usage, protocol support (e.g., HTTPS, S3), database integrations, and portability.

Feature Requests

We asked users about the features that they'd most like to see in future DuckDB versions. The most popular requests are listed in the table below:

Feature	Percentage
Improved partitioning and optimizations related to partitioning	39%
Improved support for time series and optimizations for pre-sorted data	35%
Support for materialized views	28%
Support for vector search	24%
Support for attaching to database systems via ODBC	24%
Support for time travel queries (query the database as of a given time)	23%
Support for the Delta Lake format	22%
Improved support for Iceberg (including writes)	17%

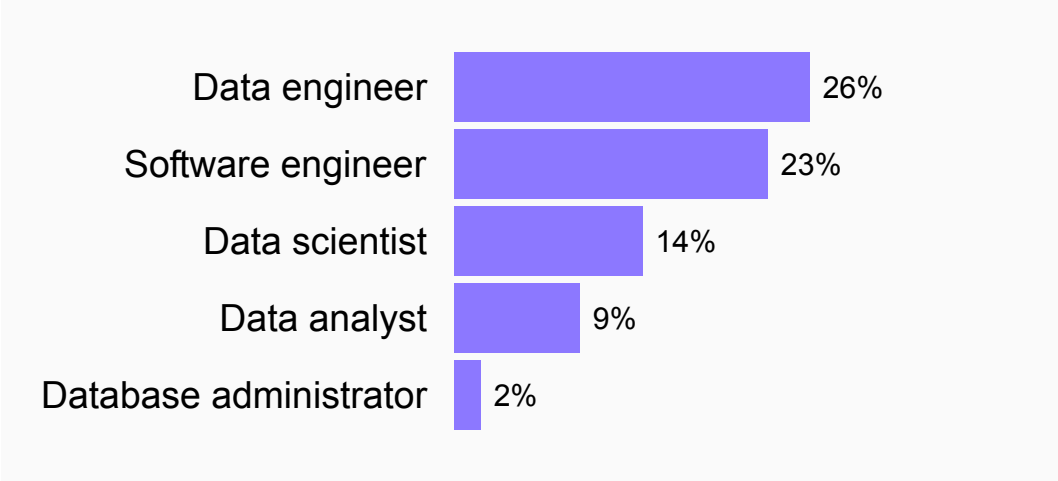
We are happy to report that, since the survey was conducted pre-v1.0.0 and DuckDB is now at version 1.1.1, some of these requests are already a reality:

- Reading Delta Lake is now possible via the `delta` extension.
- Vector search is now supported via the `vss` extension.

For the rest of the requested features, several ones are in the making at DuckLabs. Stay tuned!

User Roles

We asked respondents to indicate their main roles in their organization. The top-5 answers were as follows:



It's no surprise that DuckDB is popular in the “data” roles: 26% of the respondents are data engineers, 14% are data scientists, and 9% are data analysts. The form had a surprisingly high share of software engineers, 23%. Finally, about 2% of respondents indicated that their primary role is DBA.

Conclusion

We would like to thank all participants for taking the time to complete the survey. We will use the answers to guide the future development of DuckDB, and we hope that readers of this analysis find it informative as well.

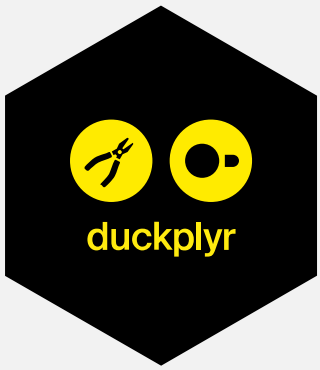
Analyzing Open Government Data with duckplyr

Publication date: 2024-10-09

Author: Hannes Mühleisen

TL;DR: We use the duckplyr R library to clean and analyze an Open Data set published by the government of New Zealand.

> For the duckplyr documentation, visit duckplyr.tidyverse.org.



Wrangling data by throwing SQL strings at it is not the most ergonomic way to perform interactive data analysis in R. For a while now, we have been working with the dplyr project team at Posit (formerly RStudio) and Kirill Müller to develop *duckplyr*. duckplyr is a high-performance drop-in replacement for dplyr, powered by DuckDB. You can read more about duckplyr in the announcement blog post. In this post, we are going to walk through a challenging real-world use case with duckplyr. For those of you wishing to follow along, we have prepared a Google Colab notebook with all the code snippets in this post. Timings reported below are also from Colab.

Like many government statistics agencies, New Zealand's "Stats NZ Tatauranga Aotearoa" thankfully provides some of the datasets they maintain as Open Data for download. The largest file available for download on that page contains "Age and sex by ethnic group (grouped total responses), for census usually resident population counts, 2006, 2013, and 2018 Censuses", CSV zipped file.

We can download that file (mirrored from our CDN, we don't want to DDoS poor Stats NZ) and unzip like so:

```
download.file("https://blobs.duckdb.org/nzcensus.zip", "nzcensus.zip")
unzip("nzcensus.zip")
```

Let's explore the CSV files in the zip and what their sizes are:

```
file.info(Sys.glob("*.csv"))["size"]
```

```
                            size
Data8277.csv           857672667
DimenLookupAge8277.csv      2720
DimenLookupArea8277.csv    65400
```

```
DimenLookupEthnic8277.csv       272
DimenLookupSex8277.csv           74
DimenLookupYear8277.csv          67
```

As we can see, there is one large (~800 MB) `Data` file and a bunch of `Dimen...` dimension files. This is a fairly common data layout, sometimes called a "star schema". From this, it's clear there are some joins in our future. But first lets focus on the main file, `Data8277.csv`. Reading sizeable CSV files is not trivial and can be very frustrating. But enough whinging, as the Kiwis would say.

To start with, let's just have a quick look what the file looks like:

```
cat(paste(readLines("Data8277.csv", n=10), collapse="\n"))
```

```
Year,Age,Ethnic,Sex,Area,count
2018,000,1,1,01,795
2018,000,1,1,02,5067
2018,000,1,1,03,2229
2018,000,1,1,04,1356
2018,000,1,1,05,180
2018,000,1,1,06,738
2018,000,1,1,07,630
2018,000,1,1,08,1188
2018,000,1,1,09,2157
```

So far this looks rather tame, there seem to be five columns. Thankfully, they have names. From just eyeballing the column values, it looks like they are all numeric and even integer values. However, looks can be deceiving, and the columns `Age`, `Area`, `count` contain character values somewhere down the line. Fun fact: we have to wait till line 431 741 until the `Area` column contains a non-integer value. Clearly we need a good CSV parser. R has no shortage of CSV readers, for example the `readr` package contains a flexible CSV parser. Reading this file with `readr` takes about a minute (on Colab).

But let's now start using DuckDB and duckplyr. First, we install duckplyr (and DuckDB, which is a dependency):

```
install.packages("duckplyr")
duckdb:::sql("SELECT version()")
```

This command prints out the installed DuckDB version, as of this writing the latest version on CRAN is 1.1.0. We can now use DuckDB's advanced data wrangling capabilities. First off, DuckDB contains probably the world's most advanced CSV parser. For the extra curious, here is a presentation on DuckDB's CSV parser. We use DuckDB's CSV reader to only read the first 10 rows from the CSV file:

```
duckdb:::sql("FROM Data8277.csv LIMIT 10")
```

```
   Year Age Ethnic Sex Area count
1  2018 000      1   1   01   795
2  2018 000      1   1   02  5067
3  2018 000      1   1   03  2229
4  2018 000      1   1   04  1356
5  2018 000      1   1   05   180
6  2018 000      1   1   06   738
7  2018 000      1   1   07   630
8  2018 000      1   1   08  1188
9  2018 000      1   1   09  2157
10 2018 000      1   1   12   177
```

This only takes a few milliseconds because DuckDB's CSV reader produces results in a streaming fashion, and because we have only requested 10 rows we are done fairly quickly.

DuckDB can also print out the schema it detected from the CSV file using the `DESCRIBE` keyword:

```
duckdb:::sql("DESCRIBE FROM Data8277.csv")
```

```
  column_name column_type ...
1        Year      BIGINT ...
2         Age     VARCHAR ...
3      Ethnic      BIGINT ...
4         Sex      BIGINT ...
5        Area     VARCHAR ...
6       count     VARCHAR ...
```

We can see that we have correctly detected the various data types for the columns. We can use the `SUMMARIZE` keyword to compute various summary statistics for all the columns in the file:

```
duckdb:::sql("SUMMARIZE FROM Data8277.csv")
```

This will take a little bit longer, but the results are very interesting:

```
# A tibble: 6 × 12
  column_name column_type min   max     approx_unique avg       std   q25   q50
  <chr>       <chr>       <chr> <chr>           <dbl> <chr>     <chr> <chr> <chr>
1 Year        BIGINT      2006  2018                3 2012.33… 4.92… 2006  2013
2 Age         VARCHAR     000   999999            149 NA        NA    NA    NA
3 Ethnic      BIGINT      1     9999               11 930.545… 2867… 3     6
4 Sex         BIGINT      1     9                   3 4.0       3.55… 1     2
5 Area        VARCHAR     001   DHB9999          2048 NA        NA    NA    NA
6 count       VARCHAR     ..C   9999            16825 NA        NA    NA    NA
# ℹ 3 more variables: q75 <chr>, count <dbl>, null_percentage <dbl>
```

This will show again the column names and their types, but also the summary statistics for minimum and maximum value, approximate count of unique values, average, standard deviations, 25, 50, and 75 quantiles, and percentage of NULL/NA values. So one gets a pretty good overview of what the data is like.

But we're not here to ogle summary statistics, we want to do actual analysis of the data. In this use case, we would like to compute the number of non-Europeans between 20 and 40 that live in the Auckland area using the 2018 census data and the results should be grouped by sex. To do so, we need to join the dimension CSV files with the main data file in order to properly filter the dimension values. In SQL, the lingua franca of large-scale data analysis, this looks like this:

We first join everything together:

```
FROM 'Data8277.csv' data
JOIN 'DimenLookupAge8277.csv' age ON data.Age = age.Code
JOIN 'DimenLookupArea8277.csv' area ON data.Area = area.Code
JOIN 'DimenLookupEthnic8277.csv' ethnic ON data.Ethnic = ethnic.Code
JOIN 'DimenLookupSex8277.csv' sex ON data.Sex = sex.Code
JOIN 'DimenLookupYear8277.csv' year ON data.Year = year.Code
```

Next, we use the `SELECT` projection to perform some basic renames and data cleaning:

```
SELECT
    year.Description AS year_,
    area.Description AS area_,
    ethnic.Description AS ethnic_,
    sex.Description AS sex_,
    TRY_CAST(replace(age.Description, ' years', '') AS INTEGER) AS age_,
    TRY_CAST(data.count AS INTEGER) AS count_
```

The data set contains various totals, so we remove them before proceeding:

```
WHERE count_ > 0
  AND age_ IS NOT NULL
  AND area_ NOT LIKE 'Total%'
  AND ethnic_ NOT LIKE 'Total%'
  AND sex_ NOT LIKE 'Total%'
```

We wrap the previous statements as a common-table-expression `expanded_cleaned_data`, and we can then compute the actual aggregation using DuckDB

```
SELECT sex_, sum(count_) AS group_count
FROM expanded_cleaned_data
WHERE age_ BETWEEN 20 AND 40
  AND area_ LIKE 'Auckland%'
  AND ethnic_ <> 'European'
  AND year_ = 2018
GROUP BY sex_
ORDER BY sex_
```

This takes ca. 20 s on the limited Colab free tier compute. The result is:

```
    sex_ group_count
1 Female      398556
2   Male      397326
```

So far, so good. However, writing SQL queries is not for everyone. The ergonomics of creating SQL strings in an interactive data analysis environment like R are questionable to say the least. Frameworks like `dplyr` have shown how data wrangling ergonomics can be massively improved. Let's express our analysis using dplyr then after first reading the data into RAM from CSV:

```
library(dplyr)

data   <- readr::read_csv("Data8277.csv")
age    <- readr::read_csv("DimenLookupAge8277.csv")
area   <- readr::read_csv("DimenLookupArea8277.csv")
ethnic <- readr::read_csv("DimenLookupEthnic8277.csv")
sex    <- readr::read_csv("DimenLookupSex8277.csv")
year   <- readr::read_csv("DimenLookupYear8277.csv")

expanded_cleaned_data <- data |>
  filter(grepl("^\\d+$", count)) |>
  mutate(count_ = as.integer(count)) |>
  filter(count_ > 0) |>
  inner_join(
    age |>
      filter(grepl("^\\d+ years$", Description)) |>
      mutate(age_ = as.integer(Code)),
    join_by(Age == Code)
  ) |>
  inner_join(area |>
    mutate(area_ = Description) |>
    filter(!grepl("^Total", area_)), join_by(Area == Code)) |>
  inner_join(ethnic |>
    mutate(ethnic_ = Description) |>
    filter(!grepl("^Total", ethnic_)), join_by(Ethnic == Code)) |>
  inner_join(sex |>
    mutate(sex_ = Description) |>
    filter(!grepl("^Total", sex_)), join_by(Sex == Code)) |>
  inner_join(year |> mutate(year_ = Description), join_by(Year == Code))

# create final aggregation, still completely lazily
twenty_till_forty_non_european_in_auckland_area <-
  expanded_cleaned_data |>
  filter(
    age_ >= 20, age_ <= 40,
    grepl("^Auckland", area_),
    year_ == "2018",
    ethnic_ != "European"
  ) |>
  summarise(group_count = sum(count_), .by = sex_) |> arrange(sex_)

print(twenty_till_forty_non_european_in_auckland_area)
```

This looks nicer and completes in ca. one minute, but there are several hidden issues. First, we read the *entire* dataset into RAM. While for this dataset this is likely possible because most computers have more than 1 GB of RAM, this will of course not work for larger datasets. Then, we execute a series of dplyr verbs. However, dplyr executes those eagerly, meaning it does not holistically optimize the sequence of verbs. For example, it cannot see that we are filtering out all non-European ethnicities in the last step and happily computes all of those for the intermediate result. The same happens with survey years that are not 2018, only in the last step we filter those out. We have computed an expensive join on all other years for nothing. Depending on data distributions, this can be extremely wasteful. And yes, it is possible to manually move the filters around but this is tedious and error-prone. At least the result is exactly the same as the SQL version above:

```
# A tibble: 2 × 2
  sex_   group_count
  <chr>        <int>
```

```
1 Female      398556
2 Male        397326
```

Now we switch the exact same script over to duckplyr. Instead of reading the CSV files into RAM entirely using `readr`, we instead use the `duckplyr_df_from_csv` function from `duckplyr`:

```
library("duckplyr")

data   <- duckplyr_df_from_csv("Data8277.csv")
age    <- duckplyr_df_from_csv("DimenLookupAge8277.csv")
area   <- duckplyr_df_from_csv("DimenLookupArea8277.csv")
ethnic <- duckplyr_df_from_csv("DimenLookupEthnic8277.csv")
sex    <- duckplyr_df_from_csv("DimenLookupSex8277.csv")
year   <- duckplyr_df_from_csv("DimenLookupYear8277.csv")
```

This takes exactly 0 seconds, because duckplyr is not actually doing much. We detect the schema of the CSV files using our award-winning "sniffer", and create the six placeholder objects for each of those files. Part of the unique design of duckplyr is that those objects are "Heisenbergian", they behave like completely normal R `data.frame`s once they are treated as such, but they can *also* act as lazy evaluation placeholders when they are passed to downstream analysis steps. This is made possible by a little-known R feature known as `ALTREP` which allows R vectors to be computed on-demand among other things.

Now we re-run the exact same dplyr pipeline as above. Only this time we are "done" in less than a second. This is because all we have done is *lazily* constructing a so-called relation tree which encapsulates the entirety of the transformations. This allows *holistic* optimization, for example pushing the year and ethnicity all the way down to the reading of the CSV file *before* joining. We can also eliminate the reading of columns that are not used in the query at all.

Only when we finally print the result

```
print(twenty_till_forty_non_european_in_auckland_area)
```

actual computation is triggered. This finishes in the same time as the hand-rolled SQL query above, only that this time we had a much more pleasant experience from using the dplyr syntax. And, thankfully, the result is still exactly the same.

This use case was also presented as part of my keynote at this year's posit::conf:

Finally, we should note that duckplyr is still being developed. We have taken great care in not breaking anything and will fall back on the existing dplyr implementation if anything cannot be run in DuckDB (yet). But we would love to hear from you if anything does not work as expected.

DuckDB Tricks – Part 2

Publication date: 2024-10-11

Author: Gábor Szárnyas

TL;DR: We continue our "DuckDB tricks" series, focusing on queries that clean, transform and summarize data.

Overview

This post is the latest installment of the DuckDB Tricks series, where we show you nifty SQL tricks in DuckDB. Here's a summary of what we're going to cover:

Operation	SQL instructions
Fixing timestamps in CSV files	`regexp_replace()`{:.language-sql .highlight} and `strptime()`{:.language-sql .highlight}
Filling in missing values	`CROSS JOIN`{:.language-sql .highlight}, `LEFT JOIN`{:.language-sql .highlight} and `coalesce()`{:.language-sql .highlight}
Repeated transformation steps	`CREATE OR REPLACE TABLE t AS ... FROM t ...`{:.language-sql .highlight}
Computing checksums for columns	`bit_xor(md5_number(COLUMNS(*)::VARCHAR))`{:.language-sql .highlight}
Creating a macro for checksum	`CREATE MACRO checksum(tbl) AS TABLE ...`{:.language-sql .highlight}

Dataset

For our example dataset, we'll use `schedule.csv`, a hand-written CSV file that encodes a conference schedule. The schedule contains the timeslots, the locations and the events scheduled.

```
timeslot,location,event
2024-10-10 9am,room Mallard,Keynote
2024-10-10 10.30am,room Mallard,Customer stories
2024-10-10 10.30am,room Fusca,Deep dive 1
2024-10-10 12.30pm,main hall,Lunch
2024-10-10 2pm,room Fusca,Deep dive 2
```

Fixing Timestamps in CSV Files

As usual in real use case, the input CSV is messy with irregular timestamps such as `2024-10-10 9am`. Therefore, if we load the `schedule.csv` file using DuckDB's CSV reader, the CSV sniffer will detect the first column as a VARCHAR field:

```
CREATE TABLE schedule_raw AS
    SELECT * FROM 'https://duckdb.org/data/schedule.csv';

SELECT * FROM schedule_raw;
```

timeslot varchar	location varchar	event varchar
2024-10-10 9am	room Mallard	Keynote
2024-10-10 10.30am	room Mallard	Customer stories
2024-10-10 10.30am	room Fusca	Deep dive 1
2024-10-10 12.30pm	main hall	Lunch
2024-10-10 2pm	room Fusca	Deep dive 2

Ideally, we would like the `timeslot` column to have the type `TIMESTAMP` so we can treat it as a timestamp in the queries later. To achieve this, we can use the table we just loaded and fix the problematic entities by using a regular expression-based search and replace operation, which unifies the format to `hours.minutes` followed by `am` or `pm`. Then, we convert the string to timestamps using `strptime` with the `%p` format specifier capturing the `am`/`pm` part of the string.

```
CREATE TABLE schedule_cleaned AS
    SELECT
        timeslot
            .regexp_replace(' (\d+)(am|pm)$', ' \1.00\2')
            .strptime('%Y-%m-%d %H.%M%p') AS timeslot,
        location,
        event
    FROM schedule_raw;
```

Note that we use the dot operator for function chaining to improve readability. For example, `regexp_replace(string, pattern, replacement)` is formulated as `string.regexp_replace(pattern, replacement)`. The result is the following table:

timeslot timestamp	location varchar	event varchar
2024-10-10 09:00:00	room Mallard	Keynote
2024-10-10 10:30:00	room Mallard	Customer stories
2024-10-10 10:30:00	room Fusca	Deep dive 1
2024-10-10 12:30:00	main hall	Lunch
2024-10-10 14:00:00	room Fusca	Deep dive 2

Filling in Missing Values

Next, we would like to derive a schedule that includes the full picture: *every timeslot* for *every location* should have its line in the table. For the timeslot-location combinations, where there is no event specified, we would like to explicitly add a string that says `<empty>`.

To achieve this, we first create a table `timeslot_location_combinations` containing all possible combinations using a `CROSS JOIN`. Then, we can connect the original table on the combinations using a `LEFT JOIN`. Finally, we replace `NULL` values with the `<empty>` string using the `coalesce` function.

> The `CROSS JOIN` clause is equivalent to simply listing the tables in the `FROM` clause without specifying join conditions. By explicitly spelling out `CROSS JOIN`, we communicate that we intend to compute a Cartesian product – which is an expensive operation on large tables and should be avoided in most use cases.

```
CREATE TABLE timeslot_location_combinations AS
    SELECT timeslot, location
    FROM (SELECT DISTINCT timeslot FROM schedule_cleaned)
    CROSS JOIN (SELECT DISTINCT location FROM schedule_cleaned);

CREATE TABLE schedule_filled AS
    SELECT timeslot, location, coalesce(event, '<empty>') AS event
    FROM timeslot_location_combinations
    LEFT JOIN schedule_cleaned
```

```
        USING (timeslot, location)
    ORDER BY ALL;

SELECT * FROM schedule_filled;
```

timeslot timestamp	location varchar	event varchar
2024-10-10 09:00:00	main hall	<empty>
2024-10-10 09:00:00	room Fusca	<empty>
2024-10-10 09:00:00	room Mallard	Keynote
2024-10-10 10:30:00	main hall	<empty>
2024-10-10 10:30:00	room Fusca	Deep dive 1
2024-10-10 10:30:00	room Mallard	Customer stories
2024-10-10 12:30:00	main hall	Lunch
2024-10-10 12:30:00	room Fusca	<empty>
2024-10-10 12:30:00	room Mallard	<empty>
2024-10-10 14:00:00	main hall	<empty>
2024-10-10 14:00:00	room Fusca	Deep dive 2
2024-10-10 14:00:00	room Mallard	<empty>
12 rows		3 columns

We can also put everything together in a single query using a `WITH` clause:

```
WITH timeslot_location_combinations AS (
    SELECT timeslot, location
    FROM (SELECT DISTINCT timeslot FROM schedule_cleaned)
    CROSS JOIN (SELECT DISTINCT location FROM schedule_cleaned)
)
SELECT timeslot, location, coalesce(event, '<empty>') AS event
FROM timeslot_location_combinations
LEFT JOIN schedule_cleaned
    USING (timeslot, location)
ORDER BY ALL;
```

Repeated Data Transformation Steps

Data cleaning and transformation usually happens as a sequence of transformations that shape the data into a form that's best fitted to later analysis. These transformations are often done by defining newer and newer tables using `CREATE TABLE ... AS SELECT` statements.

For example, in the sections above, we created `schedule_raw`, `schedule_cleaned`, and `schedule_filled`. If, for some reason, we want to skip the cleaning steps for the timestamps, we have to reformulate the query computing `schedule_filled` to use `schedule_raw` instead of `schedule_cleaned`. This can be tedious and error-prone, and it results in a lot of unused temporary data – data that may accidentally get picked up by queries that we forgot to update!

In interactive analysis, it's often better to use the same table name by running `CREATE OR REPLACE` statements:

```
CREATE OR REPLACE TABLE <table_name> AS
    ...
    FROM <table_name>
    ...;
```

Using this trick, we can run our analysis as follows:

```
CREATE OR REPLACE TABLE schedule AS
    SELECT * FROM 'https://duckdb.org/data/schedule.csv';

CREATE OR REPLACE TABLE schedule AS
    SELECT
```

```
        timeslot
            .regexp_replace(' (\d+)(am|pm)$', ' \1.00\2')
            .strptime('%Y-%m-%d %H.%M%p') AS timeslot,
        location,
        event
    FROM schedule;

CREATE OR REPLACE TABLE schedule AS
    WITH timeslot_location_combinations AS (
        SELECT timeslot, location
        FROM (SELECT DISTINCT timeslot FROM schedule)
        CROSS JOIN (SELECT DISTINCT location FROM schedule)
    )
    SELECT timeslot, location, coalesce(event, '<empty>') AS event
    FROM timeslot_location_combinations
    LEFT JOIN schedule
        USING (timeslot, location)
    ORDER BY ALL;

SELECT * FROM schedule;
```

Using this approach, we can skip any step and continue the analysis without adjusting the next one.

What's more, our script can now be re-run from the beginning without explicitly deleting any tables: the `CREATE OR REPLACE` statements will automatically replace any existing tables.

Computing Checksums for Columns

It's often beneficial to compute a checksum for each column in a table, e.g., to see whether a column's content has changed between two operations. We can compute a checksum for the `schedule` table as follows:

```
SELECT bit_xor(md5_number(COLUMNS(*)::VARCHAR))
FROM schedule;
```

What's going on here? We first list columns (`COLUMNS(*)`) and cast all of them to VARCHAR values. Then, we compute the numeric MD5 hashes with the `md5_number` function and aggregate them using the `bit_xor` aggregate function. This produces a single `HUGEINT` (`INT128`) value per column that can be used to compare the content of tables.

If we run this query in the script above, we get the following results:

```
┌──────────────────────────────────────┬──────────────────────────────────────┬──────────────────────────────────────┐
│               timeslot               │               location               │                event                 │
│                int128                │                int128                │                int128                │
├──────────────────────────────────────┼──────────────────────────────────────┼──────────────────────────────────────┤
│ -134063647976146309049043791223896883700 │ 85181227364560750048971459330392988815 │
-650144045653398519678796832146127680 44 │
└──────────────────────────────────────┴──────────────────────────────────────┴──────────────────────────────────────┘

┌──────────────────────────────────────┬──────────────────────────────────────┬──────────────────────────────────────┐
│               timeslot               │               location               │                event                 │
│                int128                │                int128                │                int128                │
├──────────────────────────────────────┼──────────────────────────────────────┼──────────────────────────────────────┤
│ 629010110167473189774697785178456459 61 │ 85181227364560750048971459330392988815 │
-650144045653398519678796832146127680 44 │
└──────────────────────────────────────┴──────────────────────────────────────┴──────────────────────────────────────┘

┌──────────────────────────────────────┬──────────┬──────────────────────────────────────┐
│               timeslot               │ location │                event                 │
│                int128                │ int128   │                int128                │
└──────────────────────────────────────┴──────────┴──────────────────────────────────────┘
```

```
┌──────────────────────────────────────────┬─────────┬──────────────────────────────────────────┐
│ -162418013182718436871288818115274808663 │       0 │ -135609337521255080720676586176293337793 │
└──────────────────────────────────────────┴─────────┴──────────────────────────────────────────┘
```

Creating a Macro for the Checksum Query

We can turn the checksum query into a table macro with the new `query_table` function:

```
CREATE MACRO checksum(table_name) AS TABLE
    SELECT bit_xor(md5_number(COLUMNS(*)::VARCHAR))
    FROM query_table(table_name);
```

This way, we can simply invoke it on the `schedule` table as follows (also leveraging DuckDB's FROM-first syntax):

```
FROM checksum('schedule');
```

```
┌──────────────────────────────────────────┬──────────────────────────────────────────┬──────────────────────────────────────────┐
│                 timeslot                 │                 location                 │                  event                   │
│                  int128                  │                  int128                  │                  int128                  │
├──────────────────────────────────────────┼──────────────────────────────────────────┼──────────────────────────────────────────┤
│ -134063647976146309049043791223896883700 │ 85181227364560750048971459330392988815 │ -65014404565339851967879683214612768044 │
└──────────────────────────────────────────┴──────────────────────────────────────────┴──────────────────────────────────────────┘
```

Closing Thoughts

That's it for today! We'll be back soon with more DuckDB tricks and case studies. In the meantime, if you have a trick that would like to share, please share it with the DuckDB team on our social media sites, or submit it to the DuckDB Snippets site (maintained by our friends at MotherDuck).

Driving CSV Performance: Benchmarking DuckDB with the NYC Taxi Dataset

Publication date: 2024-10-16

Author: Pedro Holanda

TL;DR: DuckDB's benchmark suite now includes the NYC Taxi Benchmark. We explain how our CSV reader performs on the Taxi Dataset and provide steps to reproduce the benchmark.

The NYC taxi dataset is a collection of many years of taxi rides that occurred in New York City. It is a very influential dataset, used for database benchmarks, machine learning, data visualization, and more.

In 2022, the data provider has decided to distribute the dataset as a series of Parquet files instead of CSV files. Performance-wise, this is a wise choice, as Parquet files are much smaller than CSV files, and their native columnar format allows for fast execution directly on them. However, this change hinders the number of systems that can natively load the files.

In the "Billion Taxi Rides in Redshift" blog post, a new database benchmark is proposed to evaluate the performance of aggregations over the taxi dataset. The dataset is also joined and denormalized with other datasets that contain information about the weather, cab types, and pickup/dropoff locations. It is then stored as multiple compressed, gzipped CSV files, each containing 20 million rows.

The Taxi Data Set as CSV Files

Since DuckDB is well-known for its CSV reader performance, we were intrigued to explore whether the loading process of this benchmark could help us identify new performance bottlenecks in our CSV loader. This curiosity led us on a journey to generate these datasets and analyze their performance in DuckDB. According to the recent study conducted on the AWS RedShift fleet, CSV files are the most used external source data type in S3, and 99% of them are gzipped. Therefore, the fact that the proposed benchmark also used split gzipped files caught my attention.

In this blog post, we'll guide you through how to run this benchmark in DuckDB and discuss some lessons learned and future ideas for our CSV Reader. The dataset used in this benchmark is publicly available. The dataset is partitioned and distributed as a collection of 65 gzipped CSV files, each containing 20 million rows and totaling up to 1.8 GB per file. The total dataset is 111 GB compressed and 518 GB uncompressed. We also provide more details on how we generated this dataset and highlight the differences between the dataset we distribute and the original one described in the "Billion Taxi Rides in Redshift" blog post.

Reproducing the Benchmark

Doing fair benchmarking is a difficult problem, especially when the data, queries, and results used for the benchmark are not easy to access and run. We have made the benchmark discussed in this blog post easy to run by providing scripts available in the `taxi-benchmark` GitHub repository.

This repository contains three main Python scripts:

1. `generate_prepare_data.py`: Downloads all necessary files and prepares them for the benchmark.
2. `benchmark.py`: Runs the benchmark and performs result verification.
3. `analyse.py`: Analyzes the benchmark results and produces some of the insights discussed in this blog post.

The benchmark is not intended to be flawless – no benchmark is. However, we believe that sharing these scripts is a positive step, and we welcome any contributions to make them cleaner and more efficient.

The repository also includes a README file with detailed instructions on how to use it. This repository will serve as the foundation for the experiments conducted in this blog post.

Preparing the Dataset

To start, you first need to download and prepare the files by executing `python generate_prepare_data.py`. This will download all 65 files to the `./data` folder. Additionally, the files will be uncompressed and combined into a single large file.

As a result, the `./data` folder will have 65 gzipped CSV files (i.e., from `trips_xaa.csv.gz` to `trips_xcm.csv.gz`) and a single large uncompressed CSV file containing the full data (i.e., `decompressed.csv`).

Our benchmark then run in two different settings:

1. Over 65 compressed files.
2. Over a single uncompressed file.

Once the files have been prepared, you can run the benchmark by running `python benchmark.py`.

Loading

The loading phase of the benchmark runs six times for each benchmark setting. From the first five runs, we take the median loading time. During the sixth run, we collect resource usage data (e.g., CPU usage and disk reads/writes).

Loading is performed using an in-memory DuckDB instance, meaning the data is not persisted to DuckDB storage and only exists while the connection is active. This is important to note because, as the dataset does not fit in memory and is spilled into a temporary space on disk. The decision to not persist the data has a substantial impact on performance: it makes loading the dataset significantly faster, while querying it will be somewhat slower as DuckDB will use an uncompressed representation. We made this choice for the benchmark since our primary focus is on testing the CSV loader rather than the queries.

Our table schema is defined in `schema.sql`.

The loader for the 65 files uses the following query:

```
COPY trips FROM 'data/trips_*.csv.gz' (HEADER false);
```

The loader for the single uncompressed file uses this query:

```
COPY trips FROM 'data/decompressed.csv' (HEADER false);
```

Querying

After loading, the benchmark script will run each of the benchmark queries five times to measure their execution time. It is also important to note that the results of the queries are validated against their corresponding answers. This allows us to verify the correctness of the benchmark. Additionally, the queries are identical to those used in the original "Billion Taxi Rides" benchmark.

Results

Loading Time

Although we are talking about many rows of a CSV file with 51 columns, DuckDB can ingest them rather fast.

Note that, by default, DuckDB preserves the insertion order of the data, which negatively impacts performance. In the following results, all datasets have been loaded with this option set to `false`.

```
SET preserve_insertion_order = false;
```

All experiments were run on my Apple M1 Max with 64 GB of RAM, and we compare the loading times for a single uncompressed CSV file, and the 65 compressed CSV files.

Name	Time (min)	Avg deviation of CPU usage from 100%
Single File – Uncompressed	11:52	31.57
Multiple Files – Compressed	13:52	27.13

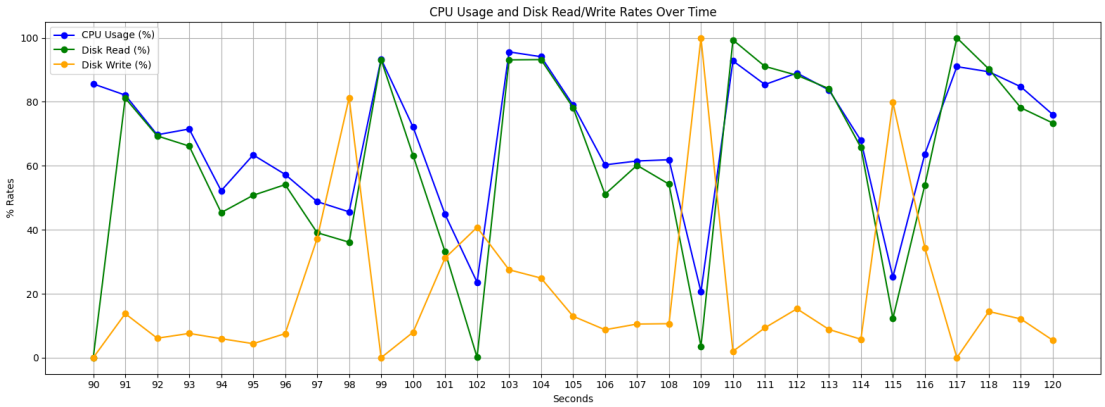
Unsurprisingly, loading data from multiple compressed files is more CPU-efficient than loading from a single uncompressed file. This is evident from the lower average deviation in CPU usage for multiple compressed files, indicating fewer wasted CPU cycles. There are two main reasons for this: (1) The compressed files are approximately eight times smaller than the uncompressed file, drastically reducing the amount of data that needs to be loaded from disk and, consequently, minimizing CPU stalls while waiting for data to be processed. (2) It is much easier to parallelize the loading of multiple files than a single file, as each thread can handle on a single file.

The difference in CPU efficiency is also reflected in execution times: reading from a single uncompressed file is 2 minutes faster than reading from multiple compressed files. The reason for this lies in our decompression algorithm, which is admittedly not optimally designed. Reading a compressed file involves three tasks: (1) loading data from disk into a compressed buffer, (2) decompressing that data into a decompressed buffer, and (3) processing the decompressed buffer. In our current implementation, tasks 1 and 2 are combined into a single operation, meaning we cannot continue reading until the current buffer is fully decompressed, resulting in idle cycles.

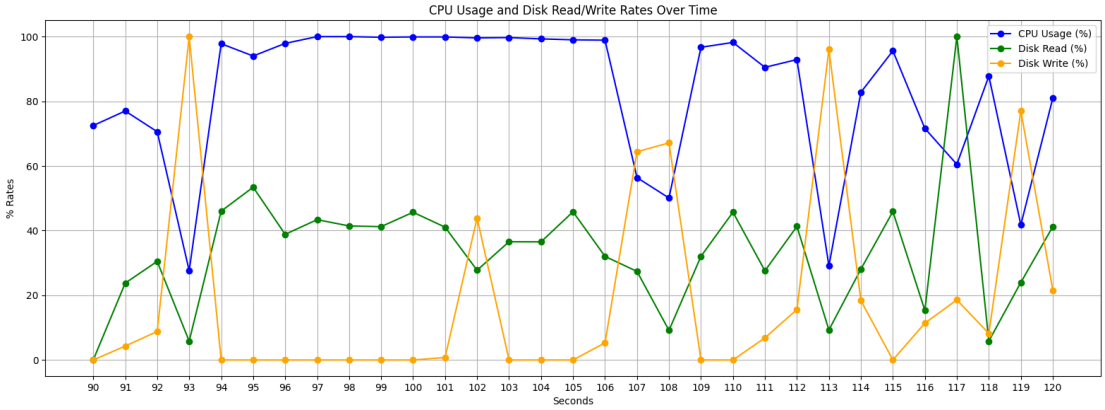
Under the Hood

We can also see what happens under the hood to verify our conclusion regarding the loading time.

In the figure below, you can see a snapshot of CPU and disk utilization for the “Single File – Uncompressed” run. We observe that achieving 100% CPU utilization is challenging, and we frequently experience stalls due to data writes to disk, as we are creating a table from a dataset that does not fit into our memory. Another key point is that CPU utilization is closely tied to disk reads, indicating that our threads often wait for data before processing it. Implementing async IO for the CSV Reader/Writer could significantly improve performance for parallel processing, as a single thread could handle most of our disk I/O without negatively affecting CPU utilization.



Below, you can see a similar snapshot for loading the 65 compressed files. We frequently encounter stalls during data writes; however, CPU utilization is significantly better because we wait less time for the data to load (remember, the data is approximately 8 times smaller than in the uncompressed case). In this scenario, parallelization is also much easier. Like in the uncompressed case, these gaps in CPU utilization could be mitigated by async I/O, with the addition of a decomposed decompression algorithm.



Query Times

For completeness, we also provide the results of the four queries on a MacBook Pro with an M1 Pro CPU. This comparison demonstrates the time differences between querying a database that does not fit in memory using a purely in-memory connection (i.e., without storage) versus one where the data is first loaded and persisted in the database.

Name	Time – without storage (s)	Time – with storage (s)
Q 01	2.45	1.45
Q 02	3.89	0.80
Q 03	5.21	2.20
Q 04	11.2	3.12

The main difference between these times is that when DuckDB uses a storage file, the data is highly compressed, resulting in much faster access when querying the dataset. In contrast, when we do not use persistent storage, our in-memory database temporarily stores data in an uncompressed `.tmp` file to allow for memory overflow, which increases disk I/O and leads to slower query results. This observation raises a potential area for exploration: determining whether applying compression to temporary data would be beneficial.

How This Dataset Was Generated

The original blog post generated the dataset using CSV files distributed by the NYC Taxi and Limousine Commission. Originally, these files included precise latitude and longitude coordinates for pickups and drop-offs. However, starting in mid-2016, these precise coordinates were anonymized using pickup and drop-off geometry objects to address privacy concerns. (There are even stories of broken marriages resulting from checking the actual destinations of taxis.) Furthermore, in recent years, the TLC decided to redistribute the data as Parquet files and to fully anonymize these data points, including data prior to mid-2016.

This is a problem, as the dataset from the "Billion Taxi Rides in Redshift" blog post relies on having this detailed information. Let's take the following snippet of the data:

```
649084905,VTS,2012-08-31 22:00:00,2012-08-31 22:07:00,0,1,-73.993908,40.741383000000006,-
73.989915,40.75273800000001,1,1.32,6.1,0.5,0.5,0,0,0,0,7.1,CSH,0,0101000020E6100000E6CE4C309C7F52C0BA675DA3E55E4440,
Yards-Chelsea-Flatiron-Union Square,3807,132,109,1,Manhattan,010900,1010900,I,MN17,Midtown-Midtown
South,3807
```

We see precise longitude and latitude data points: `-73.993908`, `40.741383000000006`, `-73.989915`, `40.75273800000001`, along with a PostGIS Geometry hex blob created from this longitude and latitude information: `0101000020E6100000E6CE4C309C7F52C0BA675DA3E55E4440`,

`0101000020E610000078B471C45A7F52C06D3A02B859604440` (generated as `ST_SetSRID(ST_Point(longitude, latitude), 4326)`).

Since this information is essential to the dataset, producing files as described in the "Billion Taxi Rides in Redshift" blog post is no longer feasible due to the missing detailed location data. However, the internet never forgets. Hence, we located instances of the original dataset distributed by various sources, such as [1], [2], and [3]. Using these sources, we combined the original CSV files with weather information from the scripts referenced in the "Billion Taxi Rides in Redshift" blog post.

How Does This Dataset Differ from the Original One?

There are two significant differences between the dataset we distribute and the one from the "Billion Taxi Rides in Redshift" blog post:

1. Our dataset includes data up to the last date that longitude and latitude information was available (June 30, 2016), whereas the original post only included data up to the end of 2015 (understandable, as the post was written in February 2016).
2. We also included Uber trips, which were excluded from the original post.

If you wish to run the benchmark with a dataset as close to the original as possible, you can generate a new table by filtering out the additional data. For example:

```
CREATE TABLE trips_og AS
    FROM trips
    WHERE pickup_datetime < '2016-01-01'
      AND cab_type != 'uber';
```

Conclusion

In this blog post, we discussed how to run the taxi benchmark on DuckDB, and we've made all scripts available so you can benchmark your preferred system as well. We also demonstrated how this highly relevant benchmark can be used to evaluate our operators and gain insights into areas for further improvement.

What's New in the Vector Similarity Search Extension?

Publication date: 2024-10-23

Author: Max Gabrielsson

TL;DR: DuckDB is another step closer to becoming a vector database! In this post, we show the new performance optimizations implemented in the vector search extension.

In the previous blog post, we introduced the DuckDB Vector Similarity Search (VSS) extension. While the extension is still quite experimental, we figured it would be interesting to dive into the details of some of the new features and improvements that we've been working on since the initial release.

Indexing Speed Improvements

As previously documented, creating an HNSW (Hierarchical Navigable Small Worlds) index over an already populated table is much more efficient than first creating the index and then inserting into the table. This is because it is much easier to predict how large the index will be if the total amount of rows are known up-front, which makes its possible to divide the work into chunks large enough to distribute over multiple threads. However, in the initial release this work distribution was a bit too coarse-grained as we would only schedule an additional worker thread for each *row group* (about 120,000 rows by default) in the table.

We've now introduced an extra buffer step in the index creation pipeline which enables more fine-grained work distribution, smarter memory allocation and less contention between worker threads. This results in much higher CPU saturation and a significant speedup when building HNSW indexes in environments with many threads available, regardless of how big or small the underlying table is.

Another bonus of this change is that we can now emit a progress bar when building the index, which is a nice touch when you still need to wait a while for the index creation to finish (despite the now much better use of system resources!).

New Distance Functions

In the initial release of VSS we supported three different distance functions: `array_distance`, `array_cosine_similarity` and `array_inner_product`. However, only the `array_distance` function is actually a *distance* function in that it returns results closer to 0 when the vectors are similar, and close to 1 when they are dissimilar, in contrast to, e.g., `array_cosine_similarity` that returns 1 when the vectors are identical. Oops!

To remedy this we've introduced two new distances:

- `array_cosine_distance`, equivalent to `1 - array_cosine_simililarity`
- `array_negative_inner_product`equivalent to `-array_inner_product`

These will now be accelerated with the use of the HNSW index instead, making the query patterns and ordering consistent for all supported metrics regardless if you make use of the HNSW index or not. Additionally, if you have an HNSW using, e.g., the `cosine` metric and write a top-k style query using `1 - array_cosine_similarity` as the ranking criterium, the optimizer should be able to normalize the expression to `array_cosine_distance` and use the index for this function as well.

For completeness we've also added the equivalent distance functions for the dynamically-sized `LIST` datatype (prefixed with `list_` instead of `array_`) and changed the `<=>` binary operator to now be an alias of `array_cosine_distance`, matching the semantics of the `pgvector` extension for PostgreSQL.

Index Accelerated "Top-K" Aggregates

Another cool thing that's happened in core DuckDB since last time is that DuckDB now has extra overloads for the `min_by` and `max_by` aggregate functions (and their aliases `arg_min` and `arg_max`) These new overloads take an optional third n argument that specifies the number of top-k (or top-n) elements to keep and outputs them into a sorted `LIST` value. Here's an example:

```
-- Create a table with some example data
CREATE OR REPLACE TABLE vecs AS
    SELECT
        row_number() OVER () AS id,
        [a, b, c]::FLOAT[3] AS vec
    FROM
        range(1,4) AS x(a), range(1,4) AS y(b), range(1,4) AS z(c);

-- Find the top 3 rows with the vector closest to [2, 2, 2]
SELECT
    arg_min(vecs, array_distance(vec, [2, 2, 2]::FLOAT[3]), 3)
FROM
    vecs;
```

```
[{'id': 14, 'vec': [2.0, 2.0, 2.0]}, {'id': 13, 'vec': [2.0, 1.0, 2.0]}, {'id': 11, 'vec': [1.0, 2.0, 2.0]}]
```

Of course, the VSS extension now includes optimizer rules to use to the HNSW index to accelerate these top-k aggregates when the ordering input is a distance function that references an indexed vector column, similarly to the `SELECT a FROM b ORDER BY array_distance(a.vec, query_vec) LIMIT k` query pattern that we discussed in the previous blog post. These new overloads allow you to express the same query in a more concise and readable way, while still avoiding the need for a full scan and sort of the underlying table (as long as the table has a matching HNSW index).

Index Accelerated LATERAL Joins

After running some benchmarking on the initial version of VSS, we realized that even though index-lookups on our HNSW index is really fast (thanks to the USearch library that it is based on!), using DuckDB to search for individual vectors at a time has a lot of latency compared to other solutions. The reasons for this are many and nuanced, but we want to be clear that our choice of HNSW implementation, USearch, is not the bottleneck here as profiling revealed only about 2% of the runtime is actually spent inside of usearch.

Instead, most of the per-query overhead comes from the fact that DuckDB is just not optimized for *point queries,* i.e., queries that only really fetch and process a single row. Because DuckDB is based on a vectorized execution engine, the smallest unit of work is not 1 row but 2,048, and because we expect to crunch through a ton of data, we generally favor spending a lot of time up front to optimize the query plan and pre-allocate large buffers and caches so that everything is as efficient as possible once we start executing. But a lot of this work becomes unnecessary when the actual working set is so small. For example, is it really worthwhile to inspect and hash every single element of a constant 768-long query vector to attempt to look for common subexpressions if you know there is only going to be a handful of rows in the result?

While we have some ideas on how to improve this scenario in the future, we decided to take another approach for now and instead try focus not on our weaknesses, but on our strengths. That is, crunching through a ton of data! So instead of trying to optimize the "1:N", i.e., "given this one embedding, give me the closes N embeddings" query, what if we instead focused on the "N:M", "given all these N embeddings, pair them up with the closest M embeddings each". What would that look like? Well, that would be a `LATERAL` join of course!

Basically, we are now able to make use of HNSW indexes to accelerate `LATERAL` joins where the "inner" query looks just like the top-k style queries we normally target, for example:

```
SELECT a
FROM b
ORDER BY array_distance(a.vec, query_vec)
LIMIT k;
```

But where the `query_vec` array is now a reference to an "outer" join table. The only requirement is for the inner table to have an HNSW index on the vector column matching the distance function. Here's an example:

```
-- Set the random seed for reproducibility
SELECT setseed(0.42);

-- Create some example tables
CREATE TABLE queries AS
    SELECT
        i AS id,
        [random(), random(), random()]::FLOAT[3] AS embedding
    FROM generate_series(1, 10_000) r(i);

CREATE TABLE items AS
    SELECT
        i AS id,
        [random(), random(), random()]::FLOAT[3] AS embedding
FROM generate_series(1, 10_000) r(i);

-- Collect the 5 closest items to each query embedding
SELECT queries.id AS id, list(inner_id) AS matches
    FROM queries, LATERAL (
        SELECT
            items.id AS inner_id,
            array_distance(queries.embedding, items.embedding) AS dist
        FROM items
        ORDER BY dist
        LIMIT 5
    )
GROUP BY queries.id;
```

Executing this on my Apple M3 Pro-equipped MacBook with 36 GB memory takes about 10 seconds.

If we `EXPLAIN` this query plan, we'll see a lot of advanced operators:

```
PRAGMA explain_output = 'optimized_only';
EXPLAIN ...
```

While this plan looks very complicated, the most worrysome among these operators is the `CROSS_PRODUCT` towards the bottom of the plan, which blows up the expected cardinality and is a sign that we are doing a lot of work that we probably don't want to do. However, if we create an HNSW index on the `items` table using

```
CREATE INDEX my_hnsw_idx ON items USING HNSW(embedding);
```

and re-run `EXPLAIN`, we get this plan instead:

We can see that this plan is drastically simplified, but most importantly, the new `HNSW_INDEX_JOIN` operator replaces the `CROSS_PRODUCT` node that was there before and the estimated cardinality went from 5,000,000 to 50,000! Executing this query now takes about 0.15 seconds. That's an almost 66× speedup!

This optimization was just recently added to the VSS extension, so if you've already installed `vss` for DuckDB v1.1.2, run the following command to get the latest version:

```
UPDATE EXTENSIONS (vss);
```

Conclusion

That's all for this time folks! We hope you've enjoyed this update on the DuckDB Vector Similarity Search extension. While this update has focused a lot on new features and improvements such as faster indexing, additional distance functions and more optimizer rules, we're still working on improving some of the limitations mentioned in the previous blog post. We hope to have more to share related to custom indexes and index-based optimizations soon! If you have any questions or feedback, feel free to reach out to us on the `duckdb-vss` GitHub repository or on the DuckDB Discord. Hope to see you around!

Fast Top N Aggregation and Filtering with DuckDB

Publication date: 2024-10-25

Author: Alex Monahan

TL;DR: Find the top N values or filter to the latest N rows more quickly and easily with the N parameter in the `min`, `max`, `min_by`, and `max_by` aggregate functions.

Introduction to Top N

A common pattern when analyzing data is to look for the rows of data that are the highest or lowest in a particular metric. When interested in the highest or lowest N rows in an entire dataset, SQL's standard `ORDER BY` and `LIMIT` clauses will sort by the metric of interest and only return N rows. For example, using the scale factor 1 (SF1) data set of the TPC-H benchmark:

```
INSTALL tpch;
LOAD tpch;
-- Generate an example TPC-H dataset
CALL dbgen(sf = 1);

-- Return the most recent 3 rows by l_shipdate
FROM lineitem
ORDER BY
    l_shipdate DESC
LIMIT 3;
```

l_orderkey	l_partkey	...	l_shipmode	l_comment
354528	6116	...	MAIL	wake according to the u
413956	16402	...	SHIP	usual patterns. carefull
484581	10970	...	TRUCK	ccounts maintain. dogged accounts a

This is useful to quickly get the oldest or newest values in a dataset or to find outliers in a particular metric.

Another common approach is to query the min/max summary statistics of one or more columns. This can find outliers, but the row that contains the outlier can be different for each column, so it is answering a different question. DuckDB's helpful `COLUMNS` expression allows us to calculate the maximum value for all columns.

```
FROM lineitem
SELECT
    max(COLUMNS(*));
```

> The queries in this post make extensive use of DuckDB's FROM-first syntax. This allows the `FROM` and `SELECT` clauses to be swapped, and it even allows omitting the latter entirely.

l_orderkey	l_partkey	...	l_shipmode	l_comment
600000	20000	...	TRUCK	zzle. slyly

However, these two approaches can only answer certain kinds of questions. There are many scenarios where the goal is to understand the top N values *within a group*. In the first example above, how would we calculate the last 10 shipments from each supplier? SQL's `LIMIT` clause is not able to handle that situation. Let's call this type of analysis the top N by group.

This type of analysis is a common tool for exploring new datasets. Use cases include pulling the most recent few rows for each group or finding the most extreme few values in a group. Sticking with our shipment example, we could look at the last 10 shipments of each part number, or find the 5 highest priced orders per customer.

Traditional Top N by Group

In most databases, the way to filter to the top N within a group is to use a window function and a common table expression (CTE). This approach also works in DuckDB. For example, this query returns the 3 most recent shipments for each supplier:

```
WITH ranked_lineitem AS (
    FROM lineitem
    SELECT
        *,
        row_number() OVER
            (PARTITION BY l_suppkey ORDER BY l_shipdate DESC)
            AS my_ranking
)
FROM ranked_lineitem
WHERE
    my_ranking <= 3;
```

l_orderkey	l_partkey	l_suppkey	...	l_shipmode	l_comment	my_ranking
1310688	169532	7081	...	RAIL	ully final exc	1
910561	194561	7081	...	SHIP	ly bold excuses caj	2
4406883	179529	7081	...	RAIL	tions. furious	3
4792742	52095	7106	...	RAIL	onic, ironic courts. final deposits sleep	1
4010212	122081	7106	...	MAIL	accounts cajole finally ironic instruc	2
1220871	94596	7106	...	TRUCK	regular requests above t	3
...	...	...	...	...	...	...

In DuckDB, this can be simplified using the `QUALIFY` clause. `QUALIFY` acts like a `WHERE` clause, but specifically operates on the results of window functions. By making this adjustment, the CTE can be avoided while returning the same results.

```
FROM lineitem
SELECT
    *,
    row_number() OVER
        (PARTITION BY l_suppkey ORDER BY l_shipdate DESC)
        AS my_ranking
QUALIFY
    my_ranking <= 3;
```

This is certainly a viable approach! However, what are its weaknesses? Even though the query is interested in only the 3 most recent shipments, it must sort every shipment just to retrieve those top 3. Sorting in DuckDB has a complexity of $O(kn)$ due to DuckDB's innovative Radix sort implementation, but this is still higher than the $O(n)$ of DuckDB's hash aggregate, for example. Sorting is also a memory intensive operation when compared with aggregation.

Top N in DuckDB

DuckDB 1.1 added a new capability to dramatically simplify and improve performance of top N calculations. Namely, the functions `min`, `max`, `min_by`, and `max_by` all now accept an optional parameter N. If N is greater than 1 (the default), they will return an array of the top values.

As a simple example, let's query the most recent (top 3) shipment dates:

```
FROM lineitem
SELECT
    max(l_shipdate, 3) AS top_3_shipdates;
```

top_3_shipdates
[1998-12-01, 1998-12-01, 1998-12-01]

Top N by Column in DuckDB

The top N selection can become even more useful thanks to the `COLUMNS` expression once again – we can retrieve the 3 top values in each column. We can call this a *top N by column analysis.* It is particularly messy to try to do this analysis with ordinary SQL! You would need a subquery or window function for every single column... In DuckDB, simply:

```
FROM lineitem
SELECT
    max(COLUMNS(*), 3) AS "top_3_\0";
```

top_3_l_orderkey	top_3_l_partkey	...	top_3_l_shipmode	top_3_l_comment
[600000, 600000, 599975]	[20000, 20000, 20000]	...	[TRUCK, TRUCK, TRUCK]	[zzle. slyly, zzle. quickly bold a, zzle. pinto beans boost slyly slyly fin]

Top N by Group in DuckDB

Armed with the new N parameter, how can we speed up a top N by group analysis?

Want to cut to the chase and see the final output? Feel free to skip ahead!

We will take advantage of three other DuckDB SQL features to make this possible:

- The `max_by` function (also known as `arg_max`)
- The `unnest` function
- Automatically packing an entire row into a STRUCT column

The `max` function will return the max (or now the max N!) of a specific column. In contrast, the `max_by` function will find the maximum value in a column, and then retrieve a value from the same row, but a different column. For example, this query will return the ids of the 3 most recently shipped orders for each supplier:

```
FROM lineitem
SELECT
    l_suppkey,
    max_by(l_orderkey, l_shipdate, 3) AS recent_orders
GROUP BY
    l_suppkey;
```

l_suppkey	recent_orders
2992	[233573, 3597639, 3060227]
8516	[4675968, 5431174, 4626530]
3205	[3844610, 4396966, 3405255]
2152	[1672000, 4209601, 3831138]
1880	[4852999, 2863747, 1650084]
...	...

The `max_by` function is an aggregate function, so it takes advantage of DuckDB's fast hash aggregation rather than sorting. Instead of sorting by `l_shipdate`, the `max_by` function scans through the dataset just once and keeps track of the N highest `l_shipdate` values. It then returns the order id that corresponds with each of the most recent shipment dates. The radix sort in DuckDB must scan through the dataset once per byte, so scanning only once provides a significant speedup. For example, if sorting by a 64-bit integer, the sort algorithm must loop through the dataset 8 times vs. 1 with this approach! A simple micro-benchmark is included in the Performance Comparisons section.

However, this SQL query has a few gaps. The query returns results as a `LIST` rather than as separate rows. Thankfully the `unnest` function can split a `LIST` into separate rows:

```
FROM lineitem
SELECT
    l_suppkey,
    unnest(
        max_by(l_orderkey, l_shipdate, 3)
    ) AS recent_orders
GROUP BY
    l_suppkey;
```

l_suppkey	recent_orders
2576	930468
2576	2248354
2576	3640711
5559	4022148
5559	1675680
5559	4976259
...	...

The next gap is that there is no way to easily see the `l_shipdate` associated with the returned `l_orderkey` values. This query only returns a single column, while typically a top N by group analysis will require the entire row.

Fortunately, DuckDB allows us to refer to the entire contents of a row as if it were just a single column! By referring to the name of the table itself (here, `lineitem`) instead of the name of a column, the `max_by` function can retrieve all columns.

```
FROM lineitem
SELECT
    l_suppkey,
    unnest(
        max_by(lineitem, l_shipdate, 3)
    ) AS recent_orders
GROUP BY
    l_suppkey;
```

l_suppkey	recent_orders
5411	{'l_orderkey': 2543618, 'l_partkey': 105410, 'l_suppkey': 5411, ...
5411	{'l_orderkey': 580547, 'l_partkey': 130384, 'l_suppkey': 5411, ...
5411	{'l_orderkey': 3908642, 'l_partkey': 132897, 'l_suppkey': 5411, ...
90	{'l_orderkey': 4529697, 'l_partkey': 122553, 'l_suppkey': 90, ...
90	{'l_orderkey': 4473346, 'l_partkey': 160089, 'l_suppkey': 90, ...
...	...

Let's make that a bit friendlier looking by splitting the STRUCT out into separate columns to match our original dataset.

The Final Top N by Group Query

Passing in one more argument to UNNEST will split this out into separate columns by running recursively. In this case, that means that UNNEST will run twice: once to convert each LIST into separate rows, and then again to convert each STRUCT into separate columns. The `l_suppkey` column can also be excluded, since it will automatically be included already.

```
FROM lineitem
SELECT
    unnest(
        max_by(lineitem, l_shipdate, 3),
        recursive := 1
    ) AS recent_orders
GROUP BY
    l_suppkey;
```

l_orderkey	l_partkey	l_suppkey	...	l_shipinstruct	l_shipmode	l_comment
1234726	6875	6876	...	COLLECT COD	FOB	cajole carefully slyly fin
2584193	51865	6876	...	TAKE BACK RETURN	TRUCK	fully regular deposits at the q
2375524	26875	6876	...	DELIVER IN PERSON	AIR	nusual ideas. busily bold deposi
5751559	95626	8136	...	NONE	SHIP	ers nag fluffily against the spe
3103457	103115	8136	...	TAKE BACK RETURN	FOB	y slyly express warthogs-- unusual, e
5759105	178135	8136	...	COLLECT COD	TRUCK	es. regular pinto beans haggle.
...	...	...	...	...	...	...

> This approach can also be useful for the common task of de-duplicating by finding the latest value within a group. One pattern is to find the current state of a dataset by returning the most recent event in an events table. Simply use an N of 1!

We now have a way to use an aggregate function to calculate the top N rows per group! So, how much more efficient is it?

Performance Comparisons

We will compare the `QUALIFY` approach with the `max_by` approach for solving the top N by group problem. We have discussed both queries, but for reference they are repeated below.

While the main query is running, we will also kick off a background thread to periodically measure DuckDB's memory use. This uses the built in table function `duckdb_memory()` and includes information about Memory usage as well as temporary disk usage. The small

Python script used for benchmarking is included below the results. The machine used for benchmarking was an M1 MacBook Pro with 16 GB RAM.

SF	max_memory	Metric	QUALIFY	max_by	Improvement
1	Default	Total time	0.58 s	0.24 s	2.4×
5	Default	Total time	6.15 s	1.26 s	4.9×
10	36 GB	Total time	36.8 s	25.4 s	1.4×
1	Default	Memory usage	1.7 GB	0.2 GB	8.5×
5	Default	Memory usage	7.9 GB	1.5 GB	5.3×
10	36 GB	Memory usage	15.7 GB	17.1 GB	0.9×

We can see that in each of these situations, the `max_by` approach is faster, in some cases nearly 5× faster! However, as the data grows larger, the `max_by` approach begins to weaken relative to `QUALIFY`.

In some cases, the memory use is significantly lower with `max_by` also. However, the memory use of the `max_by` approach becomes more significant as scale increases, because the number of distinct `l_suppkey` values increases linearly with the scale factor. This increased memory use likely explains the performance decrease, as both algorithms approached the maximum amount of RAM on my machine and began to swap to disk.

In order to reduce the memory pressure, let's re-run the scale factor 10 (SF10) benchmark using fewer threads (4 threads and 1 thread). We continue to use a `max_memory` setting of 36 GB. The prior SF10 results with all 10 threads are included for reference.

SF	Threads	Metric	QUALIFY	max_by	Improvement
10	10	Total time	36.8 s	25.4 s	1.4×
10	4	Total time	49.0 s	21.0 s	2.3×
10	1	Total time	115.7 s	12.7 s	9.1×
10	10	Memory usage	15.7 GB	17.1 GB	0.9×
10	4	Memory usage	15.9 GB	17.3 GB	0.9×
10	1	Memory usage	14.5 GB	1.8 GB	8.1×

The `max_by` approach is so computationally efficient that even with 1 thread it is dramatically faster than the `QUALIFY` approach that uses all 10 threads! Reducing the thread count very effectively lowered the memory use as well (a nearly 10× reduction).

So, when should we use each? As with all database things, *it depends!* If memory is constrained, `max_by` may also offer benefits, especially when the thread count is tuned to avoid spilling to disk. However, if there are approximately as many groups as there are rows, consider `QUALIFY` since we lose some of the memory efficiency of the `max_by` approach.

Conclusion

DuckDB now offers a convenient way to calculate the top N values of both `min` and `max` aggregate functions, as well as their advanced cousins `min_by` and `max_by`. They are easy to get started with, and also enable more complex analyses like calculating the top N for all columns or the top N by group. There are also possible performance benefits when compared with a window function approach.

We would love to hear about the creative ways you are able to use this new feature!

Happy analyzing!

Analytics-Optimized Concurrent Transactions

Publication date: 2024-10-30

Authors: Mark Raasveldt, Hannes Mühleisen

TL;DR: DuckDB employs unique analytics-optimized optimistic multi-version concurrency control techniques. These allow DuckDB to perform large-scale in-place updates efficiently.

> This is the second post on DuckDB's ACID support. If you have not read the first post, Changing Data with Confidence and ACID, it may be a good idea to start there.

In our previous post, we have discussed why changes to data are much saner if the formal "ACID" transaction properties hold. A data system should not allow importing "half" a CSV file into a table because of some unexpected string in line 431,741.

Ensuring the ACID properties of transactions under concurrency is very challenging and one of the "holy grails" of databases. DuckDB implements advanced methods for concurrency control and logging. In this post, we describe DuckDB's Multi-Version Concurrency (MVCC) and Write-Ahead-Logging (WAL) schemes that are specifically designed for efficiently ensuring the transactional guarantees for analytical use cases under concurrent workloads.

Concurrency Control

Pessimistic Concurrency Control. Traditional database systems use locks to manage concurrency. A transaction obtains locks in order to ensure that (a) no other transaction can see its uncommitted changes, and (b) it does not see uncommitted changes of other transactions. Locks need to be obtained both when **reading** (shared locks) and when **writing** (exclusive locks). When a different transaction tries to read data that has been written to by another transaction – it must wait for the other transaction to complete and release its exclusive lock on the data. This type of concurrency control is called **pessimistic**, because locks are always obtained, even if there are no conflicts between transactions.

This strategy works well for transactional workloads. These workloads consist of small transactions that read or modify a few rows. A typical transaction only locks a few rows, and keeps those rows locked only for a short period of time. For analytical workloads, on the other hand, this strategy does not work well. These workloads consist of large transactions that read or modify large parts of the table. An analytical transaction executed in a system that uses pessimistic concurrency control will therefore lock many rows, and keep those rows locked for a long period of time, preventing other transactions from executing.

Optimistic Concurrency Control. DuckDB uses a different approach to manage concurrency conflicts. Transactions do not hold locks – they can always read and write to any row in any table. When a conflict occurs and multiple transactions try to write to the same row at the same time – one of the conflicting transactions is instead aborted. The aborted transaction can then be retried if desired. This type of concurrency control is called **optimistic**.

In case there are never any concurrency conflicts – this strategy is very efficient as we have not unnecessarily slowed down transactions by pessimistically grabbing locks. This strategy works well for analytical workloads – as read-only transactions can never conflict with one another, and multiple writers that modify the same rows are rare in these workloads.

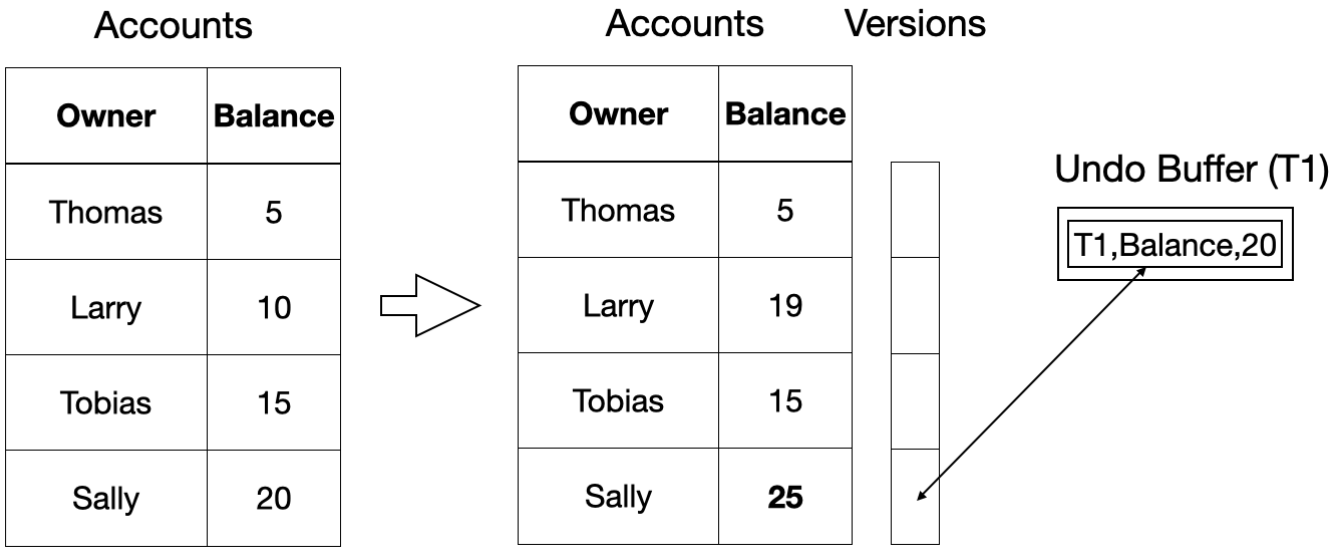
Multi-Version Concurrency Control

In an optimistic concurrency control model – multiple transactions can read and make changes to the same tables at the same time. We have to ensure these transactions cannot see each others' *half-done* changes in order to maintain ACID isolation. A well-known technique to achieve this is Multi-Version Concurrency Control (MVCC). MVCC works by keeping **multiple versions** of modified rows. When a transaction modifies a row – we can create a copy of that row and modify that instead. This allows other transactions to keep on reading the original

version of the row. This allows for each transaction to see their own, consistent state of the database. Often that state is the "version" that existed when the transaction was started. MVCC is widely used in database systems, for example PostgreSQL also uses MVCC.

DuckDB implements MVCC using a technique inspired by the paper "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems" by the one and only Thomas Neumann. This MVCC implementation works by maintaining a list of previous versions to each row in a table. Transactions will update the table data in-place, but will save the previous version of the updated row in the undo buffers. Below is an illustrated example.

```
-- add 5 to Sally's balance
UPDATE Accounts SET Balance = Balance + 5 WHERE Name = 'Sally';
```



When reading a row, a transaction will first check if there is version information for that row. If there is none, which is the common case, the transaction can read the original data. If there is version information, the transaction has to compare the transaction number at the transaction's start time with those in the undo buffers and pick the right version to read.

Efficient MVCC for Analytics

The above approach works well for transactional workloads where individual rows are changed frequently. For *analytical* use cases, we observe a very different usage pattern: changes are much more "bulky" and they often only affect a subset of columns. For example, we do not usually delete individual rows but instead delete all rows matching a pattern, e.g.:

```
DELETE FROM orders WHERE order_time < DATE '2010-01-01';
```

We also commonly bulk update columns, e.g., to fix the evergreen annoyance of people using nonsensical in-domain values to express `NULL`:

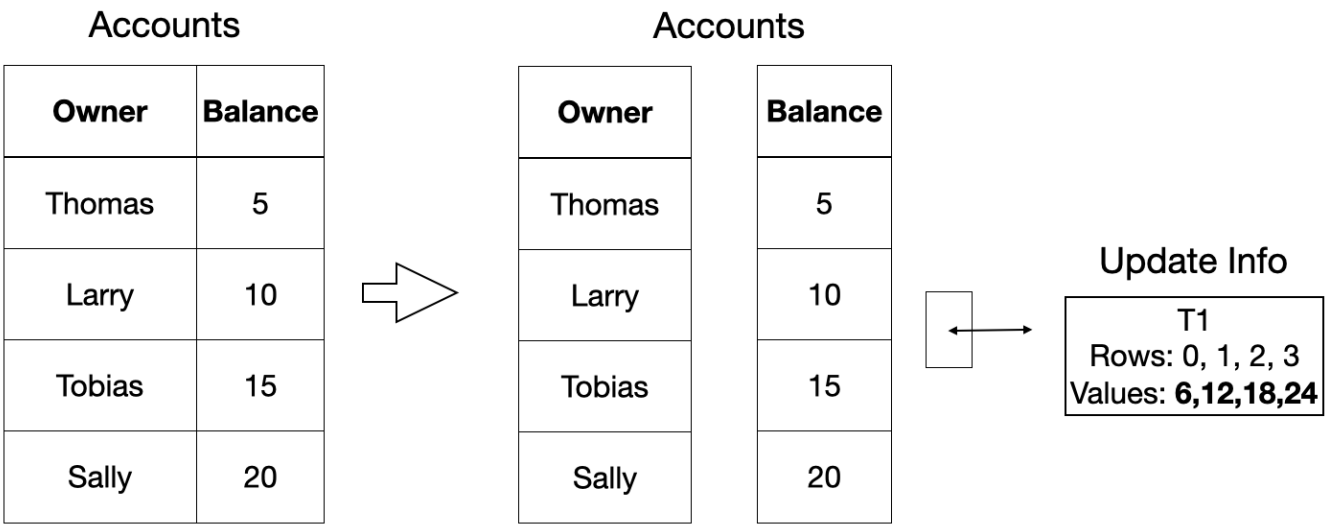
```
UPDATE people SET age = NULL WHERE age = -99;
```

If every row has version information, such bulk changes create a *huge* amount of entries in the undo buffers, which consume a lot of memory and are inefficient to operate on and read from.

There is also an added complication – the original approach relies on performing *in-place updates*. While we can efficiently perform in-place updates on uncompressed data, this is not possible when data is compressed. As DuckDB keeps data compressed, both on-disk and in-memory, in-place updates cannot be performed.

In order to address these issues – DuckDB instead stores **bulk version information** on a per-column basis. For every batch of `2048` rows, a single version information entry is stored. The version information stores the changes made to the data, instead of the old data, as we cannot modify the original data in-place. Instead, any changes made to the data are flushed to disk during a checkpoint. Below is an illustrated example.

```
-- add 20% interest to all accounts
UPDATE Accounts SET Balance = Balance + Balance / 5;
```



One beautiful aspect of this undo buffer scheme is that it is largely performance-transparent: if no changes are made, there are no extra computational cost associated with providing support for transactions. To the best of our knowledge, DuckDB is the *only transactional data management system that is optimized for bulk changes to data* that are common in analytical use cases. But even with changes present, our transaction scheme is very fast for the kind of transactions that we expect for analytical use cases.

Benchmarks

Here is a small experiment, comparing DuckDB 1.1.0, HyPer 9.1.0, SQLite 3.43.2, and PostgreSQL 14.13 on a recent MacBook Pro, showing some of the effects that an OLAP-optimized transaction scheme will have. We should note that HyPer implements the MVCC scheme from the Neumann paper mentioned above. SQLite does not actually implement MVCC, it is mostly included as a comparison point.

We create two tables with either 1 or 100 columns, each with 10 million rows, containing the integer values 1-100 repeating.

```
CREATE TABLE mvcc_test_1 (i INTEGER);
INSERT INTO mvcc_test_1
    SELECT s1
    FROM
        generate_series(1, 100) s1(s1),
        generate_series(1, 100_000) s2(s2);

CREATE TABLE mvcc_test_100 (i INTEGER,
    j1 INTEGER, j2 INTEGER, ..., j99 INTEGER);
INSERT INTO mvcc_test_100
    SELECT s1, s1, s1, ..., s1
    FROM
        generate_series(1, 100) s1(s1),
        generate_series(1, 100_000) s2(s2);
```

We then run three transactions on both tables that increment a single column, with an increasing number of affected rows, 1%, 10% and 100%:

```
UPDATE mvcc_test_... SET i = i + 1 WHERE i <= 1;
UPDATE mvcc_test_... SET i = i + 1 WHERE i <= 10;
UPDATE mvcc_test_... SET i = i + 1 WHERE i <= 100;
```

For the **single-column case**, there should not be huge differences between using a row-major or a column-major concurrency control scheme, and indeed the results show this:

1 Column	1%	10%	100%
DuckDB	0.02	0.07	0.43
SQLite	0.21	0.25	0.61
HyPer	0.66	0.28	2.37
PostgreSQL	1.44	2.48	19.07

Changing more rows took more time. The rows are small, each row only contain a single value. DuckDB and HyPer, having more modern MVCC scheme based on undo buffers as outlined above, are generally much faster than PostgreSQL. SQLite is doing well, but of course it does not have any MVCC. Timings increase roughly 10× as the amount of rows changed is increased tenfold. So far so good.

For the **100 column case**, results look drastically different:

100 Columns	1%	10%	100%
DuckDB	0.02	0.07	0.43
SQLite	0.51	1.79	12.93
HyPer	0.66	6.06	61.54
PostgreSQL	1.42	5.45	50.05

Recall that here we are changing a single column out of 100, a common use case in wrangling analytical data sets. Because DuckDB's MVCC scheme is *designed* for those use cases, it shows exactly the same runtime as in the single-column experiment above. In SQLite, there is a clear impact of the larger row size on the time taken to complete the updates even without MVCC. HyPer and PostgreSQL also show much larger, up to 100× (!) slowdowns as the amount of changed rows is increased.

This neatly brings us to checkpointing.

Write-Ahead Logging and Checkpointing

Any data that's not written to disk but instead still lingers in CPU caches or main memory will be lost in case the operating system crashes or if power is lost. To guarantee durability of changes in the presence of those adverse events, DuckDB needs to *ensure that any committed changes are written to persistent storage*. However, changes in a transaction can be scattered all over potentially large tables, and fully writing them to disk can be quite slow, especially if it has to happen before any transaction can commit. Also, we don't yet know if we actually want to persist a change, we may encounter a failure in the very process of committing.

The traditional approach of transactional data management systems to balance the requirement of writing changes to persistent storage with the requirement of not taking forever is the write-ahead log (WAL). The WAL can be thought of as a log file of all changes to the database. On each transaction commit, its changes are written to the WAL. On restart, the database files are re-loaded from disk, the changes in the WAL are re-applied (if present), and things happily continue. After some amount of changes, the changes in the WAL need to be physically applied to the table, a process known as “checkpointing”. Afterward, the WAL entries can be discarded, a process known as “truncating”. This scheme ensures that changes persist even if a crash occurs or power is lost immediately after a commit.

DuckDB implements write-ahead logging and you may have seen a `.wal` file appearing here and there. Checkpointing normally happens *automatically* whenever the WAL file reached a limit, by default 16 MB but this can be adjusted with the `checkpoint_threshold` setting. Checkpoints also automatically happen at database shutdown. Checkpoints can also be explicitly triggered with the `CHECKPOINT` and `FORCE CHECKPOINT` commands, the difference being that the latter will abort (rollback) any active transactions to ensure the checkpoint is happening *right now* while the former will wait.

DuckDB explicitly calls the `fsync()` system call to make sure any WAL entries will be forced to be written to persistent storage, ignoring the many caches on the way. This is *necessary* because those caches may also be lost in the event of, e.g., power failure, so it's no use to only write log entries to the WAL if they end up not being actually written to storage because the operating system or the disk decided that

it was better to wait for performance reasons. However, `fsync()` does take some time, and while it's generally considered bad practice, there are systems out there that don't do this at all or not by default in order to boast about more transactions per second.

In DuckDB, even bulk loads such as loading large files into tables (e.g., using the COPY statement) are fully transactional. This means you can do something like this:

```
BEGIN TRANSACTION;
CREATE TABLE people (age INTEGER, ...);
COPY people FROM 'many_people.csv';
UPDATE people SET age = NULL WHERE age = -99;
SELECT
    CASE
        WHEN (SELECT count(*) FROM people) = 1_000_000 THEN true
        ELSE error('expected 1m rows')
    END;
COMMIT;
```

This transaction creates a table, copies a large CSV file into the table, and then updates the table to replace a magic value. Finally, a check is performed to see if there is the expected number of rows in the table. All this is bound together into a *single transaction*. If anything goes wrong at any point in the process or the check fails, the transaction will be aborted and zero changes to the database will have happened, the table will not even exist. This is great because it allows implementing all-or-nothing semantics for complex loading tasks, possibly into many tables.

However, logging large changes is a problem. Imagine the `many_people.csv` file being large, say ten gigabytes. As discussed, all changes are written to the WAL and eventually checkpointed. The changes in the file are large enough to immediately trigger a checkpoint. So now we're first writing ten gigabytes to the WAL, and then reading them again, and then writing them again to the database file. Instead of reading ten and writing ten, we have read twenty and written twenty. This is not ideal, but rather than allowing to bypass transactions for bulk loads, DuckDB will instead *optimistically write large changes to new blocks in the database file directly*, and merely add a reference to the WAL. On commit, these new blocks are added to the table. On rollback, the blocks are marked as free space. So while this can lead to the database file pointlessly increasing in size if transactions are aborted, the common case will benefit greatly. Again, this means that users experience near-zero-cost transactionality.

More Experiments

Making concurrency control and write-ahead looking work correctly in the face of failure is very challenging. Software engineers are biased towards the “happy path”, where everything works as intended. The well-known TPC-H benchmark actually contains tests that stress concurrency and logging schemes (Section 3.5.4, “Durability Tests”). Our previous blog post also implemented this test and DuckDB passed.

In addition, we also defined our own, even more challenging test for durability: we run the TPC-H refresh sets one-by-one, in a sub-process. The sub-process reports the last committed refresh. As they are run, after a random (short) time interval, that sub-process is being killed (using `SIGKILL`). Then, DuckDB is restarted, it will likely start recovering from WAL and then continue with the refresh sets. Because of the random time interval, it is likely that DuckDB also gets killed during WAL recovery. This of course should not have any impact on the contents of the database. Finally, we have pre-computed the correct result after running 4000 refresh sets using DuckDB, and after all is set and done we check if there are any differences. There were none, luckily.

To stress our implementation further, we have repeated this experiment on a special file system, LazyFS. This FUSE file system is specifically designed to help uncover bugs in database systems by – among other things – not properly flushing changes to disk using `fsync()`. In our LazyFS configuration, any change that is written to a file is discarded *unless* sync-ed, which also happens if a file is closed. So in our experiment where we kill the database any un-sync-ed entries in the WAL would be lost. We've re-run our durability tests described above on LazyFS and are also **happy to report that no issues were found**.

Conclusion

In this post, we described DuckDB's approaches to concurrency control and write-ahead logging. Of course, we are constantly working on improving them. One nasty failure mode that can appear in real-world systems are partial (“torn”) writes to files, where only parts of write requests actually make it to the file. Luckily, LazyFS can be configured to be even more hostile, for example failing read and write system

calls entirely, returning partial or wrong data, or only partially writing the data to disk. We plan to expand our experimentation on this, to make sure DuckDB's transaction handling is as bullet-proof as it can be.

And who knows, maybe we even dare to unleash the famous Kyle of Jepsen on DuckDB at some point.

Optimizers: The Low-Key MVP

Publication date: 2024-11-14

Author: Tom Ebergen

TL;DR: The query optimizer is an important part of any analytical database system as it provides considerable performance improvements compared to hand-optimized queries, even as the state of your data changes.

Optimizers don't often give “main character” energy in the database community. Databases are usually popular because of their performance, ease of integration, or reliability. As someone who mostly works on the optimizer in DuckDB, I have been wanting to write a blog post about how important optimizers are and why they merit more recognition. In this blog post we will analyze queries that fall into one of three categories: unoptimized, hand-optimized, and optimized by the DuckDB query optimizer. I will also explain why built-in optimizers are almost always better than any hand optimizations. Hopefully, by the end of this blog post, you will agree that optimizers play a silent, but vital role when using a database. Let's first start by understanding where in the execution pipeline query optimization happens.

Before any data is read from the database, the given SQL text must be parsed and validated. If this process finishes successfully, a tree-based query plan is created. The query plan produced by the parser is naïve, and can be extremely inefficient depending on the query. This is where the optimizer comes in, the inefficient query plan is passed to the optimizer for modification and, you guessed it, optimization. The optimizer is made up of many optimization rules. Each rule has the ability to reorder, insert, and delete query operations to create a slightly more efficient query plan that is also logically equivalent. Once all the optimization rules are applied, the optimized plan can be much more efficient than the plan produced by the parser.

> In practice an optimization rule can also be called an optimizer. For the rest of this blog post, optimizer rule will be used for a specific optimization, and optimizer will refer to the database optimizer, unless the word optimizer names a specific optimization rule, (i.e., *Join Order Optimizer*).

Normal Queries vs. Optimized Queries

To examine the effect of the DuckDB query optimizer, let's use a subset of the NYC taxi dataset. You can create native DuckDB tables with the following commands (note that `taxi-data-2019.parquet` is approximately 1.3 GB):

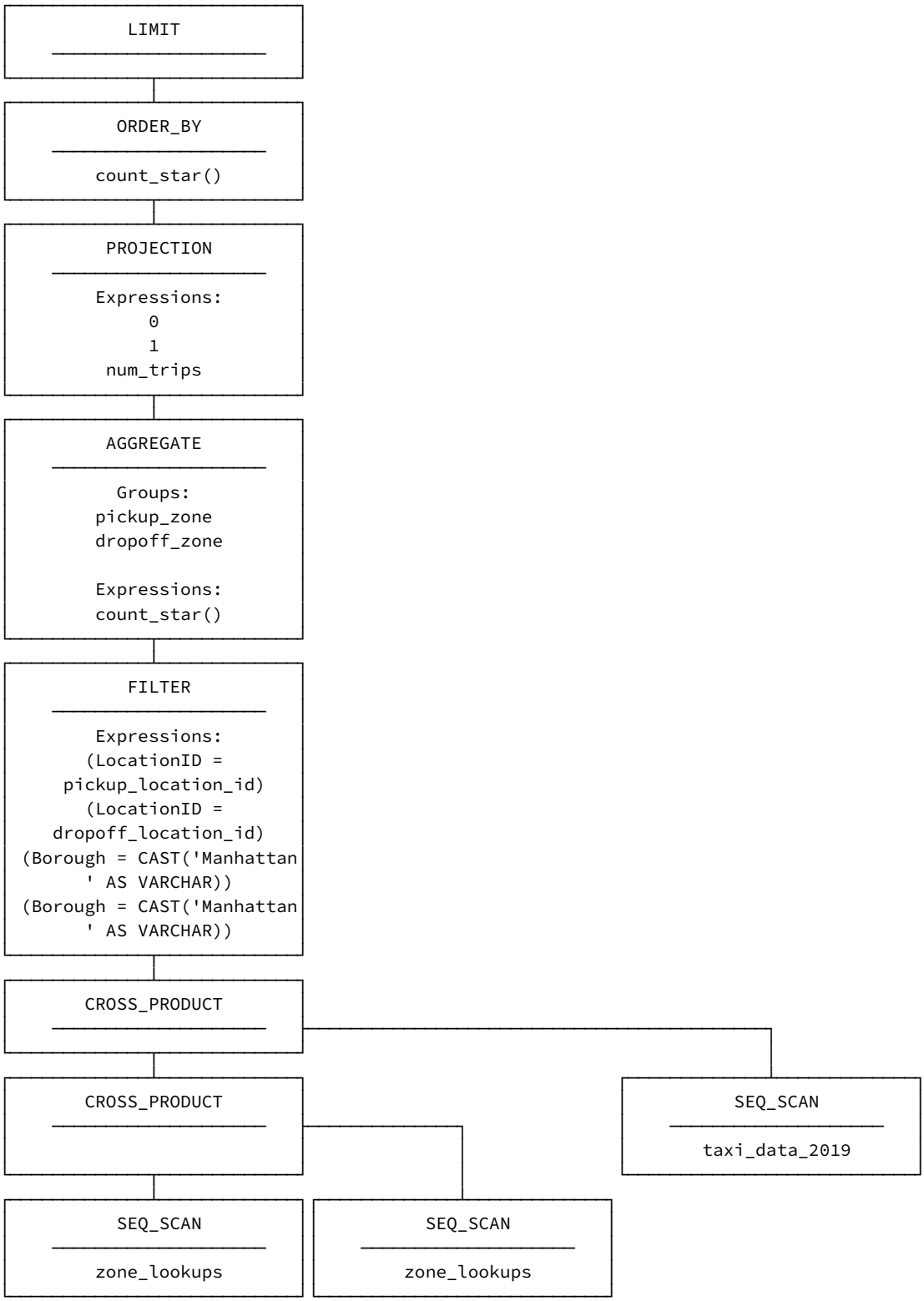
```
CREATE TABLE taxi_data_2019 AS
    FROM 'https://blobs.duckdb.org/data/taxi-data-2019.parquet';
CREATE TABLE zone_lookups AS
    FROM 'https://blobs.duckdb.org/data/zone-lookups.parquet';
```

Now that we have all 2019 data, let's look at the unoptimized vs. optimized plans for a simple query. The following SQL query gets us the most common pickup and drop-off pairs in the Manhattan borough.

```
PRAGMA disable_optimizer;
PRAGMA explain_output = 'optimized_only';
EXPLAIN SELECT
    pickup.zone AS pickup_zone,
    dropoff.zone AS dropoff_zone,
    count(*) AS num_trips
FROM
    zone_lookups AS pickup,
    zone_lookups AS dropoff,
    taxi_data_2019 AS data
WHERE pickup.LocationID = data.pickup_location_id
  AND dropoff.LocationID = data.dropoff_location_id
  AND pickup.Borough = 'Manhattan'
  AND dropoff.Borough = 'Manhattan'
```

```
GROUP BY pickup_zone, dropoff_zone
ORDER BY num_trips DESC
LIMIT 5;
```

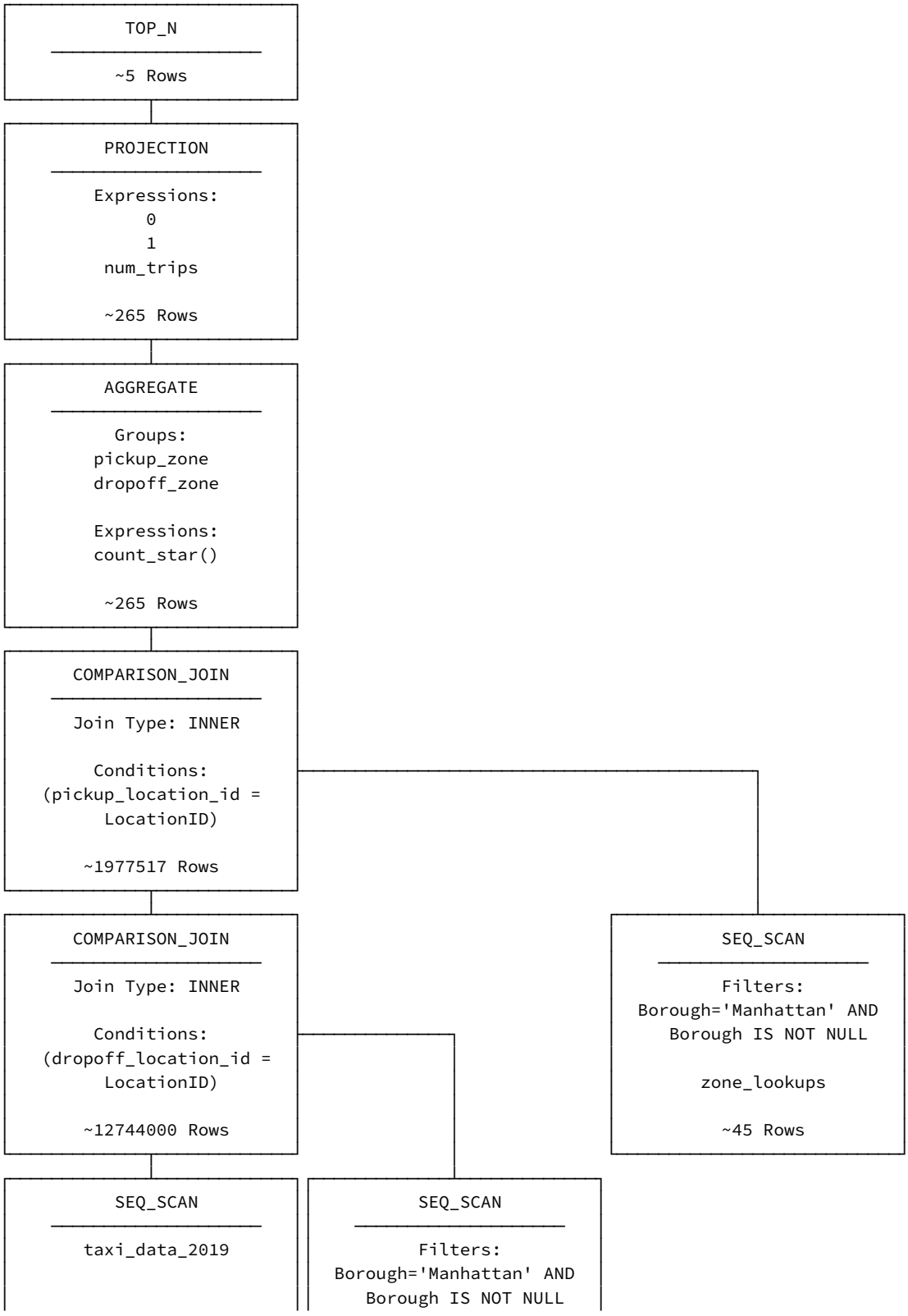
Running this EXPLAIN query gives us the following plan.

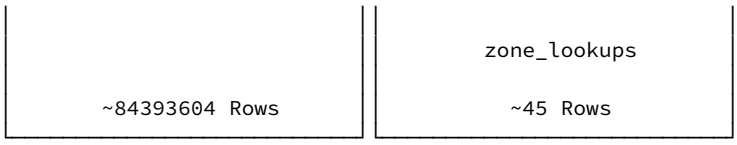


The cross products alone make this query extremely inefficient. The cross-products produce `256 * 256 * |taxi_data_2019|` rows of data, which is 5 trillion rows of data. The filter only matches 71 million rows, which is only 0.001% of the data. The aggregate produces

4,373 rows of data, which need to be sorted by the `ORDER BY` operation, which runs in `O(N * log N)`. Producing 5 trillion tuples alone is an enormous amount of data processing, which becomes clear when you try to run the query and notice it doesn't complete. With the optimizer enabled, the query plan produced is much more efficient because the operations are re-ordered to avoid many trillions of rows of intermediate data. Below is the query plan with the optimizer enabled:

```
PRAGMA enable_optimizer;
EXPLAIN ...
```





Let's first look at the difference in execution times on my MacBook with an M1 Max and 32 GB of memory before talking about the optimizations that have taken place.

	Unoptimized	Optimized
Runtime	>24 hours	0.769 s

Hopefully this performance benefit illustrates how powerful the DuckDB Optimizer is. So what optimization rules are responsible for these drastic performance improvements? For the query above, there are three powerful rules that are applied when optimizing the query: *Filter Pushdown, Join Order Optimization,* and *TopN Optimization*.

The *Filter Pushdown Optimizer* is very useful since it reduces the amount of intermediate data being processed. It is an optimization rule that is sometimes easy to miss for humans and will always result in faster execution times if the filter is selective in any way. It takes a filter, like `Borough = 'Manhattan'` and pushes it down to the operator that first introduces the filtered column, in this case the table scan. In addition, it will also detect when a filtered column like `col1` is used in an equality condition (i.e., `WHERE col1 = col2`). In these cases, the filter is duplicated and applied to the other column, `col2`, further reducing the amount of intermediate data being processed.

The *Join Order Optimizer* recognizes that the filters `pickup.LocationID = data.pickup_location_id` and `dropoff.LocationID = data.dropoff_location_id` can be used as join conditions and rearranges the scans and joins accordingly. This optimizer rule does a lot of heavy lifting to reduce the amount of intermediate data being processed since it is responsible for removing the cross products.

The *TopN Optimizer* is very useful when aggregate data needs to be sorted. If a query has an `ORDER BY` and a `LIMIT` operator, a TopN operator can replace these two operators. The TopN operator orders only the highest/lowest `N` values, instead of all values. If `N` is 5, then DuckDB only needs to keep 5 rows with the minimum/maximum values in memory and can throw away the rest. So if you are only interested in the top `N` values out of `M`, where `N << M`, the TopN operator can run in `O(M + N * log N)` instead of `O(M * log M)`.

These are just a few of the optimizations DuckDB has. More optimizations are explained in the section Summary of All Optimizers.

Hand-Optimized Queries

For the query above, it is possible to achieve almost the same plan by carefully writing the SQL query by hand. To achieve a similar plan as the one generated by DuckDB, you can write the following.

```
SELECT
    pickup.zone AS pickup_zone,
    dropoff.zone AS dropoff_zone,
    count(*) AS num_trips
FROM
    taxi_data_2019 data
INNER JOIN
    (SELECT * FROM zone_lookups WHERE Borough = 'Manhattan') pickup
    ON pickup.LocationID = data.pickup_location_id
INNER JOIN
    (SELECT * FROM zone_lookups WHERE Borough = 'Manhattan') dropoff
    ON dropoff.LocationID = data.dropoff_location_id
GROUP BY pickup_zone, dropoff_zone
ORDER BY num_trips desc
LIMIT 5;
```

Inspecting the runtimes again we get:

	Unoptimized	Hand-optimized	Optimized
Runtime	>24 hours	0.926 s	0.769 s

The SQL above results in a plan similar to the DuckDB optimized plan, but it is wordier and more error-prone to write, which can potentially lead to bugs. In very rare cases, it is possible to hand write a query that produces a more efficient plan than an optimizer. These cases are extreme outliers, and in all other cases the optimizer will produce a better plan. Moreover, a hand-optimized query is optimized for the current state of the data, which can change with many updates over time. Once a sufficient amount of changes are applied to the data, the assumptions of a hand-optimized query may no longer hold, leading to bad performance. Let's take a look at the following example.

Suppose an upstart company has an `orders` and `parts` table and every time some dashboard loads, the most popular ordered parts needs to be calculated. Since the company is still relatively new, they only have a small amount orders, but their catalog of parts is still quite large. A hand-optimized query would look like this:

```
CREATE OR REPLACE TABLE orders AS
    SELECT RANGE order_id, range % 10_000 pid FROM range(1_000);
CREATE TABLE parts AS
    SELECT range p_id, range::VARCHAR AS part_name FROM range(10_000);
SELECT
    parts.p_id,
    parts.part_name,
    count(*) AS ordered_amount
FROM parts
INNER JOIN orders
    ON orders.pid = parts.p_id
GROUP BY ALL;
```

Naturally, the number of orders will increase as this company gains customers and grows in popularity. If the query above continues to run without the use of an optimizer, the performance will slowly decline. This is because the execution engine will build the hash table on the orders table, which potentially will have 100 million rows. If the optimizer is enabled, the Join Order Optimizer will be able to inspect the statistics of the table during the optimization process and produce a new plan according to the new state of the data.

Here is a breakdown of running the queries with and without the optimizer as the orders table increases.

	Unoptimized	Optimized
|orders| = 1K	0.004 s	0.003 s
|orders| = 10K	0.005 s	0.005 s
|orders| = 100K	0.013 s	0.008 s
|orders| = 1M	0.055 s	0.014 s
|orders| = 10M	0.240 s	0.044 s
|orders| = 100M	2.266 s	0.259 s

At first the difference in execution time is not really noticeable, so no one would think a query rewrite would be the solution. But once enough orders are reached, waiting 2 seconds every time the dashboard loads becomes tedious. If the optimizer is enabled, the query performance improves by a factor of 10×. So if you ever think you have identified a scenario where you are smarter than the optimizer, make sure you have also thought about all possible updates to the data and have hand-optimized for those as well.

Optimizations That Are Impossible by Hand

Some optimization rules are also impossible to write by hand. For example, the TopN optimization can not be optimized by hand.

Another good example is the Join Filter Pushdown optimization. The Join Filter Pushdown optimization works in scenarios where the build side of a hash join has a subset of the join keys. In its current state the join filter pushdown optimization keeps track of the minimum value

key and maximum value key and pushes a table filter into the probe side to filter out keys greater than the maximum join value and smaller than the minimum join value.

With a small change, we can use the query from above to demonstrate this. Suppose we first filter our `parts` table to only include parts with a specific prefix in the `part_name`. When the `orders` table has 100 million rows and the `parts` table only has ~20,000 after filtering, then the `orders` table will be the probe side and the `parts` table will be the hash/build side. When the hash table is built, the min and max `p_id` values in the `parts` table are recorded, in this case it could be 20,000 and 80,000. These min and max values get pushed as a filter into the `orders` table scan, filtering out all parts with `p_id > 80,000` and `pid < 20,000`. 40% of the `orders` table has a `pid` greater than 80,000, and less than 20,000 so this optimization does a lot of heavy lifting in join queries.

Imagine trying to express this logic in your favorite data frame API; it would be extremely difficult and error-prone. The library would need to implement this optimization automatically for all hash joins. The Join Filter Pushdown optimization can improve query performance by 10×, so it should be a key factor when deciding what analytical system to use.

If you use a data frame library like collapse, pandas, data.table, modin, then you are most likely not enjoying the benefits of query optimization techniques. This means your optimizations need to be applied by hand, which is not sustainable if your data starts changing. Moreover, you are most likely writing imperatively, using a syntax specific to the dataframe library. This means the scripts responsible for analyzing data are not very portable. SQL, on the other hand, can be much more intuitive to write since it is a declarative language, and can be ported to practically any other database system.

Summary of All Optimizers

Below is a non-exhaustive list of all the optimization rules that DuckDB applies.

Expression Rewriter

The *Expression Rewriter* simplifies expressions within each operator. Sometimes queries are written with expressions that are not completely evaluated or they can be rewritten in a way that takes advantage of features within the execution engine. Below is a table of common expression rewrites and the optimization rules that are responsible for them. Many of these rules rewrite expressions to use specialized DuckDB functions so expression evaluation is much faster during execution. If an expression can be evaluated to `true` in the optimizer phase, there is no need to pass the original expression to the execution engine. In addition, the optimized expressions are more likely to allow DuckDB to make further improvements to the query plan. For example, the "Move constants" rule could enable filter pushdown to occur.

Rewriter rule	Original expression	Optimized expression
Move constants	`x + 1 = 6`	`x = 5`
Constant folding	`2 + 2 = 4`	`true`
Conjunction simplification	`(1 = 2 AND b)`	`false`
Arithmetic simplification	`x * 1`	`x`
Case simplification	`CASE WHEN true THEN x ELSE y END`	`x`
Equal or NULL simplification	`a = b OR (a IS NULL AND b IS NULL)`	`a IS NOT DISTINCT FROM b`
Distributivity	`(x AND b) OR (x AND c) OR (x AND d)`	`x AND (b OR c OR d)`
Like optimization	`regexp_matches(c, '^Prefix')`	`LIKE 'Prefix%'`

Filter Pull-Up & Filter Pushdown

Filter Pushdown was explained briefly above. *Filter Pull-Up* is also important to identify cases where a filter can be applied on columns in other tables. For example, the query below scans column a from both `t1` and `t2`. `t1.a` has a filter, but in the presence of the equality condition, `t2.a` can have the same filter. For example:

```
SELECT *
FROM t1, t2
WHERE t1.a = t2.a
  AND t1.a = 50;
```

This can be optimized to:

```
SELECT *
FROM t1, t2
WHERE t1.a = t2.a
  AND t1.a = 50
  AND t2.a = 50;
```

Filter Pull-Up pulls up the filter `t1.a = 50` above the join, and when the filter is pushed down again, the optimizer rule recognizes the filter can be applied to both columns `t1.a` and `t2.a`.

IN Clause Rewriter

If there is a filter with an `IN` clause, sometimes it can be re-written so execution is more efficient. Some examples are below:

Original	Optimized
`c1 IN (1)`	`c1 = 1`
`c1 IN (3, 4, 5)`	`c1 >= 3 AND c1 <= 5`

In addition, the *IN Clause Rewriter* will transform expensive `IN` expressions into `MARK` joins. If a query has an expression like `c1 IN (x1, ..., xn)` where `n` is quite large, it can be expensive to evaluate this expression for every row in the table. The runtime would be `O(n * m)` where `n` is the number of rows and `m` is the length of the list. The `IN` clause rewriter will transform the expression into `SELECT c1 FROM t1, VALUES (x1, ..., xn) t(c0) WHERE c1 = c0` turning the expression into a `HASH` join that can complete in `O(n + m)` time!

Join Order Optimizer

The *Join Order Optimizer* can provide an enormous performance benefit by limiting the number of intermediate tuples that are processed between joins. By processing fewer intermediate tuples, the query can execute faster.

Statistics Propagation

Statistics Propagation is another optimization that works even when the state of the data changes. By traversing the query plan and keeping note of all equality join conditions, the Statistics Propagation optimizer can create new filters by inspecting the statistics of the columns that are eventually joined. For example, suppose `t1.a` and `t2.a` will be joined with the equality condition `t1.a = t2.a`. If our internal statistics tell us `t1.a` has a maximum value of `50` and a minimum value of `25`, the optimizer can create a new filter when scanning table `t2`. The filter would be `t2.a >= 25 AND t2.a <= 50`.

Reorder Filters

If there are multiple filters on a column, the order in which these filters are executed also becomes important. It's best to execute the most efficient filters first, saving execution of expensive filters for later. For example, DuckDB can evaluate equality very quickly. So for a query like `... WHERE a = 50 AND md5(b) LIKE '%d77%'`, the optimizer will tell DuckDB to evaluate `a = 50` on every column first. If the value in column `a` passes the check `a = 50`, DuckDB will evaluate the `md5` hash for the values in column `b`.

Conclusion

A well-written optimizer can provide significant performance improvements when allowed to optimize freely. Not only can the optimizer apply the many optimization rules a human might naturally miss, an optimizer can respond to changes in the data. Some optimizations can result in a performance improvement of 100×, which might be the difference when deciding to use analytical system *A* vs. analytical system *B*. With DuckDB, all optimization rules are applied automatically to every query, so you can continually enjoy the benefits. Hopefully this blog post has convinced you to consider the optimizer next time you hear about the next database that has everyone's ears burning.

Runtime-Extensible SQL Parsers Using PEG

Publication date: 2024-11-22

Authors: Hannes Mühleisen, Mark Raasveldt

TL;DR: Despite their central role in processing queries, parsers have not received any noticeable attention in the data systems space. State-of-the art systems are content with ancient old parser generators. These generators create monolithic, inflexible and unforgiving parsers that hinder innovation in query languages and frustrate users. Instead, parsers should be rewritten using modern abstractions like Parsing Expression Grammars (PEG), which allow dynamic changes to the accepted query syntax and better error recovery. In this post, we discuss how parsers could be re-designed using PEG, and validate our recommendations using experiments for both effectiveness and efficiency.

> **Update.** In March 2026, DuckDB v1.5 shipped an experimental parser. You can opt-in to use it via:
>
> ```
> CALL enable_peg_parser();
> ```

> This post is a shortened version of our peer-reviewed research paper "Runtime-Extensible Parsers" that was accepted for publication and presentation at the 2025 Conference on Innovative Data Systems Research (CIDR) that is going to be held in Amsterdam between January 19 and 22, 2025. You can read the full paper if you prefer.

The parser is the DBMS component that is responsible for turning a query in string format into an internal representation which is usually tree-shaped. The parser defines which queries are going to be accepted at all. Every single SQL query starts its journey in a parser. Despite its prominent position in the stack, very little research has been published on parsing queries for data management systems. There seems to have been very little movement on the topic in the past decades and their implementations are largely stuck in sixty-year-old abstractions and technologies.

The constant growth of the SQL specification with niche features (e.g., support for graph queries in SQL/PGQ or XML support) as well as the desire to support alternative query notations like dplyr, piped SQL, PRQL or SaneQL makes monolithic parsers less and less practical: in their traditional design, parser construction is a *compile-time* activity where enormous grammar files are translated into state machine transition lookup tables which are then baked in a system binary. Having those *always* be present in the parser might be wasteful especially for size-conscious binary distributions like WebAssembly (Wasm).

Many if not most SQL systems use a static parser created using a YACC-style parser toolkit: we are able to easily confirm this for open-source systems like PostgreSQL and MySQL/MariaDB. From analyzing their binaries' symbol names, we also found indications that Oracle, SQL Server and IBM Db2 use YACC. Internally, YACC and its slightly more recent variant GNU Bison as well as the "Lemon" parser generator used by SQLite all use a "single look-ahead left-to-right rightmost derivation" LALR(1) parser generator. This generator translates a formal context-free set of grammar rules in Extended Backus-Naur Form (EBNF) to a parser state machine. LALR parsers are a more space-efficient specialization of LR(k) parsers as first described by Knuth. But in effect, **the most advanced SQL systems of 2024 use parser technology from the 1960s**. Given that the rest of data management systems have been greatly overhauled since this should raise the question of why the parser did not receive any serious engineering attention.

Database systems are moving towards becoming *ecosystems* instead of pre-built monoliths. Much of the innovation in the PostgreSQL, SQLite, and DuckDB communities now comes from extensions, which are shared libraries that are loaded into the database system at run-time to extend the database system with features like vector similarity search, geospatial support, file systems, or graph processing. Bundling all those features upfront would be difficult due to additional binary size, external dependencies. In addition, they are often maintained independently by their communities. Thus far, at least in part due to the ubiquity of YACC-style parsers, those community extensions have been restricted from extending syntax. While this is also true in other ecosystems like Python, the design of SQL with its heavy focus on syntax and not function calls makes the extensions second-class citizens that have to somehow work around the restrictions by the original parser, e.g., by embedding custom expressions in strings.

We propose to *re-think data management system parser design* to create modern, *extensible* parsers, which allow a dynamic configuration of the accepted syntax *at run-time*, for example to allow syntax extensions, new statements, or to add entirely new query languages. This

would allow to break up the monolithic grammars currently in use and enable more creativity and flexibility in what syntax a data management system can accept, both for industrial and research use. Extensible parsers allow for new grammar features to be easily integrated and tested, and can also help bridge the gap between different SQL dialects by adding support for the dialect of one system to the parser of another. Conversely, it might also be desirable in some use cases to *restrict* the acceptable grammar, e.g., to restrict the complexity of queries, or to enforce strict compliance with the SQL standard.

Modernizing parser infrastructure also has additional benefits: one of the most-reported support issues with data management systems are unhelpful syntax errors. Some systems go to great lengths to try to provide a meaningful error message, e.g., `this column does not exist, did you mean ...`, but this is typically limited to resolving identifiers following the actual parsing. YACC-style parsers exhibit "all-or-nothing" behavior, the *entire* query or set of queries either is accepted entirely or not at all. This is why queries with actual syntactical errors (e.g., `SELEXT` instead of `SELECT` are usually harshly rejected by a DBMS. MySQL for example is notorious for its unhelpful error messages:

```
You have an error in your SQL syntax; check the manual that corresponds
to your MySQL server version for the right syntax to use near 'SELEXT'
at line 1.
```

Parsing Expression Grammar

Parsing Expression Grammar (PEG) parsers represent a more modern approach to parsing. PEG parsers are top-down parsers that effectively generate a recursive-descent style parser from a grammar. Through the "packrat" memoization technique PEG parsers exhibit linear time complexity in parsing at the expense of a grammar-dependent amount of extra memory. The biggest difference from a grammar author perspective is the choice operator where multiple syntax options can be matched. In LALR parsers options with similar syntax can create ambiguity and reduce conflicts. In PEG parsers the *first* matching option is always selected. Because of this, PEG parsers cannot be ambiguous by design.

As their name suggests, parsing expression grammar consists of a set of *parsing expressions*. Expressions can contain references to other rules, or literal token references, both as actual strings or character classes similar to regular expressions. Expressions can be combined through sequences, quantifiers, optionals, groupings and both positive and negative look-ahead. Each expression can either match or not, but it is required to consume a part of the input if it matches. Expressions are able to look ahead and consider the remaining input but are not required to consume it. Lexical analysis is typically part of the PEG parser itself, which removes the need for a separate step.

One big advantage is that PEG parsers *do not require a compilation step* where the grammar is converted to for example a finite state automaton based on lookup tables. PEG can be executed directly on the input with minimal grammar transformation, making it feasible to re-create a parser at runtime. PEG parsers are gaining popularity, for example, the Python programming language has recently switched to a PEG parser.

Another big advantage of PEG parsers is *error handling*: the paper "Syntax Error Recovery in Parsing Expression Grammars" describes a practical technique where parser rules are annotated with "recovery" actions, which can (1) show more than a single error and (2) annotate errors with a more meaningful error message.

A possible disadvantage of memoized packrat parsing is the memory required for memoization: the amount required is *proportional to the input size*, not the stack size. Of course, memory limitations have relaxed significantly since the invention of LALR parsers sixty years ago and queries typically are not "Big Data" themselves.

Proof-of-Concept Experiments

To perform experiments on parser extensibility, we have implemented an – admittedly simplistic – experimental prototype PEG parser for enough of SQL to parse *all* the TPC-H and TPC-DS queries. This grammar is compatible with the `cpp-peglib` single-header C++17 PEG execution engine.

`cpp-peglib` uses a slightly different grammar syntax, where `/` is used to denote choices. The symbol `?` shows an optional element, and `*` defines arbitrary repetition. The special rules `Parens()` and `List()` are grammar macros that simplify the grammar for common elements. The special `%whitespace` rule is used to describe tokenization.

Below is an abridged version of our experimental SQL grammar, with the `Expression` and `Identifier` syntax parsing rules omitted for brevity:

```
Statements <- SingleStmt (';' SingleStmt )* ';'*
SingleStmt <- SelectStmt
SelectStmt <- SimpleSelect (SetopClause SimpleSelect)*

SetopClause <-
    ('UNION' / 'EXCEPT' / 'INTERSECT') 'ALL'?

SimpleSelect <- WithClause? SelectClause FromClause?
    WhereClause? GroupByClause? HavingClause?
    OrderByClause? LimitClause?

WithStatement <- Identifier 'AS' SubqueryReference
WithClause <- 'WITH' List(WithStatement)
SelectClause <- 'SELECT' ('*' / List(AliasExpression))
ColumnsAlias <- Parens(List(Identifier))

TableReference <-
    (SubqueryReference 'AS'? Identifier ColumnsAlias?) /
    (Identifier ('AS'? Identifier)?)

ExplicitJoin <- ('LEFT' / 'FULL')? 'OUTER'?
    'JOIN' TableReference 'ON' Expression

FromClause <- 'FROM' TableReference
    ((',' TableReference) / ExplicitJoin)*

WhereClause <- 'WHERE' Expression
GroupByClause <- 'GROUP' 'BY' List(Expression)
HavingClause <- 'HAVING' Expression
SubqueryReference <- Parens(SelectStmt)

OrderByExpression <- Expression ('DESC' / 'ASC')?
    ('NULLS' 'FIRST' / 'LAST')?

OrderByClause <- 'ORDER' 'BY' List(OrderByExpression)
LimitClause <- 'LIMIT' NumberLiteral
AliasExpression <- Expression ('AS'? Identifier)?
%whitespace <- [ \t\n\r]*
List(D) <- D (',' D)*
Parens(D) <- '(' D ')'
```

All experiments were run on a 2021 MacBook Pro with the M1 Max CPU and 64 GB of RAM. The experimental grammar and the code for experiments are available on GitHub.

Loading the base grammar from its text representation into the `cpp-peglib` grammar dictionary with symbolic rule representations takes 3 ms. In case that delay should become an issue, the library also allows to define rules programmatically instead of as strings. It would be straightforward to pre-compile the grammar file into source code for compilation, YACC-style. While somewhat counter-intuitive, it would reduce the time required to initialize the initial, unmodified parser. This difference matters for some applications of e.g., DuckDB where the database instance only lives for a few short milliseconds.

For the actual parsing, YACC parses TPC-H Query 1 in ca. 0.03 ms, where `cpp-peglib` takes ca. 0.3 ms, a ca. 10 times increase. To further stress parsing performance, we repeated all TPC-H and TPC-DS queries six times to create a 36,840 line SQL script weighing in at ca. 1 MB. Note that a recent study has found that the 99-percentile of read queries in the Amazon Redshift cloud data warehouse are smaller than 16.5 kB.

Postgres takes on average 24 ms to parse this file using YACC. Note that this time includes the execution of grammar actions that create Postgres' parse tree. `cpp-peglib` takes on average 266 ms to parse the test file. However, our experimental parser does not have grammar actions defined yet. When simulating actions by generating default AST actions for every rule, parsing time increases to 339 ms. Note that the AST generation is more expensive than required, because a node is created for each matching rule, even if there is no semantic meaning in the grammar at hand.

Overall, we can observe a ca. 10 times slowdown in parsing performance when using the `cpp-peglib` parser. However, it should be noted that the *absolute duration* of those two processes is still tiny; at least for analytical queries, sub-millisecond parsing time is more

than acceptable as parsing still only accounts for a tiny fraction of overall query processing time. Furthermore, there are still ample optimization opportunities in the experimental parsers we created using an off-the-shelf PEG library. For example, the library makes heavy use of recursive function calls, which can be optimized e.g., by using a loop abstraction.

In the following, we present some experiments in extending the prototype parser with support for new statements, entirely new syntax and with improvements in error messages.

> It is already possible to replace DuckDB's parser by providing an alternative parser. Several community extensions such as `duckpgq`, `prql` and `psql` use this approach. When trying to parse a query string, DuckDB first attempts to use the default parser. If this fails, it switches to the extension parsers as failover. Therefore, these extensions cannot simply extend the parser with a few extra rules – instead, they implement the complete grammar of their target language.

Adding the `UNPIVOT` Statement

Let's assume we would want to add a new top-level `UNPIVOT` statement to turn columns into rows to a SQL dialect. `UNPIVOT` should work on the same level as e.g., `SELECT`, for example to unpivot a table `t1` on a specific list of columns or all columns (`*`), we would like to be able to write:

```
UNPIVOT t1 ON (c1, c2, c3);
UNPIVOT t1 ON (*);
```

It is clear that we would have to somehow modify the parser to allow this new syntax. However, when using a YACC parser, this would require modifying the grammar, re-running the parser generator, hoping for the absence of shift-reduce conflicts, and then recompiling the actual database system. However, this is not practical at run-time which is when extensions are loaded, ideally within milliseconds.

In order to add `UNPIVOT`, we have to define a grammar rule and then modify `SingleStmt` to allow the statement in a global sequence of SQL statements. This is shown below. We define the new `UnpivotStatement` grammar rule by adding it to the dictionary, and we then modify the `SingleStmt` rule entry in the dictionary to also allow the new statement.

```
UnpivotStatement <- 'UNPIVOT' Identifier
    'ON' Parens(List(Identifier) / '*')

SingleStmt <- SelectStatement / UnpivotStatement
```

Note that we re-use other machinery from the grammar like the `Identifier` rule as well as the `Parens()` and `List()` macros to define the `ON` clause. The rest of the grammar dictionary remains unchanged. After modification, the parser can be re-initialized in another 3 ms. Parser execution time was unaffected.

Extending `SELECT` with `GRAPH_TABLE`

Let's now assume we would want to modify the `SELECT` syntax to add support for SQL/PGQ graph matching patterns. Below is an example query in SQL/PGQ that finds the university name and year for all students called Bob:

```
SELECT study.classYear, study.name
FROM GRAPH_TABLE (pg,
    MATCH
        (a:Person WHERE a.firstName = 'Bob')-[s:studyAt]->(u:University)
        COLUMNS (s.classYear, u.name)
) study;
```

We can see that this new syntax adds the `GRAPH_TABLE` clause and the pattern matching domain-specific language (DSL) within. To add support for this syntax to a SQL parser at runtime, we need to modify the grammar for the `SELECT` statement itself. This is fairly straightforward when using a PEG. We replace the rule that describes the `FROM` clause to also accept a sub-grammar starting at the `GRAPH_TABLE` keyword following by parentheses. Because the parser does not need to generate a state machine, we are immediately able to accept the new syntax.

Below we show a small set of grammar rules that are sufficient to extend our experimental parser with support for the SQL/PGQ `GRAPH_TABLE` clause and the containing property graph patterns. With this addition, the parser can parse the query above. Parser construction and parser execution timings were unaffected.

```
Name <- (Identifier? ':' Identifier) / Identifier
Edge <- ('-' / '<-') '[' Name ']' ('->' / '-')

Pattern <- Parens(Name WhereClause?) Edge
    Parens(Name WhereClause?)

PropertyGraphReference <- 'GRAPH_TABLE'i '('
        Identifier ','
        'MATCH'i List(Pattern)
        'COLUMNS'i Parens(List(ColumnReference))
    ')' Identifier?

TableReference <-
    PropertyGraphReference / ...
```

`dplyr`, the "Grammar of Data Manipulation", is the de facto standard data transformation language in the R Environment for Statistical Computing. The language uses function calls and a special chaining operator (`%>%`) to combine operators. Below is an example dplyr query:

```
df %>%
  group_by(species) %>%
  summarise(
    n = n(),
    mass = mean(mass, na.rm = TRUE)
  ) %>%
  filter(n > 1, mass > 50)
```

For those unfamiliar with dplyr, the query is equivalent to this SQL query:

```
SELECT * FROM (
    SELECT count(*) AS n, AVG(mass) AS mass
        FROM df
        GROUP BY species)
    WHERE n > 1 AND mass > 50;
```

With an extensible parser, it is feasible to add support for completely new query languages like `dplyr` to a SQL parser. Below is a simplified grammar snippet that enables our SQL parser to accept the `dplyr` example from above.

```
DplyrStatement <- Identifier Pipe Verb (Pipe Verb)*
Verb <- VerbName Parens(List(Argument))
VerbName <- 'group_by' / 'summarise' / 'filter'
Argument <- Expression / (Identifier '=' Expression)
Pipe <- '%>%'

SingleStmt <- SelectStatement /
    UnpivotStatement / DplyrStatement
```

It is important to note that the rest of the experimental SQL parser *still works*, i.e., the `dplyr` syntax now *also* works. Parser construction and parser execution timings were again unaffected.

Better Error Messages

As mentioned above, PEG parsers are able to generate better error messages elegantly. A common novice SQL user mistake is to mix up the order of keywords in a query, for example, the `ORDER BY` must come after the `GROUP BY`. Assume an inexperienced user types the following query:

```
SELECT customer, SUM(sales)
FROM revenue
ORDER BY customer
GROUP BY customer;
```

By default, both the YACC and the PEG parsers will report a similar error message about an `unexpected 'GROUP' keyword` with a byte position. However, with a PEG parser we can define a "recovery" syntax rule that will create a useful error message. We modify the `OrderByClause` from our experimental grammar like so:

```
OrderByClause <- 'ORDER'i 'BY'i List(OrderByExpression)
    %recover(WrongGroupBy)?
WrongGroupBy <- GroupByClause
    { error_message "GROUP BY must precede ORDER BY" }
```

Here, we use the `%recover` construct to match a misplaced GROUP BY clause, re-using the original definition, and then trigger a custom error message that advises the user on how to fix their query. And indeed, when we parse the wrong SQL example, the parser will output the custom message.

Conclusion and Future Work

In this post, we have proposed to modernize the ancient art of SQL parsing using more modern parser generators like PEG. We have shown how by using PEG, a parser can be extended at run-time at minimal cost without re-compilation. In our experiments we have demonstrated how minor grammar adjustments can fundamentally extend and change the accepted syntax.

An obvious next step is to address the observed performance drawback observed in our prototype. Using more efficient implementation techniques, it should be possible to narrow the gap in parsing performance between YACC-based LALR parsers and a dynamic PEG parser. Another next step is to address some detail questions for implementation: for example, parser extension load order should ideally not influence the final grammar. Furthermore, while parser actions can in principle execute arbitrary code, they may have to be restrictions on return types and input handling.

We plan to switch DuckDB's parser, which started as a fork of the Postgres YACC parser, to a PEG parser in the near future. As an initial step, we have performed an experiment where we found that it is possible to interpret the current Postgres YACC grammar with PEG. This should greatly simplify the transitioning process, since it ensures that the same grammar will be accepted in both parsing frameworks.

Acknowledgments

We would like to thank **Torsten Grust**, **Gábor Szárnyas** and **Daniël ten Wolde** for their valuable suggestions. We would also like to thank **Carlo Piovesan** for his translation of the Postgres YACC grammar to PEG.

DuckDB Tricks – Part 3

Publication date: 2024-11-29

Authors: Andra Ionescu, Gábor Szárnyas

TL;DR: In this new installment of the DuckDB Tricks series, we present features for convenient handling of tables and performance optimization tips for Parquet and CSV files.

Overview

We continue our DuckDB Tricks series with a third part, where we showcase friendly SQL features and performance optimizations.

Operation	SQL instructions
Excluding columns from a table	`EXCLUDE`{:.language-sql .highlight}/`COLUMNS(...)`{:.language-sql .highlight}, `NOT SIMILAR TO`{:.language-sql .highlight}
Renaming columns with pattern matching	`COLUMNS(...) AS ...`{:.language-sql .highlight}
Loading with globbing	`FROM '*.csv'`{:.language-sql .highlight}
Reordering Parquet files	`COPY (FROM ... ORDER BY ...) TO ...`{:.language-sql .highlight}
Hive partitioning	`hive_partitioning = true`{:.language-sql .highlight}

Dataset

We'll use a subset of the Dutch railway services dataset, which was already featured in a blog post earlier this year. This time, we'll use the CSV files between January and October 2024: `services-2024-01-to-10.zip`. If you would like to follow the examples, download and decompress the data set before proceeding.

Excluding Columns from a Table

First, let's look at the data in the CSV files. We pick the CSV file for August and inspect it with the `DESCRIBE` statement.

```
DESCRIBE FROM 'services-2024-08.csv';
```

The result is a table with the column names and the column types.

column_name	column_type	null	key	default	extra
Service:RDT-ID	BIGINT	YES	NULL	NULL	NULL
Service:Date	DATE	YES	NULL	NULL	NULL
Service:Type	VARCHAR	YES	NULL	NULL	NULL
Service:Company	VARCHAR	YES	NULL	NULL	NULL
Service:Train number	BIGINT	YES	NULL	NULL	NULL
…	…	…	…	…	…

Now, let's use `SUMMARIZE` to inspect some statistics about the columns.

```
SUMMARIZE FROM 'services-2024-08.csv';
```

With `SUMMARIZE`, we get 10 statistics about our data (`min`, `max`, `approx_unique`, etc.). If we want to remove a few of them the result, we can use the `EXCLUDE` modifier. For example, to exclude `min`, `max` and the quantiles `q25`, `q50`, `q75`, we can use issue the following command:

```
SELECT * EXCLUDE(min, max, q25, q50, q75)
FROM (SUMMARIZE FROM 'services-2024-08.csv');
```

Alternatively, we can use the `COLUMNS` expression with the `NOT SIMILAR TO` operator. This works with a regular expression:

```
SELECT COLUMNS(lambda c: c NOT SIMILAR TO 'min|max|q.*')
FROM (SUMMARIZE FROM 'services-2024-08.csv');
```

In both cases, the resulting table will contain the 5 remaining statistical columns:

column_name	column_type	approx_unique	avg	std	count	null_percentage
Service:RDT-ID	BIGINT	259022	14200071.03736433	59022.836209662266	1846574	0.00
Service:Date	DATE	32	NULL	NULL	1846574	0.00
Service:Type	VARCHAR	20	NULL	NULL	1846574	0.00
Service:Company	VARCHAR	12	NULL	NULL	1846574	0.00
Service:Train number	BIGINT	17264	57781.81688196628	186353.76365744913	1846574	0.00
…	…	…	…	…	…	…

Renaming Columns with Pattern Matching

Upon inspecting the columns, we see that their names contain spaces and semicolons (`:`). These special characters makes writing queries a bit tedious as they necessitate quoting column names with double quotes. For example, we have to write `"Service:Company"` in the following query:

```
SELECT DISTINCT "Service:Company" AS company,
FROM 'services-2024-08.csv'
ORDER BY company;
```

Let's see how we can rename the columns using the `COLUMNS` expression. To replace the special characters (up to 2), we can write the following query:

```
SELECT COLUMNS('(.*?)_*$') AS "\1"
FROM (
    SELECT COLUMNS('(\w*)\W*(\w*)\W*(\w*)') AS "\1_\2_\3"
    FROM 'services-2024-08.csv'
);
```

Add `DESCRIBE` at the beginning of the query and we can see the renamed columns:

column_name	column_type	null	key	default	extra
Service_RDT_ID	BIGINT	YES	NULL	NULL	NULL
Service_Date	DATE	YES	NULL	NULL	NULL
Service_Type	VARCHAR	YES	NULL	NULL	NULL
Service_Company	VARCHAR	YES	NULL	NULL	NULL
Service_Train_number	BIGINT	YES	NULL	NULL	NULL

column_name	column_type	null	key	default	extra
…	…	…	…	…	…

Let's break down the query starting with the first COLUMNS expression:

```
SELECT COLUMNS('(\w*)\W*(\w*)\W*(\w*)') AS "\1_\2_\3"
```

Here, we use regular expression with `(\w*)` groups that capture 0…n word characters (`[0-9A-Za-z_]`). Meanwhile, the expression `\W*` captures 0…n non-word characters (`[^0-9A-Za-z_]`). In the alias part we refer to the capture group `i` with `\i` so `"\1_\2_\3"` means that we only keep the word characters and separate their groups with underscores (_). However, because some column names contain words separated by a space, while others don't, after this SELECT statement we get column names with a trailing underscore (_), e.g., `Service_Date_`. Thus, we need an additional processing step:

```
SELECT COLUMNS('(.*?)_*$') AS "\1"
```

Here, we capture the group of characters without the trailing underscore(s) and rename the columns to `\1`, which removes the trailing underscores.

To make writing queries even more convenient, we can rely on the case-insensitivity of identifiers to query the column names in lowercase:

```
SELECT DISTINCT service_company
FROM (
    SELECT COLUMNS('(.*?)_*$') AS "\1"
    FROM (
       SELECT COLUMNS('(\w*)\W*(\w*)\W*(\w*)') AS "\1_\2_\3"
       FROM 'services-2024-08.csv'
    )
)
ORDER BY service_company;
```

Service_Company
Arriva
Blauwnet
Breng
DB
Eu Sleeper
…

> The returned column name preserves its original cases even though we used lowercase letters in the query.

Loading with Globbing

Now that we can simplify the column names, let's ingest all 3 months of data to a table:

```
CREATE OR REPLACE TABLE services AS
    SELECT COLUMNS('(.*?)_*$') AS "\1"
    FROM (
        SELECT COLUMNS('(\w*)\W*(\w*)\W*(\w*)') AS "\1_\2_\3"
        FROM 'services-2024-*.csv'
    );
```

In the inner FROM clause, we use the `*` glob syntax to match all files. DuckDB automatically detects that all files have the same schema and unions them together. We have now a table with all the data from January to October, amounting to almost 20 million rows.

Reordering Parquet Files

Suppose we want to analyze the average delay of the Intercity Direct trains operated by the Nederlandse Spoorwegen (NS), measured at the final destination of the train service. While we can run this analysis directly on the `.csv` files, the lack of metadata (such as schema and min-max indexes) will limit the performance. Let's measure this in the CLI client by turning on the timer:

```
.timer on
```

```
SELECT avg("Stop:Arrival delay")
FROM 'services-*.csv'
WHERE "Service:Company" = 'NS'
  AND "Service:Type" = 'Intercity direct'
  AND "Stop:Departure time" IS NULL;
```

This query takes about 1.8 seconds. Now, if we run the same query on `services` table that's already loaded to DuckDB, the query is much faster:

```
SELECT avg(Stop_Arrival_delay)
FROM services
WHERE Service_Company = 'NS'
  AND Service_Type = 'Intercity direct'
  AND Stop_Departure_time IS NULL;
```

The run time is about 35 milliseconds.

If we would like to use an external binary file format, we can also export the database to a single Parquet file:

```
EXPORT DATABASE 'railway' (FORMAT parquet);
```

We can then directly query it as follows:

```
SELECT avg(Stop_Arrival_delay)
FROM 'railway/services.parquet'
WHERE Service_Company = 'NS'
  AND Service_Type = 'Intercity direct'
  AND Stop_Departure_time IS NULL;
```

The runtime for this format is about 90 milliseconds – somewhat slower than DuckDB's own file format but about 20× faster than reading the raw CSV files.

If we have a priori knowledge of the fields a query filters on, we can reorder the Parquet file to improve query performance.

```
COPY
(FROM 'railway/services.parquet' ORDER BY Service_Company, Service_Type)
TO 'railway/services.parquet';
```

If we run the query again, it's noticeably faster, taking only 35 milliseconds. This is thanks to partial reading, which uses the zonemaps (min-max indexes) to limit the amount of data that has to be scanned. Reordering the file allows DuckDB to skip more data, leading to faster query times.

Hive Partitioning

To speed up queries even further, we can use Hive partitioning to create a directory layout on disk that matches the filtering used in the queries.

```
COPY services
TO 'services-parquet-hive'
(FORMAT parquet, PARTITION_BY (Service_Company, Service_Type));
```

Let's peek into the directory from DuckDB's CLI using the `.sh` dot command:

```
.sh tree services-parquet-hive
```

```
services-parquet-hive
├── Service_Company=Arriva
│   ├── Service_Type=Extra%20trein
│   │   └── data_0.parquet
│   ├── Service_Type=Nachttrein
│   │   └── data_0.parquet
│   ├── Service_Type=Snelbus%20ipv%20trein
│   │   └── data_0.parquet
│   ├── Service_Type=Sneltrein
│   │   └── data_0.parquet
│   ├── Service_Type=Stopbus%20ipv%20trein
│   │   └── data_0.parquet
│   ├── Service_Type=Stoptrein
│   │   └── data_0.parquet
│   └── Service_Type=Taxibus%20ipv%20trein
│       └── data_0.parquet
├── Service_Company=Blauwnet
│   ├── Service_Type=Intercity
│   │   └── data_0.parquet
...
```

We can now run the query on the Hive partitioned data set by passing the `hive_partitioning = true` flag:

```
SELECT avg(Stop_Arrival_delay)
FROM read_parquet(
         'services-parquet-hive/**/*.parquet',
         hive_partitioning = true
     )
WHERE Service_Company = 'NS'
  AND Service_Type = 'Intercity direct'
  AND Stop_Departure_time IS NULL;
```

This query now takes about 20 milliseconds as DuckDB can use the directory structure to limit the reads even further. And the neat thing about Hive partitioning is that it even works with CSV files!

```
COPY services
TO 'services-csv-hive'
(FORMAT csv, PARTITION_BY (Service_Company, Service_Type));

SELECT avg(Stop_Arrival_delay)
FROM read_csv('services-csv-hive/**/*.csv', hive_partitioning = true)
WHERE Service_Company = 'NS'
  AND Service_Type = 'Intercity direct'
  AND Stop_Departure_time IS NULL;
```

While the CSV files lack any sort of metadata, DuckDB can rely on the directory structure to limit the scans to the relevant directories, resulting in execution times around 150 milliseconds, more than 10× faster compared to reading all CSV files.

If all these formats and results got your head spinning, no worries. We got your covered with this summary table:

Format	Query runtime (ms)
DuckDB file format	35
CSV (vanilla)	1800
CSV (Hive-partitioned)	150
Parquet (vanilla)	90
Parquet (reordered)	35
Parquet (Hive-partitioned)	20

Oh, and we forgot to report the result. The average delay of Intercity Direct trains is 3 minutes!

Closing Thoughts

That's it for part three of DuckDB tricks. If you have a trick that would like to share, please share it with the DuckDB team on our social media sites, or submit it to the DuckDB Snippets site (maintained by our friends at MotherDuck).

CSV Files: Dethroning Parquet as the Ultimate Storage File Format — or Not?

Publication date: 2024-12-05

Author: Pedro Holanda

TL;DR: Data analytics primarily uses two types of storage format files: human-readable text files like CSV and performance-driven binary files like Parquet. This blog post compares these two formats in an ultimate showdown of performance and flexibility, where there can be only one winner.

File Formats

CSV Files

Data is most commonly stored in human-readable file formats, like JSON or CSV files. These file formats are easy to operate on, since anyone with a text editor can simply open, alter, and understand them.

For many years, CSV files have had a bad reputation for being slow and cumbersome to work with. In practice, if you want to operate on a CSV file using your favorite database system, you must follow this recipe:

1. Manually discover its schema by opening the file in a text editor.
2. Create a table with the given schema.
3. Manually figure out the dialect of the file (e.g., which character is used for a quote?)
4. Load the file into the table using a COPY statement and with the dialect set.
5. Start querying it.

Not only is this process tedious, but parallelizing a CSV file reader is far from trivial. This means most systems either process it single-threaded or use a two-pass approach.

Additionally, CSV files are wild: although RFC-4180 exists as a CSV standard, it is commonly ignored. Systems must therefore be sufficiently robust to handle these files as if they come straight from the wild west.

Last but not least, CSV files are wasteful: data is always laid out as strings. For example, numeric values like `1000000000` take 10 bytes instead of 4 bytes if stored as an `int32`. Additionally, since the data layout is row-wise, opportunities to apply lightweight columnar compression are lost.

Parquet Files

Due to these shortcomings, performance-driven file formats like Parquet have gained significant popularity in recent years. Parquet files cannot be opened by general text editors, cannot be easily edited, and have a rigid schema. However, they store data in columns, apply various compression techniques, partition the data into row groups, maintain statistics about these row groups, and define their schema directly in the file.

These features make Parquet a monolith of a file format — highly inflexible but efficient and fast. It is easy to read data from a Parquet file since the schema is well-defined. Parallelizing a scanner is straightforward, as each thread can independently process a row group. Filter pushdown is also simple to implement, as each row group contains statistical metadata, and the file sizes are very small.

The conclusion should be simple: if you have small files and need flexibility, CSV files are fine. However, for data analysis, one should pivot to Parquet files, right? Well, this pivot may not be a hard requirement anymore – read on to find out why!

Reading CSV Files in DuckDB

For the past few releases, DuckDB has doubled down on delivering not only an easy-to-use CSV scanner but also an extremely performant one. This scanner features its own custom CSV sniffer, parallelization algorithm, buffer manager, casting mechanisms, and state machine-based parser.

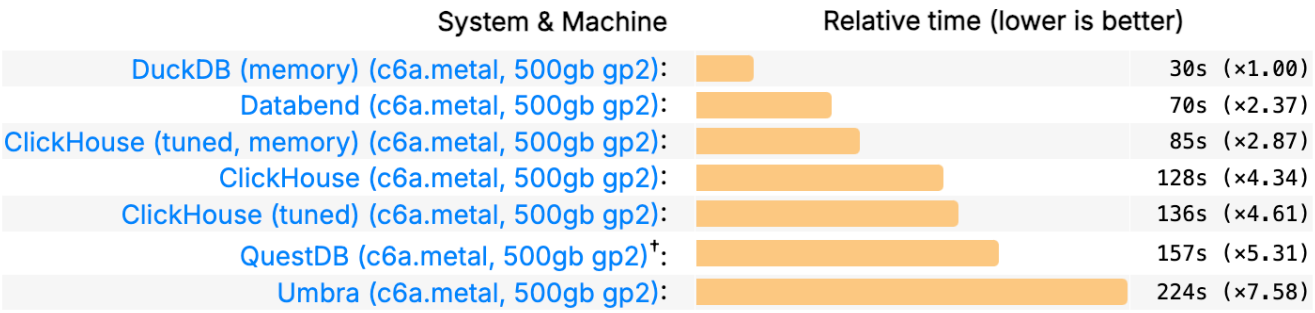
For usability, the previous paradigm of manual schema discovery and table creation has been changed. Instead, DuckDB now utilizes a CSV Sniffer, similar to those found in dataframe libraries like Pandas. This allows for querying CSV files as easily as:

```
FROM 'path/to/file.csv';
```

Or tables to be created from CSV files, without any prior schema definition with:

```
CREATE TABLE t AS FROM 'path/to/file.csv';
```

Furthermore, the reader became one of the fastest CSV readers in analytical systems, as can be seen by the load times of the latest iteration of ClickBench. In this benchmark, the data is loaded from an 82 GB uncompressed CSV file into a database table.



ClickBench CSV loading times (2024-12-05)

Comparing CSV and Parquet

With the large boost in usability and performance for the CSV reader, one might ask: what is the actual difference in performance when loading a CSV file compared to a Parquet file into a table? Additionally, how do these formats differ when running queries directly on them?

To find out, we will run a few examples using both CSV and Parquet files containing TPC-H data to shed light on their differences. All scripts used to generate the benchmarks of this blogpost can be found in a repository.

Usability

In terms of usability, scanning CSV files and Parquet files can differ significantly.

In simple cases, where all options are correctly detected by DuckDB, running queries on either CSV or Parquet files can be done directly.

```
FROM 'path/to/file.csv';
FROM 'path/to/file.parquet';
```

Things can differ drastically for wild, rule-breaking Arthur Morgan-like CSV files. This is evident from the number of parameters that can be set for each scanner. The Parquet scanner has a total of six parameters that can alter how the file is read. For the majority of cases, the user will never need to manually adjust any of them.

The CSV reader, on the other hand, depends on the sniffer being able to automatically detect many different configuration options. For example: What is the delimiter? How many rows should it skip from the top of the file? Are there any comments? And so on. This results in over 30 configuration options that the user might have to manually adjust to properly parse their CSV file. Again, this number of options is necessary due to the lack of a widely adopted standard. However, in most scenarios, users can rely on the sniffer or, at most, change one or two options.

The CSV reader also has an extensive error-handling system and will always provide suggestions for options to review if something goes wrong.

To give you an example of how the DuckDB error-reporting system works, consider the following CSV file:

```
Clint Eastwood;94
Samuel L. Jackson
```

In this file, the second line is missing the value for the second column.

```
Invalid Input Error:
CSV Error on Line: 2
Original Line: Samuel L. Jackson
Expected Number of Columns: 2 Found: 1
Possible fixes:
* Enable null padding (null_padding=true) to replace missing values with NULL
* Enable ignore errors (ignore_errors=true) to skip this row

  file = western_actors.csv
  delimiter = , (Auto-Detected)
  quote = " (Auto-Detected)
  escape = " (Auto-Detected)
  new_line = \n (Auto-Detected)
  header = false (Auto-Detected)
  skip_rows = 0 (Auto-Detected)
  comment = \0 (Auto-Detected)
  date_format =  (Auto-Detected)
  timestamp_format =  (Auto-Detected)
  null_padding = 0
  sample_size = 20480
  ignore_errors = false
  all_varchar = 0
```

DuckDB provides detailed information about any errors encountered. It highlights the line of the CSV file where the issue occurred, presents the original line, and suggests possible fixes for the error, such as ignoring the problematic line or filling missing values with `NULL`. It also displays the full configuration used to scan the file and indicates whether the options were auto-detected or manually set.

The bottom line here is that, even with the advancements in CSV usage, the strictness of Parquet files make them much easier to operate on.

Of course, if you need to open your file in a text editor or Excel, you will need to have your data in CSV format. Note that Parquet files do have some visualizers, like TAD.

Performance

There are primarily two ways to operate on files using DuckDB:

1. The user creates a DuckDB table from the file and uses the table in future queries. This is a loading process, commonly used if you want to store your data as DuckDB tables or if you will run many queries on them. Also, note that these are the only possible scenarios for most database systems (e.g., Oracle, SQL Server, PostgreSQL, SQLite, ...).
2. One might run a query directly on the file scanner without creating a table. This is useful for scenarios where the user has limitations on memory and disk space, or if queries on these files are only executed once. Note that this scenario is typically not supported by database systems but is common for dataframe libraries (e.g., Pandas).

To fairly compare the scanners, we provide the table schemas upfront, ensuring that the scanners produce the exact same data types. We also set `preserve_insertion_order = false`, as this can impact the parallelization of both scanners, and set `max_temp_directory_size = '0GB'` to ensure no data is spilled to disk, with all experiments running fully in memory.

We use the default writers for both CSV files and Parquet (with the default Snappy compression), and also run a variation of Parquet with `CODEC 'zstd', COMPRESSION_LEVEL 1`, as this can speed up querying/loading times.

For all experiments, we use an Apple M1 Max, with 64 GB RAM. We use TPC-H scale factor 20 and report the median times from 5 runs.

Creating Tables For creating the table, we focus on the `lineitem` table.

After defining the schema, both files can be loaded with a simple COPY statement, with no additional parameters set. Note that even with the schema defined, the CSV sniffer will still be executed to determine the dialect (e.g., quote character, delimiter character, etc.) and match types and names.

Name	Time (s)	Size (GB)
CSV	11.76	15.95
Parquet Snappy	5.21	3.78
Parquet Zstd	5.52	3.22

We can see that the Parquet files are definitely smaller. About 5× smaller than the CSV file, but the performance difference is not drastic.

The CSV scanner is only about 2× slower than the Parquet scanner. It's also important to note that some of the cost associated with these operations (~1-2 seconds) is related to the insertion into the DuckDB table, not the scanner itself.

However, it is still important to consider this in the comparison. In practice, the raw CSV scanner is about 3× slower than the Parquet scanner, which is a considerable difference but much smaller than one might initially think.

Directly Querying Files We will run two different TPC-H queries on our files.

Query 01. First, we run TPC-H Q01. This query operates solely on the `lineitem` table, performing an aggregation and grouping with a filter. It filters on one column and projects 7 out of the 16 columns from `lineitem`.

Therefore, this query will stress the filter pushdown, which is supported by the Parquet reader but not the CSV reader, and the projection pushdown, which is supported by both.

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) AS sum_qty,
    sum(l_extendedprice) AS sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
    avg(l_quantity) AS avg_qty,
    avg(l_extendedprice) AS avg_price,
    avg(l_discount) AS avg_disc,
    count(*) AS count_order
FROM
    lineitem
WHERE
    l_shipdate <= CAST('1996-09-02' AS date)
GROUP BY
    l_returnflag,
    l_linestatus
ORDER BY
    l_returnflag,
    l_linestatus;
```

Name	Time (s)
CSV	6.72
Parquet Snappy	0.88
Parquet Zstd	0.95

We can see that running this query directly on our file presents a much larger performance gap of approximately 7x compared to simply loading the data into the table. In the Parquet file, we can directly skip row groups that do not match our filter `l_shipdate <= CAST('1996-09-02' AS date)`. Note that this filter, eliminates approximately 30% of the data. Not only that, but we can also skip individual rows that do not match the filter. Additionally, since the Parquet format is column-oriented, we can completely skip any computation on columns that are not projected.

Unfortunately, the CSV reader does not benefit from these filters. Since it lacks partitions, it can't efficiently skip parts of the data. Theoretically, a CSV scanner could skip the computation of rows that do not match a filter, but this is not currently implemented.

Furthermore, the CSV projection skips much of the computation on a column (e.g., it does not cast or copy the value), but it still must parse the value to be able to skip it.

Query 21. Query 21 is a query that not only heavily depends on filter and projection pushdown but also relies significantly on join ordering based on statistics to achieve good performance. In this query, four different files are used and joined together.

```
SELECT
    s_name,
    count(*) AS numwait
FROM
    supplier,
    lineitem l1,
    orders,
    nation
WHERE
    s_suppkey = l1.l_suppkey
    AND o_orderkey = l1.l_orderkey
    AND o_orderstatus = 'F'
    AND l1.l_receiptdate > l1.l_commitdate
    AND EXISTS (
        SELECT
            *
        FROM
            lineitem l2
        WHERE
            l2.l_orderkey = l1.l_orderkey
            AND l2.l_suppkey <> l1.l_suppkey)
    AND NOT EXISTS (
        SELECT
            *
        FROM
            lineitem l3
        WHERE
            l3.l_orderkey = l1.l_orderkey
            AND l3.l_suppkey <> l1.l_suppkey
            AND l3.l_receiptdate > l3.l_commitdate)
    AND s_nationkey = n_nationkey
    AND n_name = 'SAUDI ARABIA'
GROUP BY
    s_name
ORDER BY
    numwait DESC,
    s_name
LIMIT 100;
```

Name	Time (s)
CSV	19.95
Parquet Snappy	2.08
Parquet Zstd	2.12

We can see that this query now has a performance difference of approximately 10×. We observe an effect similar to Query 01, but now we

also incur the additional cost of performing join ordering with no statistical information for the CSV file.

Conclusion

There is no doubt that the performance of CSV file scanning has drastically increased over the years. If we were to take a guess at the performance difference in table creation a few years ago, the answer would probably have been at least one order of magnitude.

This is excellent, as it allows data to be exported from legacy systems that do not support performance-driven file formats.

But oh boy, don't let super-convenient and fast CSV readers fool you. Your data is still best kept in self-describing, column-binary compressed formats like Parquet — or the DuckDB file format, of course! They are much smaller and more consistent. Additionally, running queries directly on Parquet files is much more beneficial due to efficient projection/filter pushdown and available statistics.

One thing to note is that there exists an extensive body of work on indexing CSV files (i.e., building statistics in a way) to speed up future queries and enable filter pushdown. However, DuckDB does not perform these operations yet.

Bottom line: Parquet is still the undisputed champion for most scenarios, but we will continue working on closing this gap wherever possible.

DuckDB: Running TPC-H SF100 on Mobile Phones

Publication date: 2024-12-06

Authors: Gábor Szárnyas, Laurens Kuiper, Hannes Mühleisen

TL;DR: DuckDB runs on mobile platforms such as iOS and Android, and completes the TPC-H benchmark faster than state-of-the-art research systems on big iron machines 20 years ago.

A few weeks ago, we set out to perform a series of experiments to answer two simple questions:

1. Can DuckDB complete the TPC-H queries on the SF100 data set when running on a new smartphone?
2. If so, can DuckDB complete a run in less than 400 seconds, i.e., faster than the system in the research paper that originally introduced vectorized query processing?

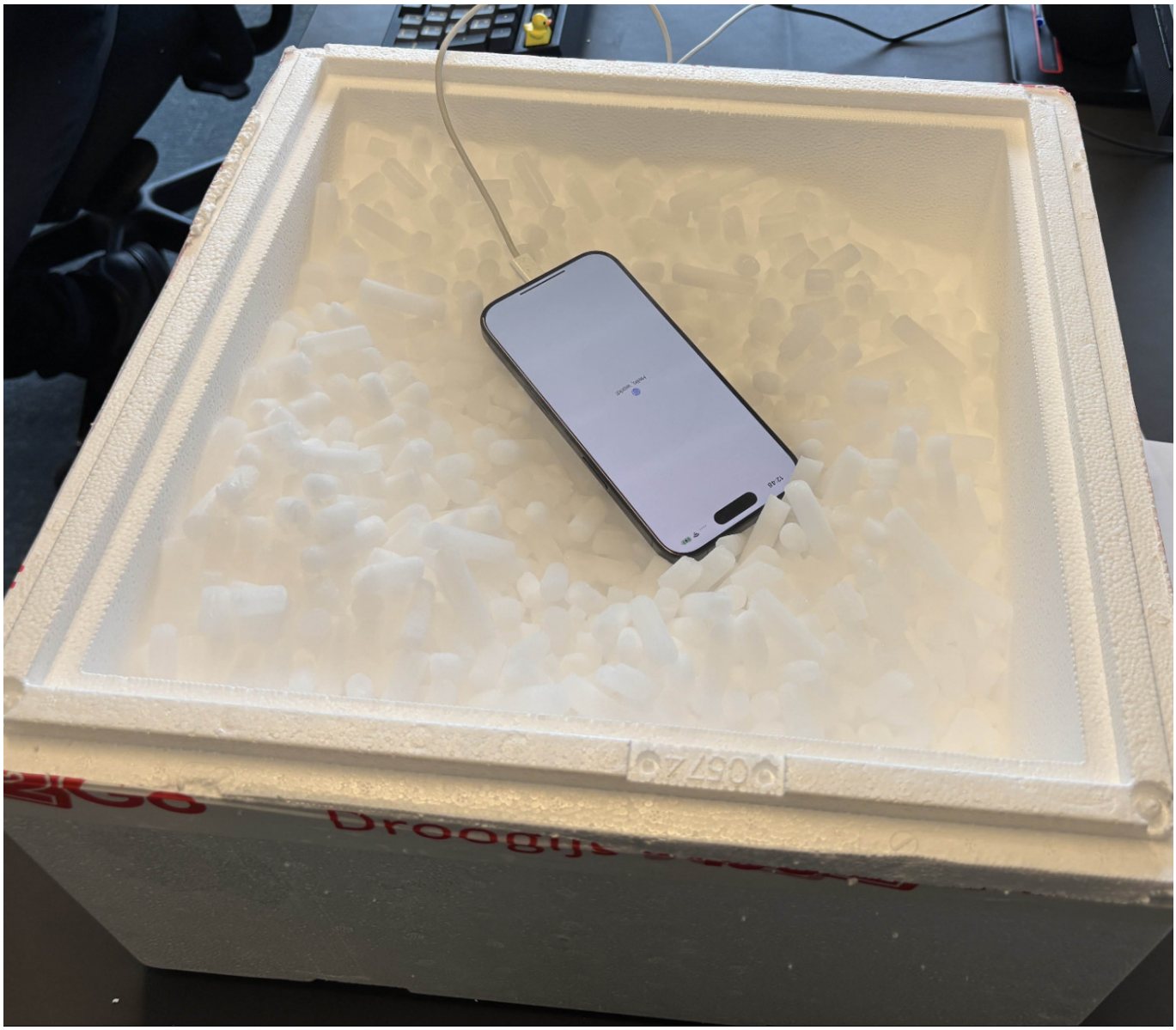
These questions took us on an interesting quest. Along the way, we had a lot of fun and learned the difference between a cold run and a *really cold* run. Read on to find out more.

A Song of Dry Ice and Fire

Our first attempt was to use an iPhone, namely an iPhone 16 Pro. This phone has 8 GB memory and a 6-core CPU with 2 performance cores (running at 4.05 GHz) and 4 efficiency cores (running at 2.42 GHz).

We implemented the application using the DuckDB Swift client and loaded the benchmark on the phone, all 30 GB of it. We quickly found that the iPhone can indeed run the workload without any problems – except that it heated up during the workload. This prompted the phone to perform thermal throttling, slowing down the CPU to reduce heat production. Due to this, DuckDB took 615.1 seconds. Not bad but not enough to reach our goal.

The results got us thinking: what if we improve the cooling of the phone? To this end, we purchased a box of dry ice, which has a temperature below -50 degrees Celsius, and put the phone in the box for the duration of the experiments.



iPhone in a box of dry ice, running TPC-H. Don't try this at home.

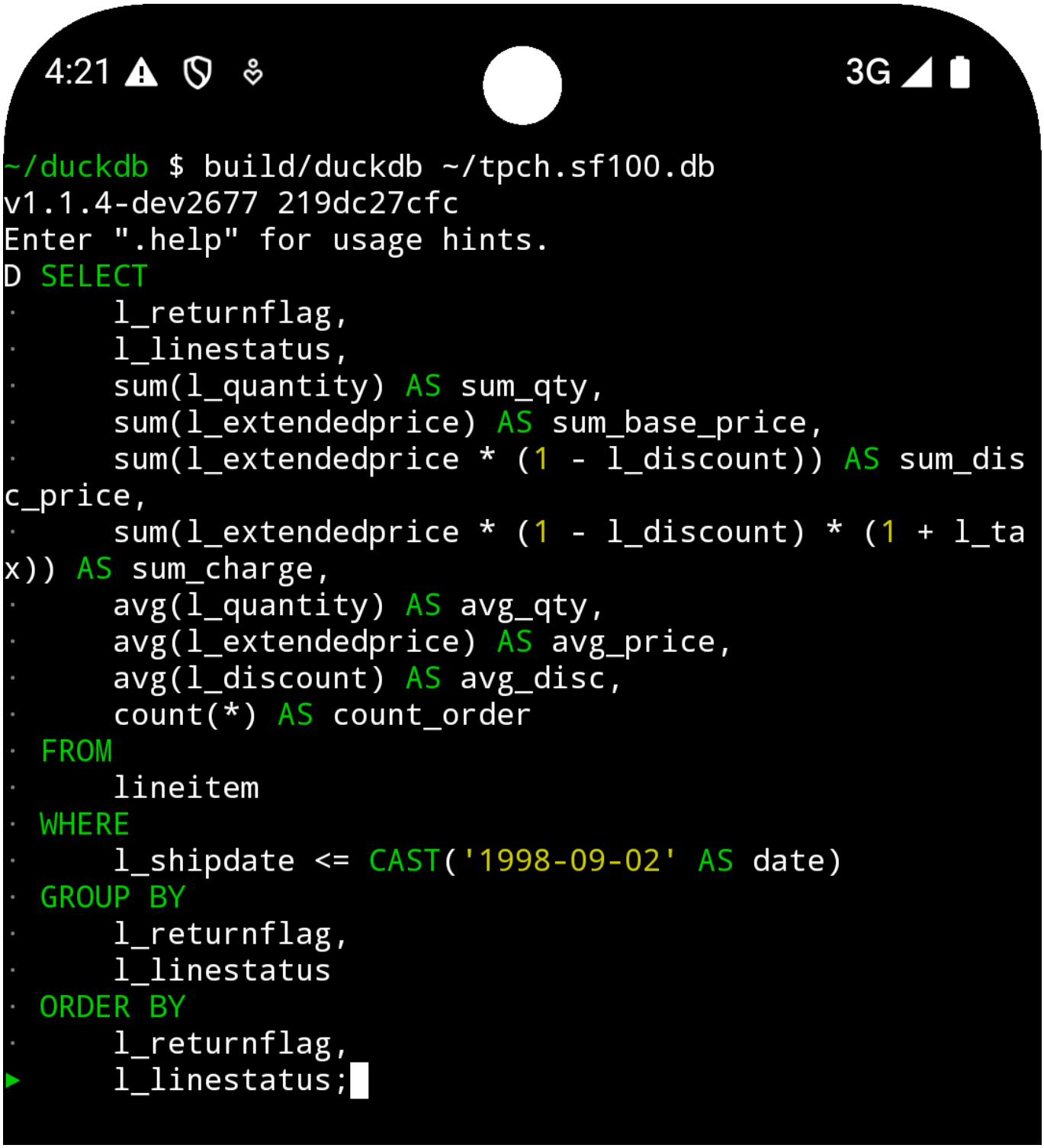
This helped a lot: DuckDB completed in 478.2 seconds. This is a more than 20% improvement – but we still didn't manage to be under 400 seconds.

The phone a few minutes after finishing the benchmark. It no longer booted because the battery was too cold!

Do Androids Dream of Electric Ducks?

In our next experiment, we picked up a Samsung Galaxy S24 Ultra phone, which runs Android 14. This phone is full of interesting hardware. First, it has an 8-core CPU with 4 different core types (1×3.39 GHz, 3×3.10 GHz, 2×2.90 GHz and 2×2.20 GHz). Second, it has a huge amount of RAM – 12 GB to be precise. Finally, its cooling system includes a vapor chamber for improved heat dissipation.

We ran DuckDB in the Termux terminal emulator. We compiled DuckDB CLI client from source following the Android build instructions and ran the experiments from the command line.



DuckDB in Termux, running in the Android emulator

In the end, it wasn't even close. The Android phone completed the benchmark in 235.0 seconds, outperforming our baseline by around 40%.

Never Was a Cloudy Day

The results got us thinking: how do the results stack up among cloud servers? We picked two x86-based cloud instances in AWS EC2 with instance-attached NVMe storage.

The details of these benchmarks are far less interesting than those of the previous ones. We booted up the instances with Ubuntu 24.04 and ran DuckDB in the command line. We found that an `r6id.large` instance (2 vCPUs with 16 GB RAM) completes the queries in 570.8 seconds, which is roughly on-par with an air-cooled iPhone. However, an `r6id.xlarge` (4 vCPUs with 32 GB RAM) completes the benchmark in 166.2 seconds, faster than any result we achieved on phones.

Summary of DuckDB Results

The table contains a summary of the DuckDB benchmark results.

Setup	CPU cores	Memory	Runtime
iPhone 16 Pro (air-cooled)	6	8 GB	615.1 s
iPhone 16 Pro (dry ice-cooled)	6	8 GB	478.2 s
Samsung Galaxy S24 Ultra	8	12 GB	235.0 s
AWS EC2 `r6id.large`	2	16 GB	570.8 s
AWS EC2 `r6id.xlarge`	4	32 GB	166.2 s

Historical Context

So why did we set out to run these experiments in the first place?

Just a few weeks ago, CWI, the birthplace of DuckDB, held a ceremony for the Dijkstra Fellowship. The fellowship was awarded to Marcin Żukowski for his pioneering role in the development of database management systems and his successful entrepreneurial career that resulted in systems such as VectorWise and Snowflake.

A lot of ideas that originate in Marcin's research are used in DuckDB. Most importantly, *vectorized query processing* allows DuckDB to be both fast and portable at the same time. With his co-authors Peter Boncz and Niels Nes, he first described this paradigm in the CIDR 2005 paper "MonetDB/X100: Hyper-Pipelining Query Execution".

> The terms *vectorization, hyper-pipelining,* and *superscalar* refer to the same idea: processing data in slices, which turns out to be a good compromise between row-at-a-time or column-at-a-time. DuckDB's query engine uses the same principle.

This paper was published in January 2005, so it's safe to assume that it was finalized in late 2004 – almost exactly 20 years ago!

If we read the paper, we learn that the experiments were carried out on an HP workstation equipped with 12 GB of memory (the same amount as the Samsung phone has today!). It also had an Itanium CPU and looked like this:

The Itanium2 workstation used in original the experiments (source: Wikimedia)

> Upon its release in 2001, the Itanium was aimed at the high-end market with the goal of eventually replacing the then-dominant x86 architecture with a new instruction set that focused heavily on SIMD (single instruction, multiple data). While this ambition did not work out, the Itanium was the state-of-the-art architecture of its day. Due to the focus on the server market, the Itanium CPUs had a large amount of cache: the 1.3 GHz Itanium2 model used in the experiments had 3 MB of L2 cache, while Pentium 4 CPUs released around that time only had 0.5–1 MB.

The paper provides a detailed breakdown of the runtimes:

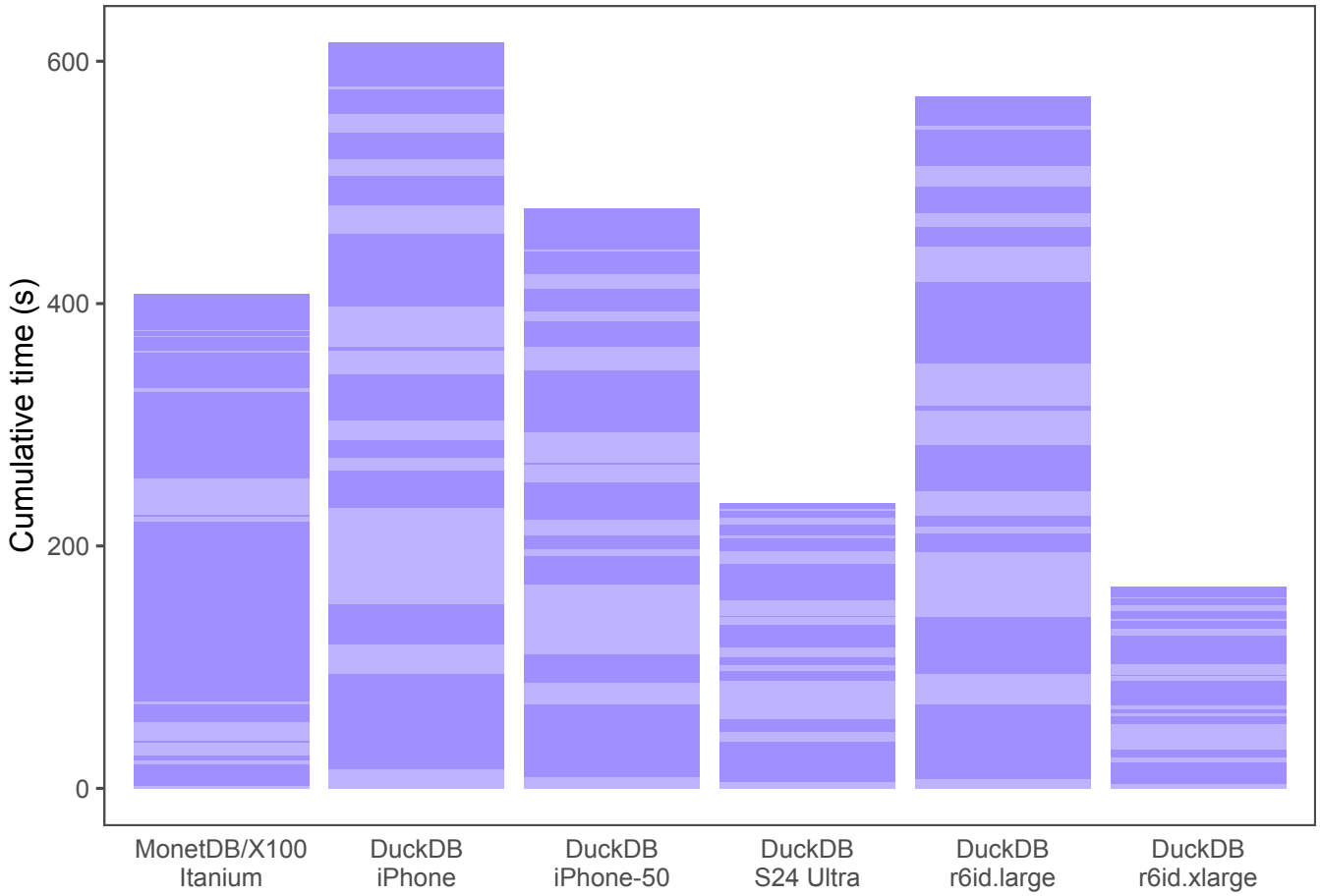
MonetDB/MIL	MonetDB/X100, 1CPU			DB2, 4CPU
Q	SF=1	SF=1	SF=100	SF=100
1	3.72	0.50	0.31	30.25	229
2	0.46	0.01	0.01	0.81	19
3	2.52	0.04	0.02	3.77	16
4	1.56	0.05	0.02	1.15	14
5	2.72	0.08	0.04	11.02	72
6	2.24	0.09	0.02	1.44	12
7	3.26	0.22	0.22	29.47	81
8	2.23	0.06	0.03	2.78	65
9	6.78	0.44	0.44	71.24	274
10	4.40	0.22	0.19	30.73	47
11	0.43	0.03	0.02	1.66	20
12	3.73	0.09	0.04	3.68	19
13	11.42	1.26	1.04	148.22	343
14	1.03	0.02	0.02	2.64	14
15	1.39	0.09	0.04	14.36	30
16	2.25	0.21	0.14	15.77	64
17	2.30	0.02	0.02	1.75	77
18	5.20	0.15	0.11	10.37	600
19	12.46	0.05	0.05	4.47	81
20	2.75	0.08	0.05	2.45	35
21	8.85	0.29	0.17	17.61	428
22	3.07	0.07	0.04	2.30	93
	AthlonMP		Itanium2	

Table 4: TPC-H Performance (seconds)

Benchmark results from the paper “MonetDB/X100: Hyper-Pipelining Query Execution”

The total runtime of the TPC-H SF100 queries was 407.9 seconds – hence our baseline for the experiments. Here is a video of Hannes presenting the results at the event:

And here are all results visualized on a plot:



TPC-H SF100 total query runtimes for MonetDB/X100 and DuckDB

Conclusion

It was a long journey from the original vectorized execution paper to running an analytical database on a phone. Many key innovations happened that allowed these results, and the big improvement in hardware is just one of them. Another crucial component is that compiler optimizations became a lot more sophisticated. Thanks to this, while the MonetDB/X100 system needed to use explicit SIMD, DuckDB can rely on the auto-vectorization of our (carefully constructed) loops.

All that's left is to answer questions that we posed at the beginning of our journey. Yes, DuckDB can run TPC-H SF100 on a mobile phone. And yes, in some cases it can even outperform a research prototype running on a high-end machine of 2004 – on a modern smartphone that fits in your pocket.

And with newer hardware, smarter compilers and yet-to-be-discovered database optimizations, future versions are only going to be faster.

The DuckDB Avro Extension

Publication date: 2024-12-09

Author: Hannes Mühleisen

TL;DR: DuckDB now supports reading Avro files.

> **Update.** Avro support is now available through the `avro` core extension.

The Apache™ Avro™ Format

Avro is a binary format for record data. Like many innovations in the data space, Avro was developed by Doug Cutting as part of the Apache Hadoop project in around 2009. Avro gets its name – somewhat obscurely – from a defunct British aircraft manufacturer. The company famously built over 7,000 Avro Lancaster heavy bombers under the challenging conditions of World War 2. But we digress.

The Avro format is yet another attempt to solve the dimensionality reduction problem that occurs when transforming a complex *multi-dimensional data structure* like tables (possibly with nested types) to a *single-dimensional storage layout* like a flat file, which is just a sequence of bytes. The most fundamental question that arises here is whether to use a columnar or a row-major layout. Avro uses a row-major layout, which differentiates it from its famous cousin, the Apache™ Parquet™ format. There are valid use cases for a row-major format: for example, appending a few rows to a Parquet file is difficult and inefficient because of Parquet's columnar layout and due to the fact the Parquet metadata is stored *at the back* of the file. In a row-major format like Avro with the metadata *up top*, we can "just" add those rows to the end of the files and we're done. This enables Avro to handle appends of a few rows somewhat efficiently.

Avro-encoded data can appear in several ways, e.g., in RPC messages but also in files. In the following, we focus on files since those survive long-term.

Header Block

Avro "object container" files are encoded using a comparatively simple binary format: each file starts with a **header block** that first has the magic bytes `Obj1`. Then, a metadata "map" (a list of string-bytearray key-value pairs) follows. The map is only strictly required to contain a single entry for the `avro.schema` key. This key contains the Avro file schema encoded as JSON. Here is an example for such a schema:

```
{
  "namespace": "example.avro",
  "type": "record",
  "name": "User",
  "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number", "type": ["int", "null"]},
     {"name": "favorite_color", "type": ["string", "null"]}
  ]
}
```

The Avro schema defines a record structure. Records can contain scalar data fields (like `int`, `double`, `string`, etc.) but also more complex types like records (similar to DuckDB STRUCTs), unions and lists. As a sidenote, it is quite strange that a data format for the definition of record structures would fall back to another format like JSON to describe itself, but such are the oddities of Avro.

Data Blocks

The header concludes with 16 randomly chosen bytes as a "sync marker". The header is followed by an arbitrary amount of **data blocks**: each data block starts with a record count, followed by a size and a byte array containing the actual records. Optionally, the bytes can be compressed with deflate (gzip), which will be known from the header metadata.

The data bytes can only be decoded using the schema. The object file specification contains the details on how each type is encoded. For example, in the example schema we know each value is a record of three fields. The root-level record will encode its entries in the order they are declared. There are no actual bytes required for this. First we will be reading the `name` field. Strings consist of a length followed by the string bytes. Like other formats (e.g., Thrift), Avro uses variable-length integers with zigzag encoding to store lengths and counts and the like. After reading the string, we can proceed to `favorite_number`. This field is a union type (encoded with the `[]` syntax). This union can have values of two types, `int` and `null`. The `null` type is a bit odd, it can only be used to encode the fact that a value is missing. To decode the `favorite_number` fields, we first read an `int` that encodes which choice of the union was used. Afterward, we use the "normal" decoders to read the values (e.g., `int` or `null`). The same can be done for `favorite_color`. Each data block again ends with the sync marker. The sync marker can be used to verify that the block was fully written and that there is no garbage in the file.

The DuckDB `avro` Community Extension

We have developed a DuckDB community extension that enables DuckDB to *read* Apache Avro™ files.

The extension does not contain Avro *write* functionality. This is on purpose, by not providing a writer we hope to decrease the amount of Avro files in the world over time.

Installation & Loading

Installation is simple through the DuckDB community extension repository, just type

```
INSTALL avro FROM community;
LOAD avro;
```

in a DuckDB instance near you.

> Since DuckDB v1.2.1, DuckDB's WebAssembly client is also supported.

The `read_avro` Function

The extension adds a single DuckDB function, `read_avro`. This function can be used like so:

```
FROM read_avro('some_example_file.avro');
```

This function will expose the contents of the Avro file as a DuckDB table. You can then use any arbitrary SQL constructs to further transform this table.

File IO

The `read_avro` function is integrated into DuckDB's file system abstraction, meaning you can read Avro files directly from e.g., HTTP or S3 sources. For example:

```
FROM read_avro('https://blobs.duckdb.org/data/userdata1.avro');
FROM read_avro('s3://<my-example-bucket>/some_example_file.avro');
```

should "just" work.

You can also *glob* multiple files in a single read call or pass a list of files to the functions:

```
FROM read_avro('some-example-file-*.avro');
FROM read_avro(['some-example-file-1.avro', 'some-example-file-2.avro']);
```

If the filenames somehow contain valuable information (as is unfortunately all-too-common), you can pass the `filename` argument to `read_avro`:

```
FROM read_avro('some-example-file-*.avro', filename = true);
```

This will result in an additional column in the result set that contains the actual filename of the Avro file.

Schema Conversion

This extension automatically translates the Avro Schema to the DuckDB schema. *All* Avro types can be translated, except for *recursive type definitions*, which DuckDB does not support.

The type mapping is very straightforward except for Avro's "unique" way of handling `NULL`. Unlike other systems, Avro does not treat `NULL` as a possible value in a range of e.g., `INTEGER` but instead represents `NULL` as a union of the actual type with a special `NULL` type. This is different to DuckDB, where any value can be `NULL`. Of course DuckDB also supports `UNION` types, but this would be quite cumbersome to work with.

This extension *simplifies* the Avro schema where possible: an Avro union of any type and the special null type is simplified to just the non-null type. For example, an Avro record of the union type `["int", "null"]` (like `favorite_number` in the example) becomes a DuckDB `INTEGER`, which just happens to be `NULL` sometimes. Similarly, an Avro union that contains only a single type is converted to the type it contains. For example, an Avro record of the union type `["int"]` also becomes a DuckDB `INTEGER`.

The extension also "flattens" the Avro schema. Avro defines tables as root-level "record" fields, which are the same as DuckDB `STRUCT` fields. For more convenient handling, this extension turns the entries of a single top-level record into top-level columns.

Implementation

Internally, this extension uses the "official" Apache Avro C API, albeit with some minor patching to allow reading Avro files from memory.

Limitations & Next Steps

In the following, we disclose the limitations of the `avro` DuckDB extension along with our plans to mitigate them in the future:

- The extension currently does not make use of **parallelism** when reading either a single (large) Avro file or when reading a list of files. Adding support for parallelism in the latter case is on the roadmap.
- There is currently no support for projection or filter **pushdown**, but this is also planned at a later stage.
- As mentioned above, DuckDB cannot express recursive type definitions that Avro has. This is unlikely to ever change.
- There is no support to allow users to provide a separate Avro schema file. This is unlikely to change, all Avro files we have seen so far had their schema embedded.
- There is currently no support for the `union_by_name` flag that other readers in DuckDB support. This is planned for the future.

Conclusion

The new `avro` community extension for DuckDB enables DuckDB to read Avro files directly as if they were tables. If you have a bunch of Avro files, go ahead and try it out! We'd love to hear from you if you run into any issues.

25 000 Stars on GitHub

Publication date: 2024-12-16

Author: The DuckDB team

TL;DR: We have recently reached 25 000 stars on GitHub. We would like to use this occasion to stop and reflect about DuckDB's recent year and our future plans.

Our GitHub repository has just passed 25,000 stars. This is great news and since it is also the end of the year it is a good moment to reflect on DuckDB's trajectory. There has been a lot of new and exciting adoption of DuckDB across the industry.

We would like to highlight two main events that have happened this year:

- We released DuckDB 1.0.0. This version introduced a stable storage format which guarantees backwards compatibility and limited forward compatibility.
- We started the DuckDB Community Extensions project. Community extensions allow developers to contribute packages to DuckDB and users to easily install these extensions using the simple command `INSTALL xyz FROM community`.

Besides the GitHub stars we have also observed a lot of growth in various metrics.

- Each month, our website handles over 1.5 million unique visitors. In addition, we see over 300 TB in traffic from ca. 30 million extension downloads. Thanks again to Cloudflare for sponsoring the project with free content delivery services!
- In one year, we rose in the DB Engines ranking from position 91 to 55 on the general board and from position 47 to 33 in the relational board, which makes DuckDB the fastest growing relational system in the top-50.
- We count 7.5M+ monthly downloads in PyPI.
- Maven Central downloads for the JDBC driver have also shot up, we now see over 500k+ downloads per month.

We should note that we're not glorifying those numbers and they are not a target per se for our much-beloved optimization in accordance with Goodhart's law. Still, they are just motivating to see grow.

As an aside, we have recently opened a Bluesky account and are seeing great discussions happening over there. The account has already exceeded 4 thousand followers!

Following our ancient two-year tradition, we hosted two DuckCon events, one in Amsterdam and another in Seattle. We also organized the first DuckDB Amsterdam Meetup.

Early next year, we are going to host DuckCon in Amsterdam, which is going to be the first event that we live stream in order to be more accessible to the growing DuckDB users in, e.g., Asia. But for now, let's sit around the syntax tree and be merry thinking about what's to come.

SELECT
INSERT
CASE
QUALIFY
CAST
ANY
UNION
CREATE
LIMIT

DuckDB Node Neo Client

Publication date: 2024-12-18

Author: Jeff Raymakers

TL;DR: The new DuckDB Node client, “Neo”, provides a powerful and friendly way to use your favorite database

Meet the newest DuckDB client API: DuckDB Node “Neo”!

You may be familiar with DuckDB’s old Node client. While it has served the community well over the years, “Neo” aims to learn from and improve upon its predecessor. It presents a friendlier API, supports more features, and uses a more robust and maintainable architecture. It provides both high-level conveniences and low-level access. Let’s take a tour!

What Does It Offer?

Friendly, Modern API

The old Node client’s API is based on SQLite’s. While familiar to many, it uses an awkward, dated callback-based style. Neo uses Promises natively.

```
const result = await connection.run(` SELECT 'Hello, Neo!'`);
```

Additionally, Neo is built from the ground up in TypeScript. Carefully chosen names and types minimize the need to check documentation.

```
const columnNames = result.columnNames();
const columnTypes = result.columnTypes();
```

Neo also provides convenient helpers to read only as many rows as needed and return them in either column-major or row-major format.

```
const reader = await connection.runAndReadUtil('FROM range(5000)',
    1000);
const rows = reader.getRows();
// OR: const columns = reader.getColumns();
```

Full Data Type Support

DuckDB supports a rich variety of data types. Neo supports every built-in type as well as custom types such as JSON. For example, ARRAY:

```
if (columnType.typeId === DuckDBTypeId.ARRAY) {
  const arrayValueType = columnType.valueType;
  const arrayLength = columnType.length;
}
```

DECIMAL:

```
if (columnType.typeId === DuckDBTypeId.DECIMAL) {
  const decimalWidth = columnType.width;
  const decimalScale = columnType.scale;
}
```

And JSON:

```
if (columnType.alias === 'JSON') {
  const json = JSON.parse(columnValue);
}
```

Type-specific utilities ease common conversions such as producing human-readable strings from TIMESTAMPs or DECIMALs, while preserving access to the raw values for lossless processing.

```
if (columnType.typeId === DuckDBTypeId.TIMESTAMP) {
  const timestampMicros = columnValue.micros; // bigint
  const timestampString = columnValue.toString();
  const {
    date: { year, month, day },
    time: { hour, min, sec, micros },
  } = columnValue.toParts();
}
```

Advanced Features

Need to bind specific types of values to prepared statements, or precisely control SQL execution? Perhaps you want to leverage DuckDB's parser to extract statements, or efficiently append data to a table. Neo has you covered, providing full access to these powerful features of DuckDB.

Binding Values to Prepared Statements When binding values to parameters of prepared statements, you can select the SQL data type. This is useful for types that don't have a natural equivalent in JavaScript.

```
const prepared = await connection.prepare('SELECT $1, $2');
prepared.bindTimestamp(1, new DuckDBTimestampValue(micros));
prepared.bindDecimal(2, new DuckDBDecimalValue(value, width, scale));
const result = await prepared.run();
```

Controlling Task Execution Using pending results allows pausing or stopping SQL execution at any point, even before the result is ready.

```
import { DuckDBPendingResultState } from '@duckdb/node-api';

// Placeholder to demonstrate doing other work between tasks.
async function sleep(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

const prepared = await connection.prepare('FROM range(10_000_000)');
const pending = prepared.start();
// Run tasks until the result is ready.
// This allows execution to be paused and resumed as needed.
// Other work can be done between tasks.
while (pending.runTask() !== DuckDBPendingResultState.RESULT_READY) {
  console.log('not ready');
  await sleep(1);
}
console.log('ready');
const result = await pending.getResult();
// ...
```

Extracting Statements and Running Them with Parameters You can run multi-statement SQL containing parameters using the extract statements API.

```
// Parse this multi-statement input into separate statements.
const extractedStatements = await connection.extractStatements(`
  CREATE OR REPLACE TABLE numbers AS FROM range(?);
  FROM numbers WHERE range < ?;
  DROP TABLE numbers;
`);
```

```
const parameterValues = [10, 7];
const stmtCount = extractedStatements.count;
// Run each statement, binding values as needed.
for (let stmtIndex = 0; stmtIndex < stmtCount; stmtIndex++) {
  const prepared = await extractedStatements.prepare(stmtIndex);
  const paramCount = prepared.parameterCount;
  for (let paramIndex = 1; paramIndex <= paramCount; paramIndex++) {
    prepared.bindInteger(paramIndex, parameterValues.shift());
  }
  const result = await prepared.run();
  // ...
}
```

Appending Data to a Table The appender API is the most efficient way to bulk insert data into a table.

```
await connection.run(
  `CREATE OR REPLACE TABLE target_table(i INTEGER, v VARCHAR)`
);

const appender = await connection.createAppender('main', 'target_table');

appender.appendInteger(100);
appender.appendVarchar('walk');
appender.endRow();

appender.appendInteger(200);
appender.appendVarchar('swim');
appender.endRow();

appender.appendInteger(300);
appender.appendVarchar('fly');
appender.endRow();

appender.close();
```

How Is It Built?

Dependencies

Neo uses a different implementation approach from most other DuckDB client APIs, including the old Node client. It binds to DuckDB's C API instead of the C++ API.

Why should you care? Using DuckDB's C++ API means building all of DuckDB from scratch. Each client API using this approach ships with a slightly different build of DuckDB. This can create headaches for both library maintainers and consumers.

Maintainers need to pull in the entire DuckDB source code. This increases the cost and complexity of the build, and thus the cost of code changes and especially DuckDB version updates. These costs often lead to significant delays in fixing bugs or supporting new versions.

Consumers are impacted by these delays. There's also the possibility of subtle behavioral differences between the builds in each client, perhaps introduced by different compile-time configuration.

> Some client APIs reside in the main DuckDB repository. This addresses some of the problems above, but increases the cost and complexity of maintaining DuckDB itself.

To use DuckDB's C API, on the other hand, one only needs to depend on released binaries. This significantly simplifies the maintenance required, speeds up builds, and minimizes the cost of updates. It removes the uncertainty and risk of rebuilding DuckDB.

Packages

DuckDB requires different binaries for each platform. Distributing platform-specific binaries in Node packages is notoriously challenging. It can often lead to inscrutable errors when installing, when the package manager attempts to rebuild some component from source, using whatever build and configuration tools happen to be around.

Neo uses a package design aimed to avoid these problems. Inspired by ESBuild, Neo packages pre-built binaries for each supported platform in a separate package. Each of these packages declares the particular platform (e.g., `os` and `cpu`) it supports. Then, the main package depends on all these platform-specific packages using `optionalDependencies`.

When the main package is installed, the package manager will only install optionalDependencies for supported platforms. So you only get exactly the binaries you need, no more. If installed on an unsupported platform, no binaries will be installed. At no point will an attempt to build from source occur during install.

Layers

The DuckDB Node Neo client has multiple layers. Most people will want to use Neo's main "api" package, @duckdb/node-api. This contains the friendly API with convenient helpers. But, for advanced use cases, Neo also exposes the lower-level "bindings" package, @duckdb/node-bindings, which implements a more direct translation of DuckDB's C API into Node.

This API has TypeScript definitions, but, as it follows the conventions of C, it can be awkward to use from Node. However, it provides a relatively unopinionated way to access DuckDB, which supports building special-purpose applications or alternate higher-level APIs.

Where Is It Headed?

Neo is currently marked "alpha". This is an indication of completeness and maturity, not robustness. Most of the functionality of DuckDB's C API is exposed, and what is exposed has extensive tests. But it's relatively new, so it may contain undiscovered bugs.

Additionally, some areas of functionality are not yet complete:

- Appending and binding advanced data types. These require additional functions in DuckDB's C API. The goal is to add these for the next release of DuckDB 1.2, currently planned for January 2025.
- Writing to data chunk vectors. Modifying binary buffers in a way that can be seen by a native layer presents special challenges in the Node environment. This is a high priority to work on in the near future.
- User-defined types & functions. The necessary functions and types were added to the DuckDB C API relatively recently, in v1.1.0. This is on the near-term roadmap.
- Profiling info. This was added in v1.1.0. It's on the roadmap.
- Table descriptions. This was also added in v1.1.0. It's on the roadmap.

New versions of DuckDB will include additions to the C API. Since Neo aims to cover all the functionality of the C API, these additions will be added to the roadmap as they are released.

If you have a feature request, or other feedback, let us know! Pull requests are also welcome.

What Now?

DuckDB Node Neo provides a friendly and powerful way to use DuckDB with Node. By leveraging DuckDB's C API, it exemplifies a new, more maintainable way to build on DuckDB, providing benefits to maintainers and consumers alike. It's still young, but growing up fast. Try it yourself!

Vertical Stacking as the Relational Model Intended: UNION ALL BY NAME

Publication date: 2025-01-10

Author: Alex Monahan

TL;DR: DuckDB allows vertical stacking of datasets by column name rather than position. This allows DuckDB to read files with schemas that evolve over time and finally aligns SQL with Codd's relational model.

Overview

Ever heard of SQL's `CORRESPONDING` keyword? Yeah, me neither! Well, it has been in the SQL standard since at least 1992, and almost nobody implemented it! `CORRESPONDING` was an attempt to fix a flaw in SQL – but it failed. It's time for SQL to get back to the relational model's roots when stacking data. Let's wind the clocks back to 1969...

You just picked up your own Ford Mustang Boss 302, drifting around the corner at every street to make it to the library to read the latest research report out of IBM by Edgar Codd. (Do we need a Neflix special about databases?) Reading that report, wearing plenty of plaid, you gain a critical insight: data should be treated as unordered sets! (Technically multisets – duplicates are everywhere...) Rows should be treated as unordered and so should columns. The relational model is *the way*. Any language built atop the relational model should absolutely follow those core principles.

A few years later, you learn about SQL, and it looks like a pretty cool idea. Declarative, relational – none of this maintaining order business. You don't want to be tied down by an ordering, after all. What if you change your mind about how to query your data? Sets are the best way to think about these things.

More time passes, and then, you have the need to stack some data in SQL. Should be easy enough – I can just take two tables and stack them, and the corresponding attributes will map together. No need to worry about ordering, and certainly no need to make sure that the relations are exactly the same width.

Wait. This can't be right.

I have to get the order of my columns exactly right? And I have to have the exact same number of columns in both relations? Did these SQL folks forget about Codd??

Fast forward just a couple of decades, and DuckDB is making stacking in SQL totally groovy again.

Making Vertical Stacking Groovy Again

In addition to the traditional `UNION` and `UNION ALL` operators, DuckDB adds both `UNION BY NAME` and `UNION ALL BY NAME`. These will vertically stack multiple relations (e.g., `SELECT` statements) by matching on the names of columns independent of their order. As an example, we provide columns a and b out of order, and even introduce the entirely new column c and stacking will still succeed:

```
SELECT
    42 AS a,
    'woot' AS b

UNION ALL BY NAME

SELECT
    'woot2' AS b,
    9001 AS a,
    'more wooting' AS c;
```

a	b	c
42	woot	NULL
9001	woot2	more wooting

> Any column that is not present in all relations is filled in with NULL in the places where it is missing.

This capability unlocks a variety of useful patterns that can add flexibility and save time. Some examples include:

- Stacking datasets that have different column orders
- Adding new columns to an analysis, but only for a portion of the rows
- Combining completely unrelated datasets into a single resultset
  - This can be useful if your IDE, BI tool, or API can only return a single resultset at a time, but you need to view multiple datasets

> DuckDB has had this capability since August of 2022, but the performance and scalability of this feature has recently been greatly improved! See the end of the post for some micro-benchmarks.

`UNION` vs. `UNION ALL`

If only using the keyword `UNION`, duplicates are removed when stacking. With `UNION ALL`, duplicates are permitted and the stacking occurs without additional processing.

Unfortunately we have Codd to thank for this confusing bit! If only `UNION ALL` were the default... Typically, `UNION ALL` (and its new counterpart `UNION ALL BY NAME`!) are the desired behavior as they faithfully reproduce the input relations, just stacked together. This is higher performance as well, since the deduplication that occurs with `UNION` can be quite time intensive with large datasets. And finally, `UNION ALL` preserves the original row order.

Reading Multiple Files

This column matching functionality becomes particularly useful when querying data from multiple files with different schemas. DuckDB provides a `union_by_name` boolean parameter in the table functions used to pull external flat files:

- `read_csv`
- `read_json`
- `read_parquet`

To read multiple files, DuckDB can use glob patterns within the file path parameter (or a list of files, or a list of glob patterns!). If those files could have different schemas, adding `union_by_name=True` will allow them to be read and stacked! Any columns that do not appear in a particular file will be filled with `NULL` values. For example:

```
COPY (SELECT 'Star' AS col1) TO 'star.parquet';
COPY (SELECT 'Wars' AS col2) TO 'wars.parquet';

FROM read_parquet(
    ['star.parquet', 'wars.parquet'],
    union_by_name = true);
```

col1	col2
Star	NULL
NULL	Wars

> If your files have different schemas and you did not expect it, DuckDB's friendly error messages will suggest the `union_by_name` parameter! There is no need for memorization:
>
> `If you are trying to read files with different schemas, try setting union_by_name=True`

Data Lakes

It is very common to have schema changes over time in data lakes, so this unlocks many additional uses for DuckDB in those environments. The secondary effect of this feature is that you may now feel free to change your data lake schemas freely! Now it is painless to add more attributes to your data lake over time – DuckDB will be ready to handle the analysis!

> DuckDB's extensions to read lakehouse table formats like Delta and Iceberg handle schema evolution within the formats' own metadata, so `union_by_name` is not needed.

Inserting Data by Name

Another use case for vertically stacking data is when inserting into an existing table. The DuckDB syntax of `INSERT INTO <my_table> BY NAME{:.language-sql .highlight}` offers the same flexibility of referring to columns by name rather than by position. This allows you to provide the data to insert with any column order and even including only a subset of columns. For example:

```sql
CREATE TABLE year_info (year INTEGER, status VARCHAR);

INSERT INTO year_info BY NAME
    SELECT
        'The planet made it through' AS status,
        2024 AS year;

INSERT INTO year_info BY NAME
    SELECT
        2025 AS year;

FROM year_info;
```

year	status
2024	The planet made it through
2025	NULL

The pre-existing alternative approach was to provide an additional clause that specified the list of columns to be added in the same order as the dataset. However, this requires the ordering and number of columns to be known up front rather than determined dynamically. In many cases it also requires specifying columns in two locations: the `INSERT` statement and the `SELECT` statement producing the data. Ignoring the sage advice of "Don't Repeat Yourself" has led to more than a few unintended consequences in my own code... It is always nicer to have a single location to edit rather than having to keep things in sync!

The Inspirations for `UNION ALL BY NAME`

Other systems and communities have tackled the challenges of stacking messy data for many years. DuckDB takes inspiration from them and brings their improvements back into SQL!

The most direct inspiration is the Pandas `concat` function. It was added in January of 2012, and from the very beginning it supported the addition of new columns. Pandas is incredibly widely used and is a significant contributor to the popularity of Python today. Bringing this capability to SQL can broaden its impact beyond Python and into the other languages that DuckDB supports (Java, Node.js, Go, Rust, etc.). Databases should learn from dataframes!

PySpark added the function `unionByName` in 2018 and added the ability to handle the addition of new columns in version 3.1 in March of 2021. This is another option for Pythonistas, but carries with it the requirement for a Spark cluster and its overhead.

SQL's `UNION` clause had the `CORRESPONDING` keyword since 1992 (!) at the latest, but critically it lacks the ability to handle new or missing columns. As a result, it is useless for handling schema evolution.

It is our hope that we inspire other SQL engines to become "friendlier" and allow for this flexibility!

Improved Performance in DuckDB 1.1

DuckDB has supported `UNION ALL BY NAME` since 2022, but version 1.1 brought some significant scalability and performance improvements. This feature used to be an "if you have to" approach, but can now be used more broadly!

The first change reduced memory usage when reading multiple files over the network using `union_by_name`. This provides scalability benefits when querying from cloud object storage like S3, especially when the files are large relative to available memory.

The second change was to parallelize reads across files when using `union_by_name`. This expectedly provides a dramatic performance improvement (~6× in the microbenchmark in the PR).

Micro-Benchmark

This micro-benchmark is a reproduction of the work done by Daniel Beach (@DataEngDude) in this post. Thanks to Daniel for his permission to reuse his benchmark for this post!

The benchmark requires reading 16 GB of CSV files stored on S3 that have changing schemas on a cloud instance with 4 GB of memory. The intent behind it is to process large datasets on small commodity hardware (which is a use case where we want to see DuckDB be helpful!). The original post uses Linode, but for this post we selected the most similar AWS instance having the same amount of memory (`c5d.large`).

We use two quarters' of CSV files from the Backblaze dataset (2023 Q2 and 2023 Q3), which are placed in an S3 bucket.

I modified the query from here very slightly to remove the `ignore_errors = true` option. The benchmark continued to use Python, but I'm just showing the SQL here for better syntax highlighting:

```
CREATE OR REPLACE VIEW metrics AS
    SELECT
        date,
        sum(failure) AS failures
    FROM read_csv_auto('<s3_path>/*.csv', union_by_name = true)
    GROUP BY date;

COPY metrics TO '<s3_path>/results/results.csv';
```

When using a 4 GB instance and an older version of DuckDB (1.0.0), I am able to replicate the out of memory errors that Daniel encountered. If I upgrade to DuckDB 1.1.3, the queries run successfully! However, they required about 5.8 minutes to complete.

As I dug more deeply into the dataset, I discovered that the columns selected in the benchmark query are present in each file. In prior versions of DuckDB, just having files with different sets of columns would require the `union_by_name = True` flag, even if the inconsistent or new columns were not used in the query. However, between the original post and version 1.1.3, DuckDB added the capability to do projection pushdown into CSV files! This means that only the columns used in the query are actually read from the CSV, not all columns. As a result, we can actually remove the `union_by_name = true` for the benchmark query and run successfully. This requires less overhead (since we do not need to invest time checking if all schemas match – we can rely on the first schema that is read). The simplified query runs in only 4 minutes, but it fails to exercise the capability we discussed – handling schema evolution!

To exercise the `BY NAME` capability, we add a column to the SQL query that is present only in some of the files.

```
CREATE OR REPLACE VIEW metrics AS
    SELECT
        date,
        count(DISTINCT datacenter) AS datacenters,
```

```
        sum(failure) AS failures
    FROM read_csv_auto('<s3_path>/*.csv', union_by_name = true)
    GROUP BY date;

COPY metrics TO '<s3_path>/results/results.csv';
```

This query runs in approximately the same amount of time as the original (5.6 minutes), so it is a good proxy for the original while showcasing how DuckDB handles schema evolution!

I then made a few tweaks to improve the performance. The first change is to skip the creation of a view and complete the operations all in one step. The reason this improves performance is that DuckDB will try to ensure that a view is correctly defined by binding it when it is created. Normally, this has negligible overhead (views are a great abstraction!), however when reading from cloud object storage and using `UNION ALL BY NAME`, this triggers a check of the schema of each file, which can take time. In this case, around 2 minutes! The updated SQL statement looks like this:

```
COPY (
    SELECT
        date,
        count(DISTINCT datacenter) AS datacenters,
        sum(failure) AS failures
    FROM read_csv_auto('<s3_path>/*.csv', union_by_name = true)
    GROUP BY date
) TO '<s3_path>/results/results.csv';
```

Performance improves to about 4.1 minutes with this change and also reduces the test down to a single query.

We can quantify the overhead of the flexibility that `UNION ALL BY NAME` provides if we keep the improved subquery syntax, but once again remove the `datacenter` column and the `union_by_name` flag.

```
COPY (
    SELECT
        date,
        sum(failure) AS failures
    FROM read_csv_auto('<s3_path>/*.csv')
    GROUP BY date
) TO '<s3_path>/results/results.csv';
```

This query runs in 3.7 minutes, so the overhead of handling schema evolution is only about 10%! That is a small price to pay for flexibility and ease of use.

However, we can improve performance further still. The next change was to increase the number of threads that DuckDB uses. By default, DuckDB will use a single thread per core. However, this is a very I/O intensive query (due to the network hops reading from then writing to S3) and less of a CPU intensive one. DuckDB uses synchronous I/O, so with the default thread count, if a thread is doing I/O, that CPU core is idle. As a result, using more threads might be more likely to fully utilize network resources, which is the bottleneck in this test. Here I just made an educated guess that this would help, but monitoring CPU utilization is a better approach.

With 4 threads, instead of the default of 2, performance improves to 3 minutes!

Adding more threads did not meaningfully improve performance any further. Additional threads do use more memory, but with the improvements in 1.1, this is no longer a significant issue (I tested up to 16 threads with only 2.2 GB of memory used).

The table below summarizes the results achieved on a `c5d.large` instance, which has 2 vCPUs and 4 GB RAM. We report the total runtime and the maximum memory usage for each query.

Query syntax	UNION type	Threads	Runtime	Memory
create view, copy	BY NAME	2	5.8 min	0.47 GB
create view, copy	BY POSITION	2	4.0 min	0.47 GB
create view, copy, new column	BY NAME	2	5.6 min	0.47 GB
copy subquery, new column	BY NAME	2	4.1 min	0.47 GB

Query syntax	UNION type	Threads	Runtime	Memory
copy subquery	BY POSITION	2	3.7 min	0.49 GB
copy subquery, new column	BY NAME	4	3.0 min	0.77 GB

Closing Thoughts

When stacking data, DuckDB brings the spirit of the relational model back to SQL! After all, stacking data should not require column orders to match... The `BY NAME` keywords can simplify common operations like combining relations with different orders or sets of columns, inserting the results of a query into a table, or querying a data lake with a changing schema. As of DuckDB version 1.1, this is now a performant and scalable approach!

Happy analyzing!

TPC-H on a Raspberry Pi

Publication date: 2025-01-17

Author: Gábor Szárnyas

TL;DR: DuckDB can run all TPC-H queries on a Raspberry Pi 5 board up to the 1,000 GiB dataset.

> **Update.** The setup described here can run all TPC-H queries on the SF100, SF300 and SF1,000 datasets with DuckDB v1.2.2 and newer versions. The timings given in this blog post were achieved in January 2025 using DuckDB v1.2-dev.

Introduction

The Raspberry Pi is an initiative to provide affordable and easy-to-program microcomputer boards. The initial model in 2012 had a single CPU core running at 0.7 GHz, 256 MB RAM and an SD card slot, making it a great educational platform. Over time the Raspberry Pi Foundation introduced more and more powerful models. The latest model, the Raspberry Pi 5, has a 2.4 GHz quad-core CPU and – with extra connectors – can even make use of NVMe SSDs for storage.

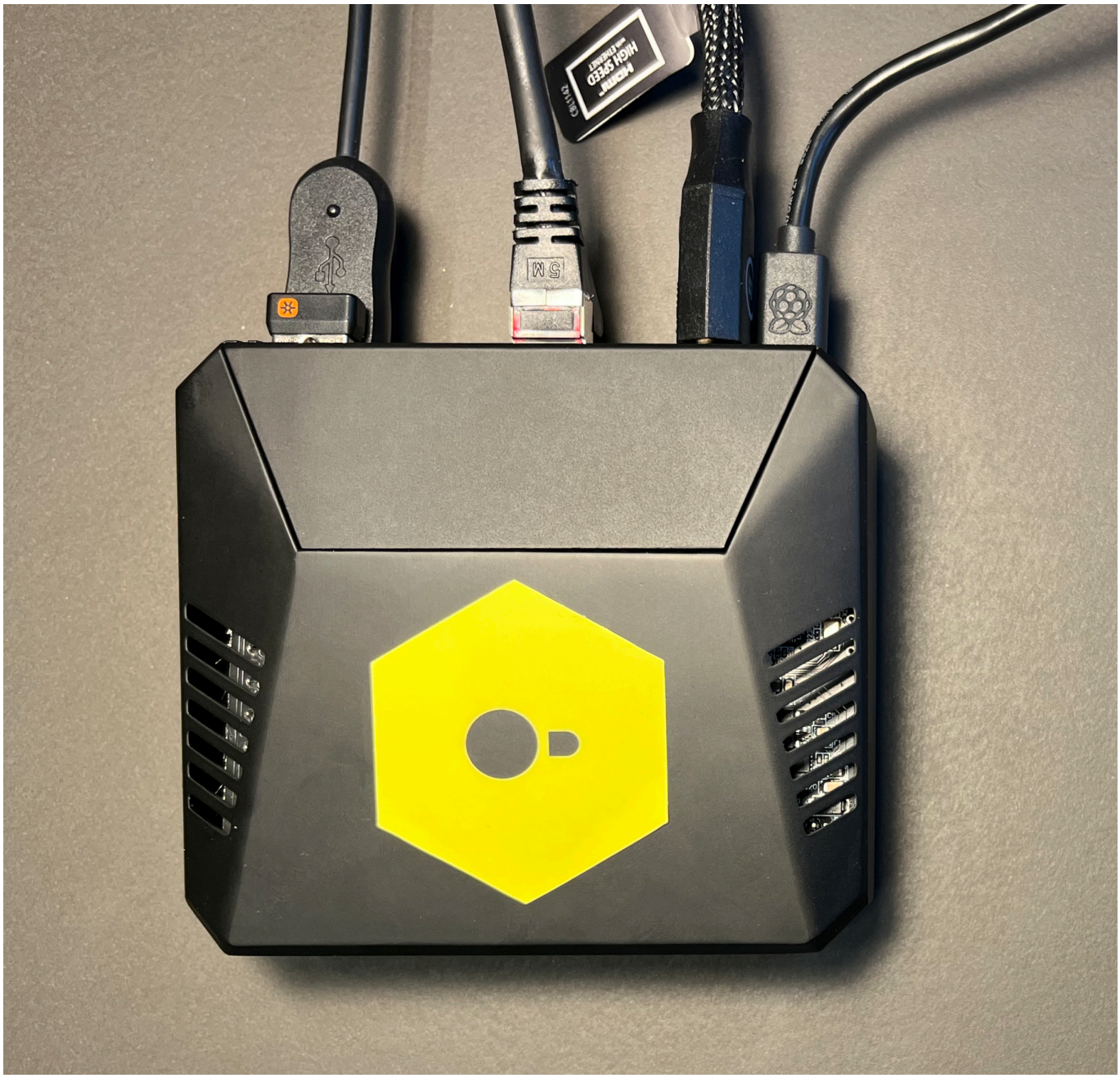
Last week, the Pi 5 got another upgrade: it can now be purchased with 16 GB RAM. The DuckDB team likes to experiment with DuckDB in unusual setups such as smartphones dipped into dry ice, so we were eager to get our hands on a new Raspberry Pi 5. After all, 16 GB or memory is equivalent to the amount found in the median gaming machine as reported in the 2024 December Steam survey. So, surely, the Pi must be able to handle a few dozen GBs of data, right? We ventured to find out.

Setup

Our setup consisted of the following components, priced at a total of $300:

Component	Price (USD)
Raspberry Pi 5 with 16 GB RAM	120.00
Raspberry Pi 27 W USB-C power supply	13.60
Raspberry Pi microSD card (128 GB)	33.40
Samsung 980 NVMe SSD (1 TB)	84.00
Argon ONE V3 Case	49.00
Total	**$300.00**

We installed the heat sinks, popped the SSD into place, and assembled the house. Here is a photo of our machine:



Experiments

So what is this little box capable of? We used the TPC-H workload to find out.

We first updated the Raspberry Pi OS (a fork of Debian Linux) to its latest version, 2024-11-19. We then compiled DuckDB version `0024e5d4be` using the Raspberry Pi build instructions. To make the queries easy to run, we also included the TPC-H extension in the build:

```
GEN=ninja BUILD_EXTENSIONS="tpch" make
```

We then downloaded DuckDB database files containing the TPC-H datasets at different scale factors (SF):

```
wget https://blobs.duckdb.org/data/tpch-sf100.db
wget https://blobs.duckdb.org/data/tpch-sf300.db
```

We used two different storage options for both scale factors. For the first run, we stored the database files on the microSD card, which is the storage that most Raspberry Pi setups have. This card works fine for serving the OS and programs, and it can also store DuckDB databases

but it's rather slow, especially for write-intensive operations, which include spilling to disk. Therefore, for the second run, we placed the database files on the 1 TB NVMe SSD disk.

For the measurements, we used our `tpch-queries.sql` script, which performs a cold run of all TPC-H queries, from Q1 to Q22.

We ran the script as follows:

```
duckdb tpch-sf${SF}.db -f tpch-queries.sql
```

We did not encounter any crashes, errors or incorrect results. The following table contains the aggregated runtimes:

Scale factor	Storage	Geometric mean runtime	Total runtime
SF100	microSD card	23.8 s	769.9 s
SF100	NVMe SSD	11.7 s	372.3 s
SF300	microSD card	171.9 s	4,866.5 s
SF300	NVMe SSD	55.2 s	1,561.8 s

Aggregated Runtimes

For the SF100 dataset, the geometric mean runtimes were 23.8 seconds with the microSD card and 11.7 seconds with the NVMe disk. The latter isn't that far off from the 7.5 seconds we achieved with the Samsung S24 Ultra, which has 8 CPU cores, most of which run at a higher frequency than the Raspberry Pi's cores.

For the SF300 dataset, DuckDB had to spill more to disk due to the limited system memory. This resulted in relatively slow queries for the microSD card setup with a geometric mean of 171.9 seconds. However, switching to the NVMe disk gave a 3× improvement, bringing the geometric mean down to 55.2 seconds.

Individual Query Runtimes

If you are interested in the individual query runtimes, you can find them below.

Perspective

To put our results into context, we looked at historical TPC-H results, and found that several enterprise solutions from 20 years ago had similar query performance, often reporting more than 60 seconds as the geometric mean of their query runtimes. Back then, these systems – with their software license and maintenance costs factored in – were priced *around $300,000!* This means that – if you ignore the maintenance aspects –, the "bang for your buck" metric (a.k.a. price–performance ratio) for TPC read queries has increased by around 1,000× over the last 20 years. This is a great demonstration of what the continuous innovation in hardware and software enables in modern systems.

Disclaimer: The results presented here are not official TPC-H results and only include the read queries of TPC-H.

Summary

We showed that you can use DuckDB in a Raspberry Pi setup that costs $300 and runs all queries on the TPC-H SF300 dataset in less than 30 minutes.

We hope you enjoyed this blog post. If you have an interesting DuckDB setup, don't forget to share it with us!

> Do you have a cool DuckDB setup? We would like to hear about it! Please post about it on social media or email it to Gábor Szárnyas.

Query Engines: Gatekeepers of the Parquet File Format

Publication date: 2025-01-22

Author: Laurens Kuiper

TL;DR: Mainstream query engines do not support reading newer Parquet encodings, forcing systems like DuckDB to default to writing older encodings, thereby sacrificing compression.

The Apache® Parquet™ Format

Apache Parquet is a popular, free, open-source, column-oriented data storage format. Whereas database systems typically load data from formats such as CSV and JSON into database tables before analyzing them, Parquet is designed to be efficiently queried directly. Parquet considers that users often only want to read some of the data, not all of it. To accommodate this, Parquet is designed to read individual columns instead of always having to read all of them. Furthermore, statistics can be used to filter out parts of files without fully reading them (this is also referred to as zone maps). Furthermore, storing data in Parquet typically results in a much smaller file than CSV or JSON due to a combination of lightweight columnar compression and general-purpose compression.

Many query engines implement reading and writing Parquet files. Therefore, it is also useful as a *data interchange* format. For example, Parquet files written by Spark in a large distributed data pipeline can later be analyzed using DuckDB. Because so many systems can read and write Parquet, it is the data format of choice for data lake solutions like Delta Lake™ and Iceberg™. While Parquet certainly has flaws, which researchers and companies are trying to address with new data formats, like it or not, it seems like Parquet is here to stay, at least for a while.

So, while we're here, we might try to make the best of it, right? SQL also has its flaws, and while researchers have certainly tried to create different query languages, we're still stuck with SQL. DuckDB embraces this and tries to make the best of it. The Parquet developers are doing the same for their format by updating it occasionally, bringing new features that make the format better.

Updates

If DuckDB adds a new compression method to its internal file format in a release, all subsequent releases must be able to read it. Otherwise, you couldn't read a database file created by DuckDB 1.1.0 after updating it to 1.2.0. This is called *backward compatibility*, and it can be challenging for developers. It sometimes requires holding onto legacy code and creating conversions from old to new. It is important to keep supporting older formats because updating DuckDB is much easier than rewriting entire database files.

Backward compatibility is also valuable for Parquet: it should be possible to read a Parquet file written years ago today. Luckily, most mainstream query engines can still read files in the Parquet 1.0 format, which was released in 2013, over ten years ago. Updates to the format do not threaten backward compatibility, as query engines simply need to continue being able to read the old files. However, it is also important that query engines add support for *reading* newer files alongside the older ones so that we can start *writing* new and improved Parquet files as well.

Here's where it gets tricky. We cannot expect query engines to be able to read the bleeding-edge Parquet format *tomorrow* if Parquet developers roll out an update *today*. We cannot start writing the new format for some time because many query engines will not be able to read it. The robustness principle states, “Be conservative in what you send, be liberal in what you accept.” If we apply this to Parquet files, query engines should strive to read new Parquet files but not write them yet, at least by default.

Encodings

DuckDB really likes lightweight compression. So, for the upcoming DuckDB 1.2.0 version, we're excited to have implemented the `DELTA_BINARY_PACKED`, `DELTA_LENGTH_BYTE_ARRAY` (added in Parquet 2.2.0 in 2015), and `BYTE_STREAM_SPLIT` (added in Parquet 2.8.0

in 2019) encodings in our Parquet writer. DuckDB, initially created in 2018, has been able to read Parquet since 2020, and has been able to read the encodings `DELTA_BINARY_PACKED` and `DELTA_LENGTH_BYTE_ARRAY` since 2022, and `BYTE_STREAM_SPLIT` since 2023.

However, despite these new encodings being available in 1.2.0, DuckDB will not write them by default. If DuckDB did this, many of our users would have a frustrating experience because some mainstream query engines still do not support reading these encodings. Having a good compression ratio does not help users if their downstream application cannot read the file. Therefore, we had to disable writing these encodings by default. They are only used when setting `PARQUET_VERSION v2` in a `COPY` command.

> DuckDB versions as old as 0.9.1 (released in late 2023) can already read files serialized with the setting `PARQUET_VERSION v2`.

Compressing data is almost always a trade-off between file size and the time it takes to write. Let's take a look at the following example (ran on a MacBook Pro with an M1 Max):

```
-- Generate TPC-H scale factor 1
INSTALL tpch;
LOAD tpch;
CALL dbgen(sf = 1);

-- Export to Parquet using Snappy compression
COPY lineitem TO 'snappy_v1.parquet'
    (COMPRESSION snappy, PARQUET_VERSION v1); -- 244 MB, ~0.46 s
COPY lineitem TO 'snappy_v2.parquet'
    (COMPRESSION snappy, PARQUET_VERSION v2); -- 170 MB, ~0.39 s

-- Export to Parquet using zstd compression
COPY lineitem TO 'zstd_v1.parquet'
    (COMPRESSION zstd, PARQUET_VERSION v1); -- 152 MB, ~0.58 s
COPY lineitem TO 'zstd_v2.parquet'
    (COMPRESSION zstd, PARQUET_VERSION v2); -- 135 MB, ~0.44 s
```

When using Snappy, DuckDB's default page compression algorithm for Parquet, which focuses mostly on speed, not compression ratio, the file is ~30% smaller and writing is ~15% faster with the encodings enabled. When using zstd, which focuses more on compression ratio than speed, the file is ~11% smaller, and writing is ~24% faster with the encodings enabled.

The compression ratio highly depends on how well data can be compressed. Here are some more extreme examples:

```
CREATE TABLE range AS FROM range(1e9::BIGINT);
COPY range TO 'v1.parquet' (PARQUET_VERSION v1); -- 3.7 GB, ~2.96 s
COPY range TO 'v2.parquet' (PARQUET_VERSION v2); -- 1.3 MB, ~1.68 s
```

The integer sequence 0, 1, 2, ... compresses extremely well with `DELTA_BINARY_PACKED`. In this case, the file is ~99% smaller, and writing is almost twice as fast.

Compressing floating points is much more difficult. Nonetheless, if there is a pattern, the data will compress quite well:

```
CREATE TABLE range AS SELECT range / 1e9 FROM range(1e9::BIGINT);
COPY range TO 'v1.parquet' (PARQUET_VERSION v1); -- 6.3 GB, ~3.83 s
COPY range TO 'v2.parquet' (PARQUET_VERSION v2); -- 610 MB, ~2.63 s
```

This sequence compresses really well with `BYTE_STREAM_SPLIT`. It is ~90% smaller and writes ~31% faster. Real-world data often does not have such extremely compressible patterns. Still, there are patterns, nonetheless, which will be exploited by these encodings.

If the query engines you're using support reading them, you can start using these encodings once DuckDB 1.2.0 is released!

Wasted Bits

Although it's difficult to get exact numbers, it's safe to assume that many TBs of data are written in Parquet each day. A large chunk of the bits written are wasted because query engines haven't implemented these newer encodings. The solution to this is surprisingly easy. There's no need to invent anything new to stop wasting all that space. Just read the specification on Parquet encodings, and implement them. Some of these “newer” encodings are almost 10 years old by now!

By reducing the size of Parquet files, we can reduce the amount of data we store in data centers. Reducing the amount of data we store even a little bit can have a big impact, as it can eventually reduce the need to build new data centers. This is not to say that data centers are evil; we will certainly need more of them in the future. However, making the most out of the data centers that we already have wouldn't hurt anyone.

Conclusion

Parquet is currently the industry standard tabular data format. Because it is also used as a data interchange format, the effectiveness of Parquet's features depends on the query engines that use it. If *some* of the mainstream query engines (you know who you are) refuse to implement these features, we *all* lose. This is not to say that all query engines must be on Parquet's bleeding edge, and DuckDB certainly isn't. However, query engine developers have a shared responsibility to make Parquet more useful.

We hope that more query engines will implement these newer encodings. Then, more query engines can write them by default and stop wasting so many bits.

Announcing DuckDB 1.2.0

Publication date: 2025-02-05

Author: The DuckDB team

TL;DR: The DuckDB team is happy to announce that today we're releasing DuckDB version 1.2.0, codenamed “Histrionicus”.

To install the new version, please visit the installation guide. For the release notes, see the release page.

> Some packages (Go, R, Java) take a few extra days to release due to the reviews required in the release pipelines.

We are proud to release DuckDB 1.2.0. This release is codenamed “Histrionicus” after the good-looking Harlequin duck (Histrionicus histrionicus), that inhabits “cold fast moving streams in North America, Greenland, Iceland and eastern Russia”.

What's New in 1.2.0

There have been far too many changes to discuss them each in detail, but we would like to highlight several particularly important and exciting features! Below is a summary of those new features with examples.

Breaking Changes

The `random` function now uses a larger state. This means that it's *even more random*™ now. Due to this change fixed seeds will now produce different values than in the previous versions of DuckDB.

`map['entry']` now returns a value, instead of a *list* of entries. For example, `map(['k'], ['v'])['k']` now returns `'v'`, while previously it returned `['v']`. We also introduced the `map_extract_value` function, which is now the alias for the bracket operator `[]`. If you would like to return a list, use the `map_extract` function: `map_extract(map(['k'], ['v']), 'k') = ['v']`.

The indexing of `list_reduce` is fixed. When indexing is applied in `list_reduce`, the index points to the last parameter of the lambda function and indexing starts from 1. Therefore, `list_reduce(['a', 'b'], lambda x, y, i: x || y || i)` returns ab2.

The `current_time` and `current_date` functions are now with respect to local timezone. This makes both functions compatible with PostgreSQL at the price of requiring ICU to be loaded to use these functions and their aliases `get_current_time()` and `today()`.

Explicit Storage Versions

DuckDB v1.2.0 ships new compression methods but *they are not yet enabled by default* to ensure that older DuckDB versions can read files produced by DuckDB v1.2.0.

In practice, this means that DuckDB v1.2.0 can read database files written by past stable DuckDB versions such as v1.0.0. When using DuckDB v1.2.0 with default settings, older versions can read files written by DuckDB v1.2.0.

You can *opt-in to newer forwards-incompatible features* using the following syntax:

```
ATTACH 'file.db' (STORAGE_VERSION 'v1.2.0');
```

This setting specifies the minimum DuckDB version that should be able to read the database file. When database files are written with this option, the resulting files cannot be opened by older DuckDB released versions than the specified version. They can be read by the specified version and all newer versions of DuckDB.

If you attach to DuckDB databases, you can query the storage versions using the following command:

```
SELECT database_name, tags FROM duckdb_databases();
```

This shows the storage versions:

database_name varchar	tags map(varchar, varchar)
file1 file2 ...	{storage_version=v1.2.0} {storage_version=v1.0.0 - v1.1.3} ...

This means that `file2` can be opened by past DuckDB versions while `file1` is compatible only with `v1.2.0` (or future versions).

To convert from the new format to the old format for compatibility, use the following sequence in DuckDB v1.2.0:

```
ATTACH 'file1.db';
ATTACH 'converted_file.db' (STORAGE_VERSION 'v1.0.0');
COPY FROM DATABASE file1 TO converted_file;
```

Indexing

`ALTER TABLE ... ADD PRIMARY KEY`. After a long while, DuckDB is finally able to add a primary key to an existing table . So it is now possible to run this:

```
CREATE TABLE tbl (id INTEGER);
INSERT INTO tbl VALUES (42);
ALTER TABLE tbl ADD PRIMARY KEY (id);
```

Over-eager constraint checking addressed. We also resolved a long-standing issue with over-eager unique constraint checking. For example, the following sequence of commands used to throw an error but now works:

```
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO students VALUES (1, 'John Doe');

BEGIN; -- start transaction
DELETE FROM students WHERE id = 1;
INSERT INTO students VALUES (1, 'Jane Doe');
```

CSV Features

Latin-1 and UTF-16 encodings. Previously, DuckDB's CSV reader was limited to UTF-8 files. It can now read Latin-1 and UTF-16 files. For example:

```
FROM read_csv('cities-latin-1.csv', encoding = 'latin-1');
```

Multi-byte delimiters. DuckDB now supports delimiters of up to 4 bytes. This means that you can finally use the duck emoji as a column delimiter. For example:

```
a🦆b
hello🦆world
```

```
FROM read_csv('example.dsv', sep = '🦆');
```

Strict CSV parsing. The RFC 4180 specification defines requirements for well-formed CSV files, e.g., having a single line delimiter. By default, DuckDB now parses CSVs in so-called strict mode (‘ strict_mode = true). For example, the following CSV file gets rejected because of mixed newline characters:

```
echo "a,b\r\nhello,42\nworld,84" > rfc_4180-defiant.csv
```

```
FROM read_csv('rfc_4180-defiant.csv');
```

```
Invalid Input Error:
Error when sniffing file "rfc_4180-defiant.csv".
It was not possible to automatically detect the CSV Parsing dialect/types
```

But it's parsed with the more lenient option `strict_mode = false`:

```
FROM read_csv('rfc_4180-defiant.csv', strict_mode = false);
```

a	b
varchar	int64
hello	42
world	84

Performance improvements. The CSV parser in the new release uses a new algorithm to find a new line on parallel execution. This leads to speedups of around 15%.

Unlimited row length. Previously, DuckDB was limited to CSV files with rows of up to 8 MB. The new version lifts this restriction, and lines can be of arbitrary length.

Parquet Features

Parquet dictionary and Bloom filter support. DuckDB now supports writing many more types using dictionary encoding. This should reduce file size in some cases. DuckDB is now also able to read and write Parquet Bloom filters. Bloom filters are small indexing data structures that can be used to exclude row groups if a filter is set. This is particularly useful for often-repeated but unordered data (e.g., categorical values). A separate blog post will follow.

Delta binary packed compression for Parquet. DuckDB now supports the `DELTA_BINARY_PACKED` compression as well as the `DELTA_LENGTH_BYTE_ARRAY` and `BYTE_STREAM_SPLIT` option for Parquet files. A few weeks ago, we elaborated on these in a blog post.

CLI Improvements

Safe mode. The DuckDB command line client now supports *safe mode*, which can be activated with the `-safe` flag or the `.safe_mode` dot command. In this mode, the CLI client is prevented from accessing external files other than the database file that it was initially connected to and prevented from interacting with the host file system. For more information, see the Securing DuckDB page in the Operations Manual.

Better autocomplete. The autocomplete in CLI now uses a Parsing Expression Grammar (PEG) for better autocomplete, as well as improved error messages and suggestions.

Pretty-printing large numbers. The CLI provides a summary of the number printed if the client is only rendering only a single row.

```
SELECT 100_000_000 AS x, pi() * 1e9 AS y;
```

x	y
int32	double
100000000 (100.00 million)	3141592653.589793 (3.14 billion)

Friendly SQL

Prefix aliases. SQL Expression and table aliases can now be specified before the thing they are referring to (instead of using the well-known syntax of using ASs). This can improve readability in some cases, for example:

```
SELECT
    e1: some_long_and_winding_expression,
    e2: t2.a_column_name
FROM
    t1: long_schema.some_long_table_name,
    t2: short_s.tbl;
```

Credit for this idea goes to Michael Toy. A separate blog post will follow soon. **Update:** the blog post on prefix aliases is out.

RENAME clause. DuckDB now supports the RENAME clause in SELECT. This allows renaming fields emitted by the * expression:

```
CREATE TABLE integers (col1 INTEGER, col2 INTEGER);
INSERT INTO integers VALUES (42, 84);
SELECT * RENAME (col1 AS new_col1) FROM integers;
```

Star LIKE. The LIKE and SIMILAR TO clauses can now be used on * expressions as a short-hand for the COLUMNS syntax.

```
CREATE TABLE key_val (key VARCHAR, val1 INTEGER, val2 INTEGER);
INSERT INTO key_val VALUES ('v', 42, 84);
SELECT * LIKE 'val%' FROM key_val;
```

val1	val2
int32	int32
42	84

Optimizations

We have spent a lot of time on DuckDB's optimizer. It is hard to quantify optimizer improvements, but as a result of these optimizations, DuckDB for example achieves a **13% improvement** on the total runtime of TPC-H SF100 queries when run on a MacBook Pro over the previous release.

C API for Extensions

Currently, DuckDB extensions use DuckDB's internal C++ structures. This – along with some fun linking issues – requires a lock-step development of extensions with mainline DuckDB and constant updates. Starting with this release, we expose a new C-style API for extensions in `duckdb_extension.h`. This API can be used to create for example scalar, aggregate or table functions in DuckDB. There are two main advantages of using this API: first, many programming languages (e.g., Go, Rust and even Java) have direct bindings to C APIs, making it rather easy to integrate. Secondly, the C Extension API is stable and backwards-compatible, meaning that extensions that target this API will keep working for new DuckDB versions. We will follow up with a new extension template.

musl Extensions

Distributing extensions for musl. The `musl` C library is often used in lightweight setups such as Docker setups running Alpine Linux. Starting with this release, we officially support musl and we distribute extensions for the `linux_amd64_musl` platform (but not yet for `linux_arm64_musl`). Note that DuckDB binaries (e.g., the CLI client) are not yet distributed for musl platforms, so you have to build them from source.

Zstd Compression

Columns can now be compressed with Zstd compression, which is suitable for bigger strings. By default, this compression is only considered for strings longer than 4096 characters, but this can be changed with the `zstd_min_string_length` setting.

Zstd compression can be enabled for a column explicitly when creating the table:

```
CREATE TABLE t (
    v VARCHAR USING COMPRESSION 'zstd';
);
```

Final Thoughts

These were a few highlights – but there are many more features and improvements in this release. There have been **over 5 000 commits** by over 70 contributors since we released 1.1.3. The full – very long – release notes can be found on GitHub.

We would like to thank again our amazing community for using DuckDB, building cool projects on DuckDB and improving DuckDB by providing us feedback. Your contributions truly mean a lot!

Catching up with Windowing

Publication date: 2025-02-10

Author: Richard Wesley

TL;DR: DuckDB implements a number of modern windowing features, some of which are extensions to the SQL standard. This posts presents a few of these features, including GROUPS framing, QUALIFY and aggregate/function modifiers.

Background

In the beginning, the relational data processing model was all about sets. This was Codd's great insight and for many years, relational processing took little notice of data ordering.

But it turns out that there are a lot of analytic operations that are related to ordering. For example, smoothing out noise in time series is very difficult to do in traditional SQL queries – it involves self-joins with inequality conditions! So in the late 1990s database vendors started adding *windowing* operations. By making the user's intent clear, the operations could be implemented much more efficiently, and these operations were eventually added to the SQL:2003 standard.

DuckDB has had support for window functions since the early days, but if they are new to you, you might want to start with my earlier blog posts on Windowing in DuckDB and Fast Moving Holistic Aggregates, or just the window function documentation. In this post, I will start by introducing the more recent functionality additions. In a follow-up post, I will spelunk into the internals to talk about some performance and scaling improvements.

For the examples in this post, I will mostly stick to using a table of athletic `results`:

Field	Type	Description
`event`	`VARCHAR`	The name of the event (e.g., 200 meter butterfly).
`athlete`	`VARCHAR`	The name of the competitor (e.g., Michael Phelps).
`date`	`TIMESTAMP`	The start time of the event.
`time`	`DECIMAL(18, 3)`	The athlete's time in that event (in seconds).

GROUPS Framing

In addition to the ROWS and RANGE frame boundary types (which we have supported for a while now), the standard also defines GROUPS as a boundary type. ROWS is pretty simple: it just counts the number of rows. RANGE is trickier: it treats its counts as distances from the value of the ORDER BY expression at the current row. This means that there can only be one such expression and you have to be able to do arithmetic on it.

GROUPS is somewhere in between. A “group” in the standard's language is all the “peers” of a row, which are all the rows with the same value of the ORDER BY expression at the current row. In the original windowing code, this was not easy to implement, but after several years of work, the infrastructure has evolved, and as of v1.2.0 we now support this last type of framing.

Frame Exclusion

Another missing piece of the 2003 specification was the EXCLUDE clause. Thanks to work by a community member, we have supported this since v0.10.0, but we somehow never got around to mentioning it in a blog post!

EXCLUDE is an optional modifier to the frame clause for excluding rows around the CURRENT ROW. This is useful when you want to compute some aggregate value of nearby rows to see how the current row compares to it. In this example, we want to know how an athlete's time in an event compares to the average of all the times recorded for their event within ±10 days:

```
SELECT
    event,
    date,
    athlete,
    avg(time) OVER w AS recent,
FROM results
WINDOW w AS (
    PARTITION BY event
    ORDER BY date
    RANGE BETWEEN INTERVAL 10 DAYS PRECEDING AND INTERVAL 10 DAYS FOLLOWING
        EXCLUDE CURRENT ROW
)
ORDER BY event, date, athlete;
```

There are four options for EXCLUDE that specify how to treat the current row:

- CURRENT ROW – exclude just the current row
- GROUP – exclude the current row and all its "peers" (rows that have the same ORDER BY value)
- TIES – exclude all peer rows, but *not* the current row (this makes a hole on either side)
- NO OTHERS – don't exclude anything (the default)

Exclusion is implemented for both windowed aggregates and for the `first`, `last` and `nth_value` functions.

QUALIFY Clause

It may not be immediately obvious, but the SQL language has rules for the order in which various expressions are computed. For example, aggregates (like sum) are computed after row-level expressions (like +). This is why SQL has two filtering clauses: WHERE and HAVING: WHERE is for row-level computations and HAVING is applied after GROUP BY.

When windowing was introduced, it added another layer of computation: window functions are computed *after* aggregates. That is great, but then how do you filter the results of an OVER function? Originally, you had to put the query in a Common Table Expression (or CTE) which is defined by a WITH clause:

```
-- Find the third fastest times in each event
WITH windowed AS (
    SELECT
        event,
        athlete,
        time,
        row_number() OVER w AS r
    FROM results
    WINDOW w AS (
        PARTITION BY event
        ORDER BY time
    )
)
SELECT event, athlete, time
FROM windowed
WHERE r = 3;
```

This was kind of clunky, so eventually the QUALIFY clause was proposed for filtering window functions. DuckDB supports this, making it easier to filter the results of window functions:

```
-- Find the third fastest times in each event
SELECT event, athlete, time
FROM results
WINDOW w AS (
    PARTITION BY event
```

```
    ORDER BY time
)
QUALIFY row_number() OVER w = 3;
```

Aggregate Modifiers

There are several modifiers for ordinary aggregate functions (`FILTER`, `DISTINCT` and `ORDER BY` as an argument) that are not part of the SQL:2003 standard for windowing, but which are also useful in a windowing context. `FILTER` is pretty straightforward (and DuckDB has supported it for a while) but the others are not easy to implement efficiently.

They can of course be implemented naïvely (academic-speak for "slow"!) by just computing each row independently:

- re-read all the values,
- filter out the ones we don't want,
- stick them into a hash table to remove duplicates,
- sort the results,
- send them off to the aggregate function to get the result.

We have an implementation that does this (which you can access by turning off the optimizer) and we use it to check fancier implementations, but it is horribly slow.

Fortunately, these last two modifiers have been the subject of research published in the last 10 years, and we have now added those algorithms to the windowing aggregates. We can then use the `DISTINCT` modifier to exclude duplicates in the frame:

```
-- Count the number of distinct athletes at a given point in time
SELECT count(DISTINCT athlete) OVER (ORDER BY date) FROM results;
-- Concatenate those distinct athletes into a list
SELECT list(DISTINCT athlete) OVER (ORDER BY date) FROM results;
```

We can also use the `ORDER BY` modifier with order-sensitive aggregates to get sorted results:

```
-- Return an alphabetized list of athletes who made or beat a time
SELECT list(athlete ORDER BY athlete) OVER (
    PARTITION BY event, date
    ORDER BY time DESC
)
FROM results;
```

I should mention that the research on these extensions is ongoing, and combining them will often force us to use the naïve implementation. So for example, if we wished to exclude the athlete who made the time in the previous example:

```
-- Return an alphabetized list athletes who beat the each time
SELECT list(athlete ORDER BY athlete) OVER (
    PARTITION BY event, date
    ORDER BY time DESC
    EXCLUDE CURRENT ROW
)
FROM results;
```

DuckDB will still compute this for you, but it may be very slow.

Function Modifiers

The `ORDER BY` modifier also makes sense for some non-aggregate window functions, especially if we let them use framing with it:

```
-- Compute the current world record holder over time for each event
SELECT
    event,
    date,
    first_value(time ORDER BY time DESC) OVER w AS record_time,
    first_value(athlete ORDER BY time DESC) OVER w AS record_holder,
```

```
FROM results
WINDOW w AS (
    PARTITION BY event
    ORDER BY date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
)
ORDER BY event, date;
```

All of the non-aggregate window functions (except `dense_rank`) now support ordering arguments and will use the frame instead of the entire partition when an ordering argument is supplied.

> **Tip.** If you wish to use the entire frame with an ordering argument, then you will need to be explicit and use `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`.

> **Note.** If you wish to use the frame ordering *and* the frame boundaries with a non-aggregate function, you will need to specify the `ORDER BY` *twice* (once in the frame specification and once in the argument list). This has not yet been optimized, but it will be in the v1.3.0 release.

Conclusion

Windowing is a very natural way to think about order-dependent analysis, but it is at odds with traditional unordered query processing. Nevertheless, since 2003 the SQL language has provided syntax for expressing a wide range of such queries. In recent years, the community has also considered further extensions to the language (such as `QUALIFY` and argument modifiers like `DISTINCT` and `ORDER BY`) to improve expressivity. Here at DuckDB we love providing this kind of expressiveness as part of our "friendly SQL" work. What may be less obvious is that when we enable users to express their problem more naturally, it helps us provide more performant solutions! In my next post, I will go deeper into recent improvements in windowing's performance and resource utilization.

Flying Through Windows

Publication date: 2025-02-14

Author: Richard Wesley

TL;DR: Dive into the details of recent DuckDB windowing performance improvements.

Introduction

In the previous post I went into some new windowing functionality in DuckDB available through SQL. But there are other changes that improve our use of resources (such as memory) without adding new functionality. So let's get “under the feathers” and look at these changes.

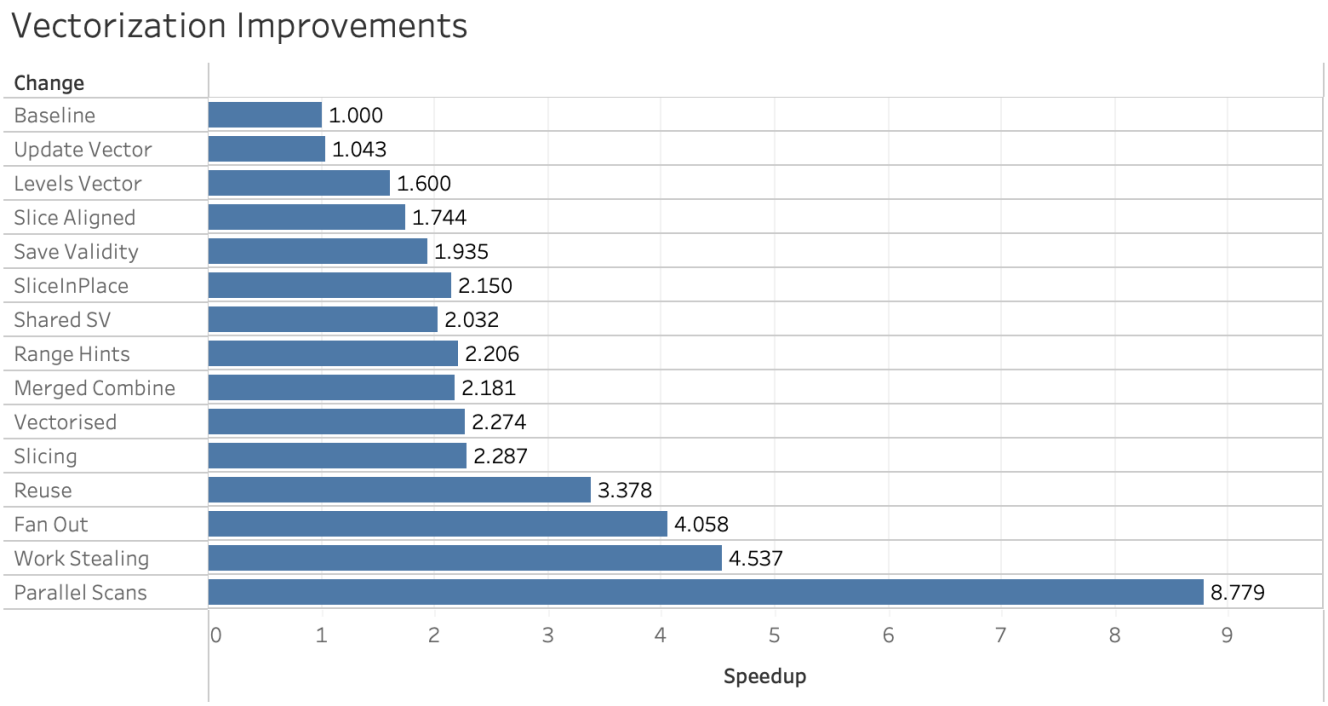
> We previously benchmarked ourselves on a window function-heavy workload, which showed great performance improvements over time. The optimizations presented in this blog post push the performance of DuckDB's window operator even further.

Segment Tree Vectorization

One important improvement that was made in the summer of 2023 was converting the segment tree evaluation code to vectorized evaluation from single-value evaluation. You might wonder why it wasn't that way to begin with in a “vectorized relational database” (!), but the answer is lost in the mists of time. My best guess is either that the published algorithm was written for values or that the aggregation API had not been nailed down yet (or both).

In the old version, we used the aggregate's `update` or `combine` APIs, but with only the values and tree states for a single row. To vectorize the segment tree aggregation, we accumulate *vectors* of leaf values and tree states and flush them into each output row's state when we reach the vector capacity of 2048 rows. Some care needed to be taken to handle order-sensitive aggregates by accumulating values in the correct order. `FILTER` and `EXCLUDE` clauses also provided some entertainment, but the segment trees are now fully vectorized. The performance gains here were about a factor of four (from “Baseline” to “Fan Out”).

The chart below shows the speedups achieved by the vectorization improvements:



Once segment trees were vectorized, we could use the same approach when implementing `DISTINCT` aggregates with merge sort trees. It may be worth updating the custom window API to handle vectorization at some point, because although most custom window aggregates are quite slow (e.g., `quantile`, `mad` and `mode`), `count(*)` is also implemented as a custom aggregate and would likely benefit from a vectorized implementation.

Constant Aggregation

A lot of window computations are aggregates over frames, and a common analytic task with these results is to compare a partial aggregate to the same aggregate over *the entire partition*. Computing this value repeatedly is expensive and potentially wasteful of memory (e.g, the old implementation would construct a segment tree even though only one value was needed.)

The previous performance workaround for this was to compute the aggregate in a subquery and join it in on the partition keys, but that was, well, unfriendly. Instead, we have added an optimization that checks for *partition-wide aggregates* and computes that value once per partition. This not only reduces memory and compute time for the aggregate itself, but we can often return a constant vector that shares the values across all rows in a chunk, reducing copy costs and potentially even downstream evaluation costs.

Returning a constant vector can yield surprisingly large memory and performance benefits. In the issue that drove this improvement, the user was constructing a constant 100k element list (!) and then computing the median with a list aggregation lambda. By returning a single constant list, we build and reduce that list only once instead of once per row!

Streaming Windows

Computing window functions is usually quite expensive! The entire relation has to be materialized, broken up into partitions, and each partition needs to be sorted.

But what if there is no partitioning or ordering? This just means that the window function is computed over the entire relation, in the "natural order", using a frame that starts with the first row and continues to the current row. Examples might be assigning row numbers or computing a running sum. This is simple enough that we can *stream* the evaluation of the function on a single thread.

First, let's step back a bit and talk about the window *operator*. During parsing and optimization of a query, all the window functions are attached to a single *logical* window operator. When it comes time to plan the query, we group the functions that have common partitions

and "compatible" orderings (see Cao et al., *Optimization of Analytic Window Functions* for more information) and hand each group off to a separate *physical* window operator that handles that partitioning and ordering. In order to use the "natural order" we have to group those functions that can be streamed and execute them first (or the order will have been destroyed!) and hand them off to the *streaming* physical window operator.

So which window functions can we stream? It turns out there are quite a few:

- Aggregates `BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` (we just update the aggregate)
- `first_value` – it's always the same
- `percent rank` – it's always 0
- `rank` – it's always 0
- `dense_rank` – it's always 0
- `row_number` – we just count the rows
- `lead` and `lag` – we just keep a buffer

There are a few more restrictions:

- `IGNORE NULLS`, `EXCLUDE` and `ORDER BY` arguments are not allowed
- `lead` and `lag` distances are restricted to a constant within ±2048 (one vector). this is not really a big deal because the distance is usually 1.

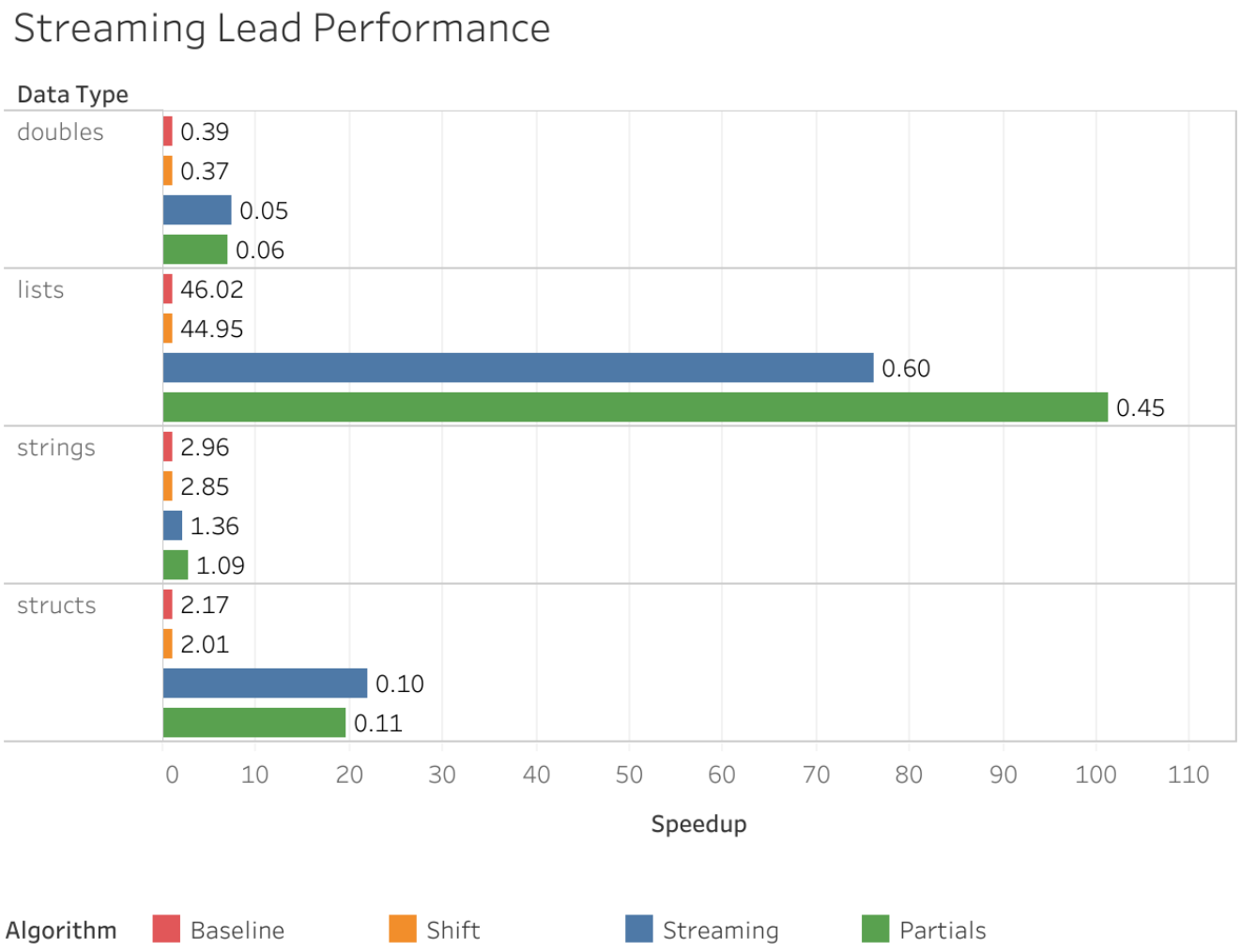
The improvements for streaming `LEAD` were quite dramatic:

```
SELECT setseed(0.8675309);
CREATE OR REPLACE TABLE df AS
    SELECT
        random() AS a,
        random() AS b,
        random() AS c,
    FROM range(10_000_000);

SELECT sum(a_1 + a_2 + b_1 + b_2)
FROM (
    SELECT
        lead(a, 1) OVER () AS a_1,
        lead(a, 2) OVER () AS a_2,
        lead(b, 1) OVER () AS b_1,
        lead(b, 2) OVER () AS b_2
    FROM df
) t;
```

The chart below shows the performance improvements.

- Baseline – Where we started
- Shift – Copying blocks of rows inside the non-streaming implementation, instead of one at a time
- Streaming – First implementation of streaming `LEAD`
- Partials – Avoid copying when the entire chunk can just be referenced

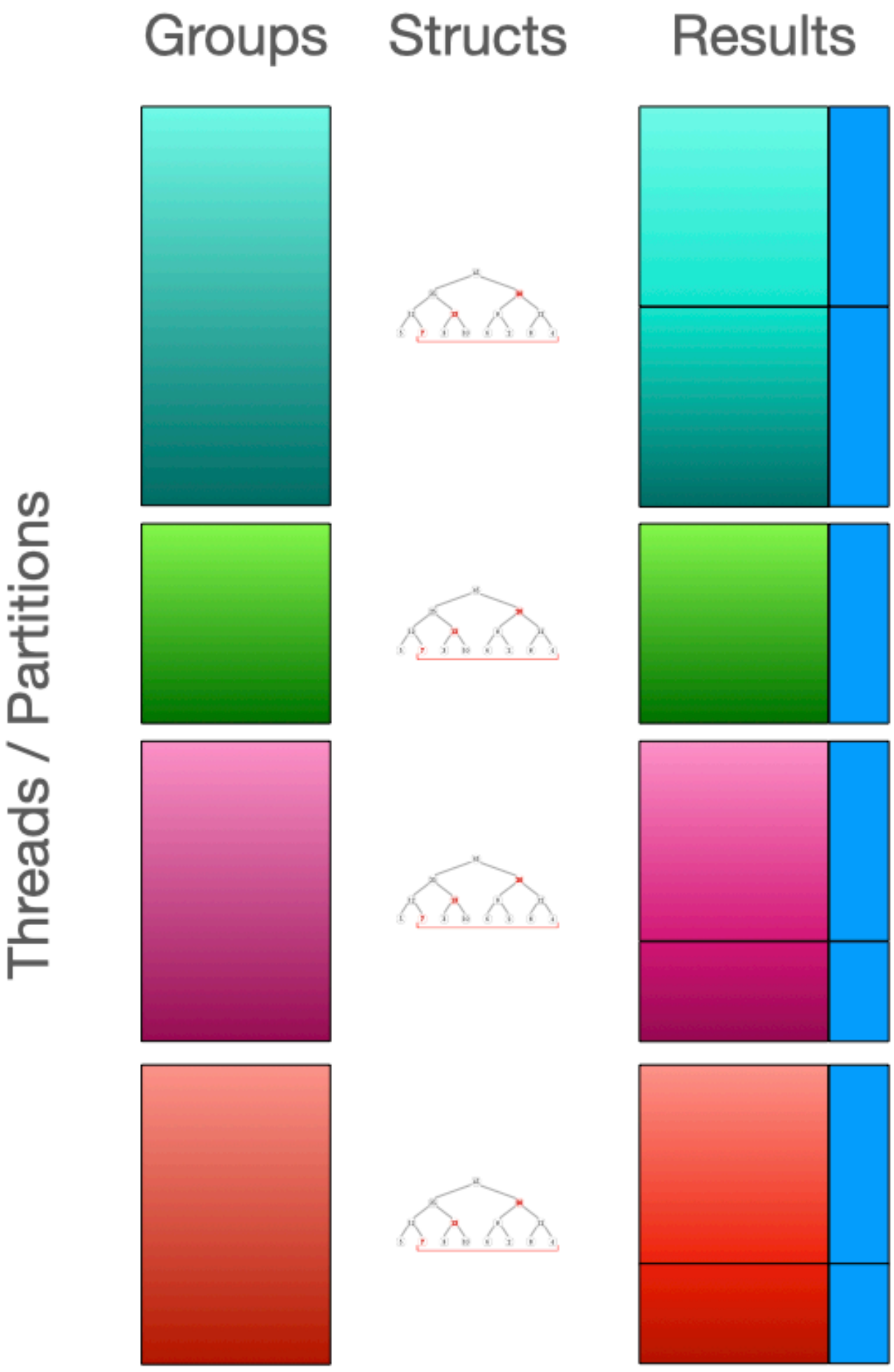
Note that the *x* axis shows the speedups compared to the baseline (1×) and the absolute runtimes (in seconds) are shown as labels.



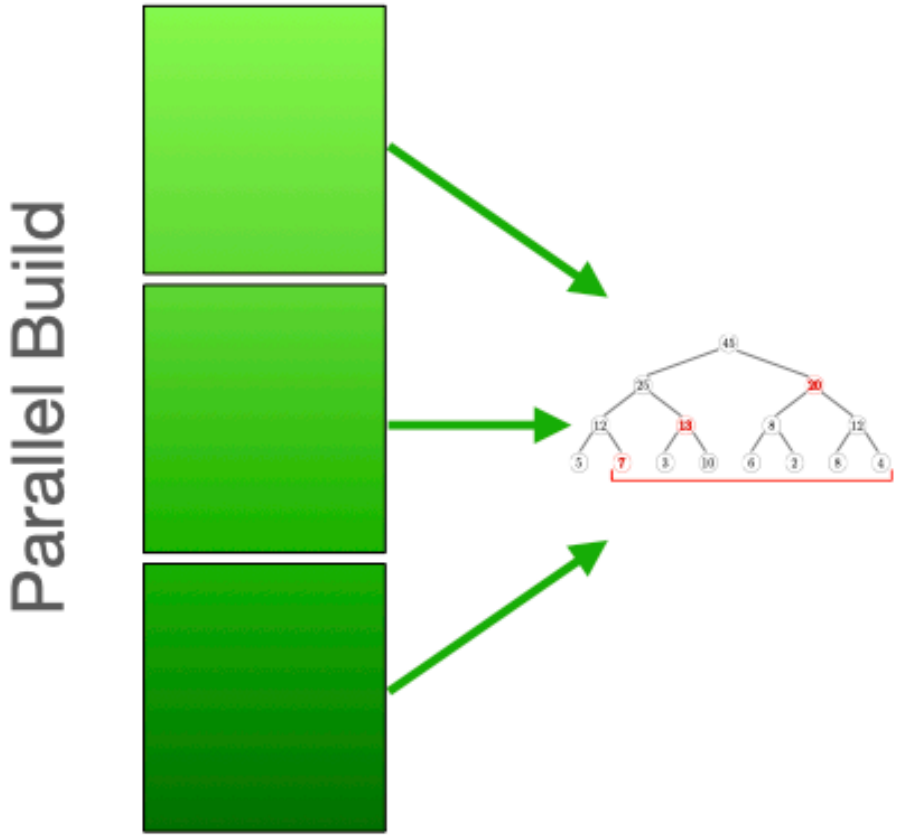
In the future we may be able to relax the end of the frame to a constant distance from the current row that fits inside the buffer length (e.g., `BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING`) but that hasn't been investigated yet.

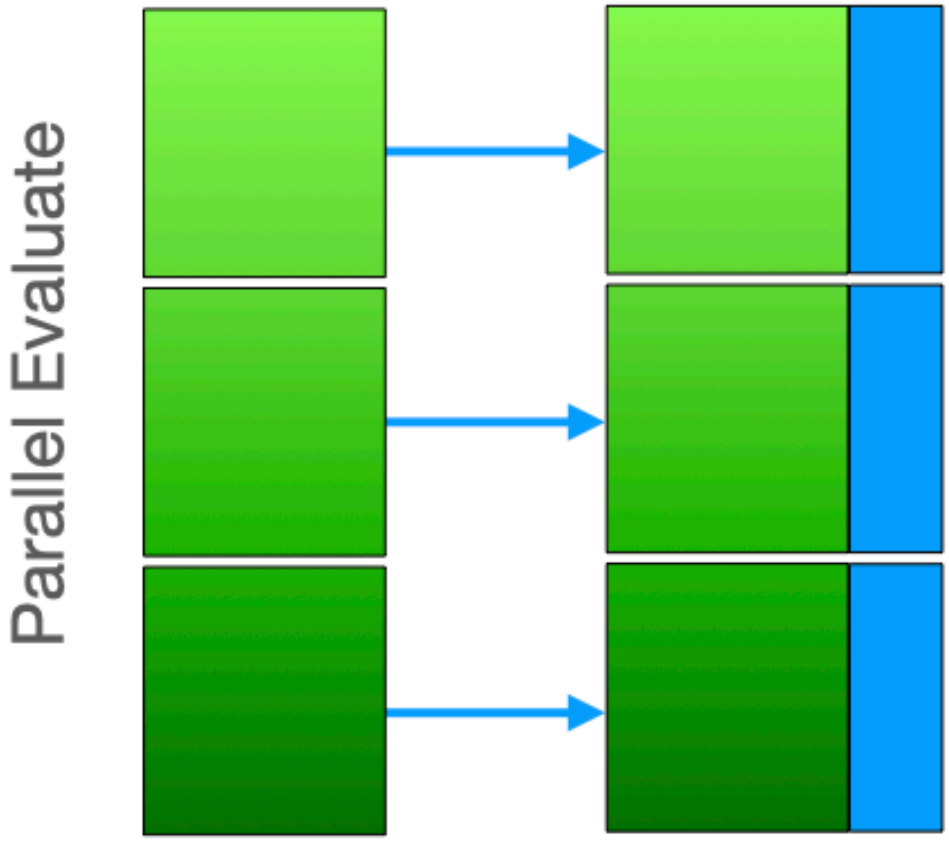
Partition Major Evaluation

Window partitions are completely independent, so evaluating them separately is attractive. Our first implementation took advantage of this and evaluated each partition on a separate thread. This works well if you have more partitions than threads, and they are all roughly the same size, but if the partition sizes are skewed or if there is only one partition (a common situation), then most of the cores will be idle, reducing throughput.


Groups
Structs
Results
Threads / Partitions

To improve the CPU utilization, we changed the execution model for v1.1 to evaluate partitions in parallel. The partitions are evaluated from largest to smallest and we then distribute each partition across as many cores as we can, while synchronizing access to shared data structures. This was a lot more challenging than independent single-threaded evaluation of partitions, and we had some synchronization issues (hopefully all sorted now!) that were dealt with in the v1.1.x releases. But we now have much better core utilization, especially for unpartitioned data. As a side benefit, we were able to reduce the memory footprint because fewer partitions were in memory at a time.


Parallel Build


Parallel Evaluate

One remaining issue is the coarseness of the sub-partitions. At the moment to avoid copying they use the blocks produced by the sorting code, which are often larger than we would like. Reducing the size of these chunks is future work, but hopefully we will get to it as part of some proposed changes to the sorting code.

Out of Memory Operation

Because windowing materializes the entire relation, it was very easy to blow out the memory budget of a query. For v1.2 we have switched from materializing active partitions in memory to using a pageable collection. So now not only do we have fewer active partitions (due to partition major evaluation above), but those partitions themselves can now spool to disk. This further reduces memory pressure during evaluation of large partitions.

Windowing is so complex, however, that there are still some remaining large data structures. The segment trees and merge sort trees used for accelerating aggregation are still in memory, especially the intermediate aggregate states in the middle of the trees. Solving this completely, will require a general method of serializing aggregate states to disk, which we do not yet have. Still, most aggregates can be serialized as binary data without special handling, so in the short term we can probably cover a lot of cases with the current aggregation infrastructure, just as we do for GROUP BY.

Shared Expressions

Window expressions are evaluated independently, but they often share expressions. Some of those expressions can be expensive to evaluate and others need to be materialized over the entire partition. As an example of the latter, aggregate functions can reference values anywhere in the partition. This could result in computing and materializing the same values multiple times:

```
-- Compute the moving average and range of x over a large window
SELECT
    x,
    min(x) OVER w AS min_x,
    avg(x) OVER w AS avg_x,
    max(x) OVER w AS max_x,
FROM data
WINDOW w AS (
    PARTITION BY p
    ORDER BY s
    ROWS BETWEEN 1_000_000 PRECEDING and 1_000_000 FOLLOWING
);
```

Paging the data for `x` reduces the memory footprint, but the segment trees used to evaluate the three aggregates will contain duplicate copies of `x` – along with the with operator itself (which has to return `x`). With v1.2 we have added a mechanism for sharing evaluation of expressions like these between functions. This not only reduces memory, but in this example we will also reduce disk paging because all three functions will be accessing the same values.

There are a number of places where we are sharing expressions, including ORDER BY arguments, `range` expressions and "value" functions like `lead`, `lag` and `nth_value`, and we are always on the lookout for more (such as frame boundaries – or even segment trees).

Future Work

While I have mentioned a number of things we would like to get to in the future, one that doesn't fit nicely into any of the topics so far is query rewriting. It turns out that some window functions can be evaluated using other techniques such as self-joins and some of our smarter aggregates (like `arg_max`). Generating these alternate query plans can have large performance benefits and we plan to investigate them.

Conclusion

As you can see, windowing is a big hairy beast! There is also not a lot of published research on effective algorithms (I've linked to pretty much all of it), so we are often stuck making it up ourselves or falling back to extremely simple and slow approaches. But I hope that many

of you will find something new and exciting in what we have been up to for the past 2-3 years – and I will try to be more timely in blogging about future window improvements.

Planning AsOf Joins

Publication date: 2025-02-19

Author: Richard Wesley

TL;DR: AsOf Joins are a great example of how DuckDB can choose different implementations for an expensive operation.

> Quote I love it when a plan comes together.
>
> -- Hannibal Smith, *The A-Team*

Introduction

AsOf joins are a very useful kind of operation for temporal analytics. As the name suggests, they are a kind of lookup for when you have a table of values that change over time, and you want to look up the most recent value at another set of times. Put another way, they let you ask *"What was the value of the property **as of this time**?"*

DuckDB added AsOf joins about 18 months ago and you can read that post to learn about their semantics.

What's the Plan?

In that earlier post, I explained why we have a custom operator and syntax for AsOf joins when you can implement them in traditional SQL. The superpower of SQL is that it is *declarative:* you tell us *what* you want and we figure out an efficient *how.* By allowing you to say you want an AsOf join, we can think about how to get you results faster!

Still, the AsOf operator has to do a lot of work. Namely:

- Read all the data in the right side (lookup) table
- Partition it on any equality conditions
- Sort it on the inequality condition
- Repeat the process for the left side (probe) table
- Do a merge join on the two tables that only returns the "most recent" value

That's a lot of data movement! Plus, if any of the tables are large, we may end up exceeding memory and spilling to disk, slowing the operation down even further. Still, as we will see, it is much faster than the plain SQL implementation.

This is such a burden, that many databases that support AsOf joins require the right side table to be partitioned and ordered on any keys you might want to join on. That doesn't fit well with DuckDB's "friendly SQL" approach, so (for now) we have to do it every time.

Let's Get Small

It turns out there is a very common case for AsOf where the left table is small. Suppose you have a year's worth of price data, recorded at high granularity (e.g., fractions of a second), but you only want to look up a small number (say 20) values that are the times you actually bought or sold?

The price table could run to hundreds of millions, if not billions of rows, and just sorting that will take a lot of time and memory. Given how expensive that is, one might wonder if there is a way to avoid all that sorting? Happily the answer is *yes!*

Simple Joins

Suppose we were to swap the sides of the join, build the old left side as a small right side table, and stream the huge table through the left side of the join. We could use the AsOf conditions for the join, and hopefully find a way to throw out the older matches (we only want to keep the latest match). This would use very little memory, and the streaming could be highly parallelized.

There are two streaming physical join operators we could use for this:

- *Nested Loop Join* – Literally what it sounds like: loop over each left block and the right side table, checking for matches;
- *Piecewise Merge Join* – A tricky join for one inequality condition that sorts the right side and each left block before merging to find matches;

We can try both of these once we have a way to eliminate the duplicates. One thing to be aware of, though, is that they are both `N^2` algorithms, so there will be a limit on how big "small" can be.

Grouping

If you have been around databases long enough, you know that the phrase "eliminate the duplicates" means GROUP BY! So to eliminate the duplicates, we want to add an aggregation operator onto the output. The tricky part is that we want to keep only the matched values that have the "largest" times. Fortunately, DuckDB has a pair of aggregate functions that do just that: `arg_max` and `arg_min` (also called `max_by` and `min_by`).

That takes care of the fields from the lookup table, but what about the fields from the small table? Well, those values will all be the same, so we can just use the `first` aggregate function for them.

Streaming Window

But what should we group on? One might be tempted to group on the times that are being looked up, but that could be problematic if there are duplicate lookup times (only one of the rows would be returned!). Instead, we need to have a unique identifier for each row being looked up. The simplest way to do this is to use the *streaming window operator* with the `row_number()` window function. We then group on this row number.

Coming Together

This all sounds good, but how does it work in practice? How big can "small" get? To answer that I ran a number of benchmarks joining small tables against large ones. The tables are called `prices` and `times`:

```
CREATE OR REPLACE TABLE prices_{prices_size} AS
    SELECT
        r AS id,
        '2021-01-01T00:00:00'::TIMESTAMP +
            INTERVAL (random() * 365 * 24 * 60 * 60) SECOND
            AS time,
        (random() * 100000)::INTEGER AS price,
    FROM range({prices_size}) tbl(r);

CREATE OR REPLACE TABLE times_{times_size} AS
    SELECT
        r AS id,
        '2021-01-01'::TIMESTAMP +
            INTERVAL ((random() * 365 * 24 * 60 * 60)::INTEGER) SECONDS
            AS probe
    FROM range({times_size}) tbl(r);
```

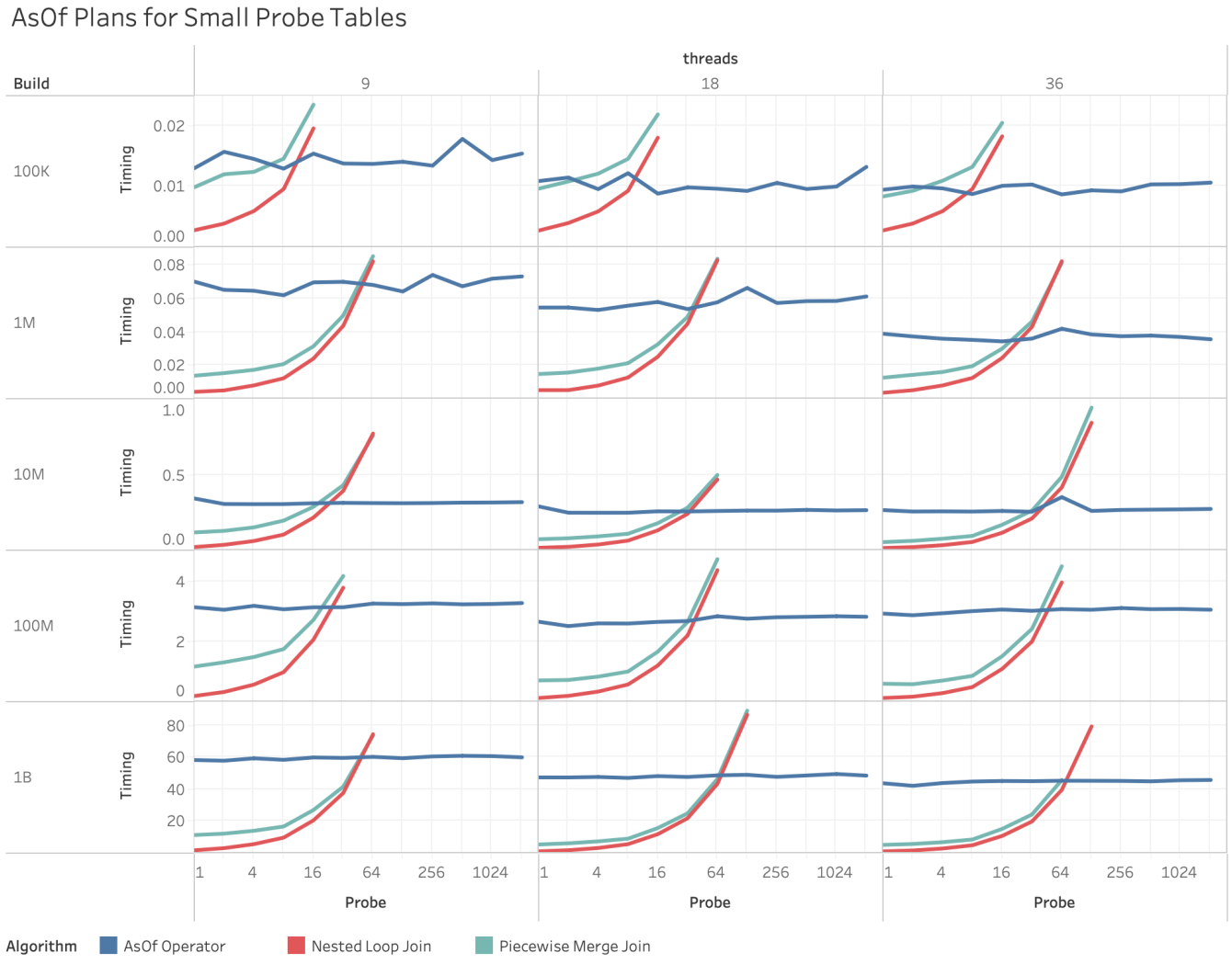
I then ran a benchmark query:

```
SELECT count(*)
FROM (
    SELECT
        t.probe,
        p.price
    FROM times_{times_size} t
        ASOF JOIN prices_{prices_size} p
        ON t.probe >= p.time
    ) t;
```

for a matrix of the following values:

- Prices – 100K to 1B rows in steps of 10×;
- Time – 1 to 2048 rows in steps of 2× (until it got too slow);
- Threads – 36, 18 and 9;

Here are the results:



As you can see, the quadratic nature of the joins means that "small" means "<= 64". That is pretty small, but the table in the original user issue had only 21 values.

We can also see that the sorting provided by piecewise merge join does not seem to help much, so plain old Nested Loop Join is the best choice.

It is clear that the performance of the standard operator is stable at each size, but decreases slowly as the number of threads increases. This makes sense because sorting is compute-intensive and the fewer cores we can assign, the longer it will take.

If you want to play with the data more, you can find the interactive vizualization on our Tableau Public site.

Memory

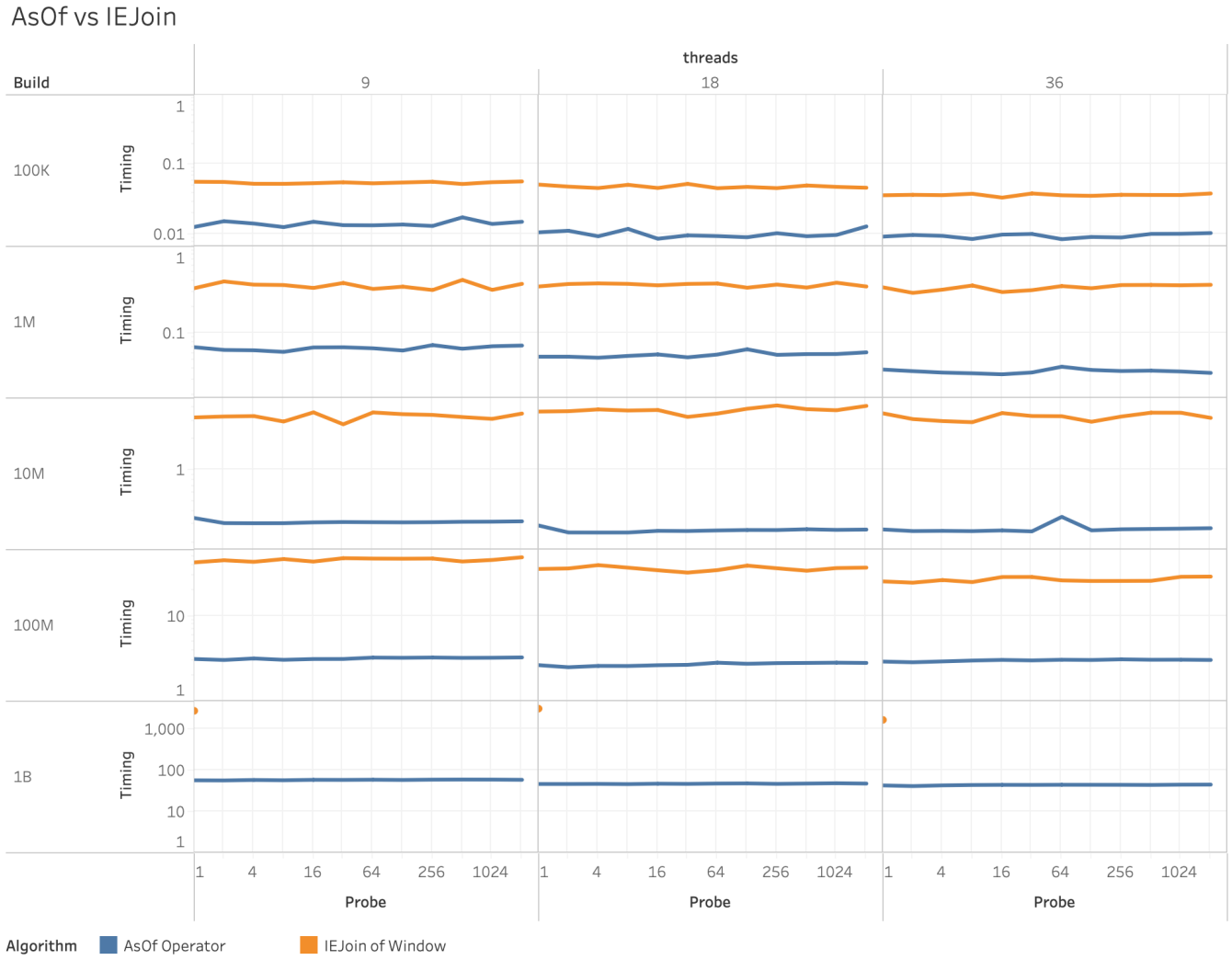
The Loop Join plan is clearly faster at small sizes, but how much memory do the two plans use? These are the rough amounts of memory needed before excessive paging or allocation failures occur:

Price Rows	AsOf Memory	Loop Join Memory
1B	48 GB	64 MB
100M	6 GB	64 MB
10M	256 MB	64 MB
1M	32 MB	64 MB
100K	32 MB	64 MB

In other words, the Loop Join plan only needs enough memory to page in the lookup table! So if the table is large and you have limited memory, the Loop Join plan is the best option, even if it is painfully slow. Just remember that the Loop Join plan has to compete with the speed of the standard operator under paging, and that may still be faster past a certain point.

Backup Plans

As part of the experiment, I also measured how the old SQL implementation would perform, and it did not fare well. At the 1B row level I had to cut it off after one run to avoid wasting time:



Note that the Y-axis is a log scale here!

Setting

While it is nice that we provide a default value for making plan choices like this, your mileage may vary as they say. If you have more time than memory, it might be worth it to you to bump up the Loop Join threshold a bit. The threshold is a new setting called `asof_loop_join_threshold` with a default value of 64, and you can change it using a PRAGMA statement:

```
PRAGMA asof_loop_join_threshold = 128;
```

Remember, though, this is a quadratic operation, and pushing it up too high might take a Very Long Time (especially if you express it in Old Entish!).

If you wish to disable the feature, you can just set it to zero:

```
PRAGMA asof_loop_join_threshold = 0;
```

Roll Your Own

This Loop Join plan optimization will not ship until v1.3, but if you are having problems today, you can always write your own version like this:

```
SELECT
    first(t.probe) AS probe,
    arg_max(p.price, p.time) AS price
FROM prices p
INNER JOIN (
    SELECT
        *,
        row_number() OVER () AS pk
    FROM times
) t
ON t.probe >= p.time
GROUP BY pk
ORDER BY 1;
```

If you know the probe times are unique, you can simplify this to:

```
SELECT
    t.probe,
    arg_max(p.price, p.time) AS price
FROM prices p
INNER JOIN times t
ON t.probe >= p.time
GROUP BY 1
ORDER BY 1;
```

Future Work

The new AsOf Loop Join plan feature only covers a common but very specific situation, and the standard operator could be made a lot more efficient if it knew that the data was already sorted. This is often the case, but we do not yet have the ability to track partitioning and ordering between operators. Tracking that kind of metadata would be very useful for speeding up a large number of operations, including sorting (!), partitioned aggregation, windowing, AsOf joins and merge joins. This is work we are very interested in, so stay tuned!

Conclusion

With apologies to Guido van Rossum, there is usually more than one way to do something, but each way may have radically different performance characteristics. One of the jobs of a relational database with a declarative query language like SQL is to make intelligent choices between the options so you the user can focus on the result. Here at DuckDB we look forward to finding more ways to plan your queries so you can focus on what you do best!

Notes

- The tests were all run on an iMac Pro with a 2.3 GHz 18-core Intel Xeon W CPU and 128 GB of RAM.
- The raw test data and visualizations are available on our Tableau Public repository.
- The script to generate the data is in my public DuckDB tools repository.

Prefix Aliases in SQL

Publication date: 2025-02-25

Author: Hannes Mühleisen

TL;DR: You can now put your aliases first in DuckDB's SQL dialect with a colon, e.g., `SELECT a: 42;`

Syntax

> Quote Perhaps we should just leave nature alone, to its simple one-assed schematics.
>
> -- Dr. Alphonse Mephesto, South Park Episode #5

There is often more than one way to do things in our beloved SQL. For example, you can define join conditions *implicitly* (and dangerously) in the `WHERE` clause or use the (better) `JOIN ... ON ...` syntax. Generally, having "more than one way to do things" can be confusing and even outright dangerous sometimes.

Having said that we here at DuckDB pride ourselves in *friendlier* SQL. There are just too many people typing this stuff by hand, especially in the more ad-hoc world of analytics.

If there is a *good* reason to expand the SQL syntax with something useful we are at least considering it. Others seem to be watching closely. For example, our `GROUP BY ALL` syntax has by now been picked up by almost every SQL system out there (and their little brothers).

Aliases

In SQL, a user can define *aliases* for lots of things like `SELECT` expressions, table names, subqueries, etc. This is sometimes just nice to have readable column names in the result and sometimes required to refer back to a complex expression in for example the `ORDER BY` clause without just repeating it and praying for the optimizer. Aliases are defined *after* the thing they alias using `AS` but actually typing the `AS` term is optional. For example, those two statements are equivalent:

```
SELECT 42 AS fortytwo;
SELECT 42 fortytwo;
```

We can see the alias follows the expression (`42`) and the `AS` is optional. Having the alias *behind* the thing it describes is actually somewhat rare in programming, it is much more typical to define the alias first and then provide the expression. For example, in C:

```
int fortytwo = 42;
```

It seems to be a good idea to first state the alias, after all, this is what we are going to refer to this thing later on. Forcing it the other way around like SQL just increases the mental load. In addition, having the aliases last can make them quite hard to find if there are several complex expressions in a query. For example, here are the first few lines of the infamous TPC-H Query 1:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) sum_qty,
    sum(l_extendedprice) sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) sum_charge,
    avg(l_quantity) avg_qty,
    avg(l_extendedprice) avg_price,
    avg(l_discount) avg_disc,
    count(*) count_order
    ...
```

It's hard to spot all the aliases here, and this is not even a complex example. What if you could put the aliases first in SQL? Well, wait no more.

Prefix Aliases in DuckDB

In the latest DuckDB release, 1.2.0, we have quietly shipped yet another useful (we think) syntax extension to allow the alias to come before the thing it names using the colon (`:`) syntax. This turned out to be less difficult than we thought, "all" it took was some modification to the Bison parser we inherited from Postgres (but are in the process of replacing). Here is the example from before again:

```
SELECT fortytwo: 42;
```

Prefix aliases also work for table names, e.g.:

```
SELECT *
FROM my_table: some_other_table;
```

Aliases can be quoted if necessary using *double* quotes:

```
SELECT "forty two": 42;
```

Prefix aliases can be used to name just about everything, for example expressions, function calls and subqueries in the `SELECT` clause:

```
SELECT
    e: 1 + 2,
    f: len('asdf'),
    s: (SELECT 42);
```

They can also apply to function calls and subqueries in the `FROM` clause, e.g.:

```
SELECT *
FROM
    r: range(10),
    v: (VALUES (42)),
    s: (FROM range(10))
```

Note that the `VALUES` clause and the subquery with `FROM` need additional parentheses to work here, this was required to pacify the evil Bison parser generator. Let's look at the Q1 example from earlier, but this time with prefix aliases:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum_qty:        sum(l_quantity),
    sum_base_price: sum(l_extendedprice),
    sum_disc_price: sum(l_extendedprice * (1-l_discount)),
    sum_charge:     sum(l_extendedprice * (1-l_discount) * (1+l_tax)),
    avg_qty:        avg(l_quantity),
    avg_price:      avg(l_extendedprice),
    avg_disc:       avg(l_discount),
    count_order:    count(*)
    ...
```

There is no semantic difference between using `:` or `AS`, they lead to the same query constructs.

Credits

Credit for this idea goes to Looker veteran Michael Toy. We're grateful for his suggestion. We also would like to thank the legendary Lloyd Tabb for recommending Michael's idea to us. Also check out Mark Needham's video on DuckDB's prefix aliases.

Reading and Writing Google Sheets in DuckDB

Publication date: 2025-02-26

Authors: Alex Monahan, Archie Wood

TL;DR: Securely read from and write to Google Sheets directly in DuckDB using the GSheets community extension! For ad hoc querying, authentication is as easy as logging into Google from a browser. Scheduled workflows can use persistent DuckDB Secrets. SQL-on-Sheets has arrived!

Spreadsheets Are Everywhere

Is anything more polarizing for data folks than spreadsheets? Wait, don't answer that, we don't have time to talk about leading and trailing commas again...

The fact is that spreadsheets are everywhere. It is estimated that there are over 750 million spreadsheet users, compared to just 20 to 30 million programmers. That includes all languages put together!

There are a number of ways that using a spreadsheet can improve a data workflow. Blasphemy you say! Well, imagine if your database could actually read and write those spreadsheets. Spreadsheets are often the best place to manually edit data and they also provide highly customizable pivoting for self-serve analytics.

Now, you can use DuckDB to seamlessly bridge the gap between data-y folks and business-y folks! With a simple in-browser authentication flow, or an automateable private key file flow, you can both query from and load into Google Sheets.

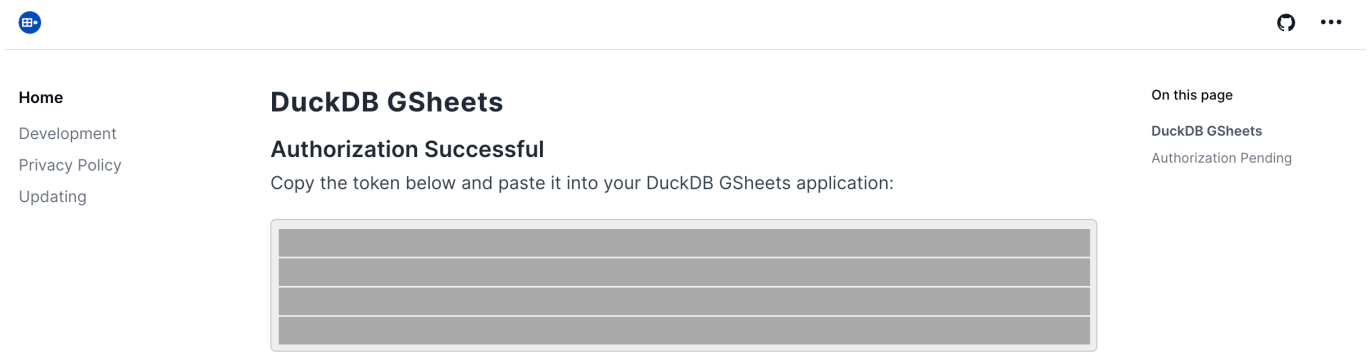
The GSheets extension was originally authored by Archie from the team at Evidence, but has since had significant contributions from Alex and Michael.

Getting Started with the GSheets Extension

The first few steps are to install the gsheets community extension and authenticate with Google.

```
INSTALL gsheets FROM community;
LOAD gsheets;

-- Authenticate with a Google Account in the browser (default)
CREATE SECRET (TYPE gsheet);
```

As a part of the CREATE SECRET command, a browser window will open and allow for a login and copying a temporary token to then paste back into DuckDB.



Examples of Reading from Sheets

Now that you are authenticated, DuckDB can query any Sheet that your Google account has access to. This includes any publicly available sheets like the one below, so give it a run!

```
FROM
'https://docs.google.com/spreadsheets/d/1B4RFuOnZ4ITZ-nR9givZ7vWVOTVddC3VTKuSqgifiyE/edit?gid=0#gid=0';
```

Gotham Wisdom
You either die a hero
or live long enough to query from spreadsheets

Copy the URL of the Sheet to query when viewing the sheet of interest within the workbook. The `gid` query string parameter is the id of that specific sheet.

There are two ways to pass in additional parameters:

1. Add them to the end of the URL as query string parameters or
2. Use the `read_gsheet` table function and specify them as separate SQL parameters.

The repository README has a variety of examples and some are included below!

> Query string parameters must be placed after a `?`. Each parameter is formatted as a `key=value` pair, and multiple are separated with `&`.

Reading a Specific Sheet and Range

By default, the GSheets extension will read all data on the first sheet in the workbook. The `sheet` and `range` parameters (or their query string equivalents) allow for targeted reads.

For example, to read only the first 3 cells on the `We <3 Ducks` sheet, these two statements are equivalent:

```
-- The sheet with the gid of 0 is named 'We <3 Ducks' (because of course it is!)
FROM read_gsheet(
    'https://docs.google.com/spreadsheets/d/1B4RFuOnZ4ITZ-nR9givZ7vWVOTVddC3VTKuSqgifiyE/edit',
    sheet = 'We <3 Ducks',
    range = 'A1:A3'
    );

FROM 'https://docs.google.com/spreadsheets/d/1B4RFuOnZ4ITZ-
nR9givZ7vWVOTVddC3VTKuSqgifiyE/edit?gid=0#gid=0&range=A1:A3';
```

The Google Sheets API helpfully skips empty rows at the end of a dataset or empty columns to the right. Feel free to specify a slightly bigger `range` if your data may grow! Additionally, the `range` can be specified as a set of columns (e.g., `D:X`) to be friendlier to a variable number of rows.

Data Types

The extension will sample the first row of data in the sheet to attempt to determine the data types of the columns. (We have plans to improve this sampling and are open to contributions!) To skip this step and define the data types within SQL, set the `all_varchar` parameter to `true`. The example below also demonstrates that the full URL is not needed – only the Google Workbook identifier.

```
FROM read_gsheet(
    '1B4RFuOnZ4ITZ-nR9givZ7vWVOTVddC3VTKuSqgifiyE',
    sheet = 'We <3 Ducks',
    range = 'A:A',
    all_varchar = true
    );
```

It is also possible to query data without a header row by setting the `header` parameter to false. Columns will be given default names and can be renamed in SQL.

Examples of Writing to a GSheet

Another key capability of the GSheets extension is to write the results of any DuckDB query to a Google Sheet!

> By default, the entire Sheet will be replaced with the output of the query (including a header row for column names), starting in cell A1 of the first sheet. See below for examples that adjust this behavior!

```
-- Here you will need to specify your own Sheet to experiment with!
-- (We can't predict what folks would write to a public Sheet...
-- Probably just memes, but there is always that one person, you know?)
COPY (FROM range(10))
TO 'https://docs.google.com/spreadsheets/d/...'  (
    FORMAT gsheet
);
```

Writing to a Specific Sheet and Range

As with reading, both query string parameters and SQL parameters can be used to write to a specific `sheet` or `range`. Similarly, the SQL parameters take precedence. These examples are equivalent:

```
COPY (FROM range(10))
TO 'https://docs.google.com/spreadsheets/d/...?' (
    FORMAT gsheet,
    sheet 'The sheet name!',
    range 'A2:Z10000'
);
```

```
COPY (FROM range(10))
TO 'https://docs.google.com/spreadsheets/d/...?gid=123#gid=123&range=A2:Z10000' (
    FORMAT gsheet
);
```

The `header` boolean parameter can also be used to determine whether the column names should be written out or not.

Overwriting or Appending

At times, it is helpful to avoid clearing out other data in a Sheet before copying. This is especially handy when writing to specific ranges. Perhaps columns C and D can come from DuckDB and the remainder can be spreadsheet formulas. It would be great to just clear out columns C and D!

To adjust this behavior, pass in these boolean parameters to the COPY function. `OVERWRITE_SHEET` is the default where the entire sheet is cleared out prior to copying. `OVERWRITE_RANGE` will only clear out the specified range.

If both are set to `false`, then data will be appended without any other cells being cleared out. Typically, when appending it is not desirable to include the column headers in the output. Helpfully, the `header` parameter defaults to `false` in the append case, but it can be adjusted if needed.

```
-- To append, set both flags to false.
COPY (FROM range(10))
TO 'https://docs.google.com/spreadsheets/d/...?gid=123#gid=123&range=A2:Z10000' (
    FORMAT gsheet,
    OVERWRITE_SHEET false,
    OVERWRITE_RANGE false
    -- HEADER false is the default in this case!
);
```

Automated Workflows

Working with spreadsheets is great for ad hoc work, but it can also be powerful when ingrained in automated processes. If you want to schedule an interaction with Google Sheets, a key file containing a private key will be needed instead of the in-browser authentication method.

After acquiring this key file, the persistent private key must be converted to a temporary token once every 30 minutes. That process is now automated with the `key_file` secret provider. Create the secret with a command like below, pointing to the JSON file exported from Google.

```
CREATE OR REPLACE PERSISTENT SECRET my_secret (
    TYPE gsheet,
    PROVIDER key_file,
    FILEPATH 'credentials.json'
);
```

As the secret is created, the private key is stored in DuckDB and a temporary token is created. The secret can be stored in memory or optionally persisted to disk (unencrypted) using the `PERSISTENT` keyword. The temporary token is cached within the SECRET as well and is recreated if it is over 30 minutes old.

This unlocks the use of the GSheets extension within pipelines, like GitHub Actions (GHA) or other orchestrators like dbt. The best practice is to store the `credentials.json` file as a secret within your orchestrator and write it out to a temporary file. An example GHA workflow is here, which uses this Python script to query a Sheet.

Developing the Extension

The Google Sheets extension is a good example of how DuckDB's extension GitHub template and CI/CD workflows can let even non-C++ experts contribute to the community! Several of the folks who have contributed thus far (thank you!!), including the authors of this post, are not traditional C++ programmers. The combination of a great template, examples from other extensions, and a little help from some LLM-powered "junior devs" made it possible. We encourage you to give your extension idea a shot and reach out on Discord if you need some help!

Roadmap

There are a few more fun features we are thinking about for the extension – we are open to PRs and collaborators!

We would like to use a better heuristic for detecting data types when reading from a Sheet. The DuckDB type system is more advanced than Sheets, so it would be beneficial to be more precise.

Enabling the GSheets extension to work in DuckDB-Wasm would allow in-browser applications to query Sheets directly – no server needed! Several `http` functions need some modification to work in a browser environment.

The OAuth flow that powers the browser-based login may be useful for authenticating to other APIs. We are wondering if maybe it would be possible to have a generic OAuth community extension. There are no concrete plans for this at the moment, but if anyone is interested, please reach out!

Closing Thoughts

At MotherDuck (where Alex works), we have this extension running in production for several internal data pipelines! We have automated exports of forecasts from our warehouse into Sheets and continually load manually collected customer support data into our (MotherDuck-powered) data warehouse. As a result, our KPI dashboards include context from folks talking directly to customers!

Michael Harris has also contributed to the extension (thank you!), and Definite has deployed GSheets scheduled jobs into production for multiple customers!

How do you use Google Sheets in your data analysis workflow and how can DuckDB help? We would love to hear your ideas on BlueSky, LinkedIn, or X / Twitter!

Now go automate that Sheet with some SQL!

Gems of DuckDB 1.2

Publication date: 2025-03-06

Author: The DuckDB team

TL;DR: We highlight a few exciting features that were introduced in DuckDB 1.2.

We published the DuckDB 1.2.1 bugfix release yesterday. As usual, please consult the release notes for the full list of changes and the installation page for instructions on installing or upgrading. In this post, we'll highlight a few features that were recently added to DuckDB and improvements that have been made in its ecosystem.

New Clients Page

DuckDB's client APIs (or "drivers") are distributed through several centralized repositories, such as CRAN for R and Maven for Java. To help users keep track of the rollout of a new DuckDB release, we reworked our "Clients" page to show the latest version for each client. The page also clarifies the support tiers that apply to clients.

Simpler Installation

In line with DuckDB's "low friction" principle, we made sure that you can install the DuckDB command line client more easily.

Installation Script on Linux and macOS

DuckDB can now be installed on UNIX-like systems with an installation script:

```
curl https://install.duckdb.org | bash
```

The script determines your operating system and architecture, fetches the tag of latest release, and if not present downloads the latest available DuckDB binary to `~/.duckdb/cli` (the `~/.duckdb` folder is already used to store extensions).

Running the script does not require root (sudo) privileges, and it only uses the `curl` and `zcat` tools, which are widely available. You can inspect the shell script by visiting `install.duckdb.org` in your browser.

Signed Binaries on Windows

Starting with v1.2.1, we ship digitally signed binaries for the DuckDB Windows command line client. This means that you can run DuckDB in environments where signed binaries are a requirement.

Unsung DuckDB 1.2 Features

Further to the "What's New in 1.2.0" section of the announcement blog post, we collected five new features that were introduced in v1.2.0.

`OR` / `IN` Filter Pushdown

Starting with version 1.2.0, DuckDB supports `OR` and `IN` expressions for filter pushdown. This optimization comes especially handy when querying remote Parquet files or DuckDB databases.

`-f` Command Line Flag

The DuckDB CLI client now supports the `-f` flag to execute SQL script files:

```
duckdb -f script.sql
```

This is equivalent to:

```
duckdb -c ".read script.sql"
```

This feature is documented in the DuckDB tldr page. If you have tldr installed, you can get this page in the CLI via `tldr duckdb`.

`allowed_directories` / `allowed_paths` Options

We continue to improve support for operating DuckDB in secure environments. The `allowed_directories` and `allowed_paths` options allow restricting DuckDB's access to certain directories or files (resp.). These options allows fine-grained access control for the file system. For example, you can set DuckDB to only use the `/tmp` directory.

```
SET allowed_directories = ['/tmp'];
SET enable_external_access = false;
FROM read_csv('test.csv');
```

With the setting applied, DuckDB will refuse to read files in the current working directory:

```
Permission Error:
Cannot access file "test.csv" - file system operations are disabled by configuration
```

`sum(BOOLEAN)`

You can now directly compute the sum of `BOOLEAN` expressions without wrapping them into a `CASE` expression:

```
SELECT sum(l_extendedprice > 500) FROM lineitem;
```

This is equivalent to:

```
SELECT sum(CASE WHEN l_extendedprice > 500 THEN 1 END) FROM lineitem;
```

Excel Extension

Prior to DuckDB 1.2, Excel files were only supported by the `spatial` extension, which is a heavyweight extension with several dependencies. Starting with 1.2, the `excel` extension – which was previously limited to computing a few formulas – can read and write Excel sheets. For example:

```
FROM read_xlsx('test.xlsx', header = true);
```

> If you would like to work with *Google Sheets* sheets, take a look at the `gsheets` community extension.

On the Interweb

There are several repositories and pieces of information on DuckDB on the internet. We highlight two important ones:

Awesome DuckDB

The ecosystem around DuckDB keeps growing: many projects are built both with DuckDB and within DuckDB. The community-driven `awesome-duckdb` repository, maintained by David Gasquez, lists these and has recently surpassed 200 entries.

DuckDB File Signature

DuckDB's file signature, DUCK (hex: `44 55 43 4B`), is now listed on Wikipedia.

Parquet Information Sheet

We added DuckDB to the "Implementation status" page of the Parquet documentation. In the process, we also improved DuckDB's Parquet support, e.g., by adding the `FLOAT16` logical type.

Parquet Bloom Filters in DuckDB

Publication date: 2025-03-07

Author: Hannes Mühleisen

TL;DR: DuckDB now supports reading and writing Parquet Bloom filters.

One of the key features of the Parquet file format is the ability for readers to *selectively* read only the data that is relevant to a particular query. To support this, Parquet files contain column *statistics*, most notably the minimum and maximum value for each column in each row group. If a query is filtering with a particular value, and the data is – as it is often – somewhat sorted, a reader can "prove" that a particular row group cannot contain values relevant to the query. DuckDB heavily leverages this, and is able to – even when querying remote endpoints – selectively only read the parts of a Parquet file relevant to a query. For details on how this works, see our by now ancient blog post "Querying Parquet with precision using DuckDB".

However, there are some limitation to this approach. What if a column's data is randomly shuffled? In that case, all values occur in all row groups and the minimum and maximum statistics are less useful because we can only exclude values that are outside the minimum and maximum *of the entire column*. Parquet will use dictionary encoding if there are not too many distinct values in the column. In theory, that dictionary could be used to eliminate row groups but there are two problems with this approach: Parquet (inexplicably) allows *switching* from dictionary to plain encoding *halfway through a row group*. If the dictionary alone were used to eliminate the row group, but the plain part contained the queried value, this would lead to incorrect results. Furthermore, the dictionary is part of the actual column data. If we're reading the column data just to look at the dictionary we have already incurred most of the cost of reading the column in the first place.

Obscure sidenote: Again, in theory the column metadata contains the list of encodings that occur in that column. *If* that list is correct and does *not* include the plain encoding the dictionary *could* – again, in theory – be used for slightly earlier row group elimination. But the usefulness of this approach is more than dubious.

Parquet Bloom Filters

The good people over at the Parquet PMC have recognized that there is room for improvement here and added Bloom filters for Parquet back in 2018. In a nutshell, Bloom filters are compact but approximate index structures for a set of values. For a given value, they can either say with certainty that a value is *not* in the set or that it *may be* in the set, with a false positive ratio depending on the size of the Bloom filter and the amount of distinct values added to it. For now, we can just treat a Bloom filter like an opaque series of bytes with magic properties.

When used, Parquet files can contain a Bloom filter for each column in each row group. Each Bloom filter can be at an arbitrary location in the file (`bloom_filter_offset`). At the offset in the file, we find another Thrift-encoded structure, the `BloomFilterHeader`. This structure has a field for the length of the filter, and some algorithmic settings which are currently redundant because there is only one valid setting for all of them. But decode the header you must to find out where the header ends and where the filter bytes begin. Finally, we have arrived at the precious magic bytes of the Bloom filter. We can now test the filter against any query predicates and see if we can skip the row group entirely.

More obscure sidenotes: while the column metadata *does* contain a field to store the size of the filter (`bloom_filter_length`) it is also optional and some writers (looking at you, Spark) annoyingly only set the offset, not the length. Also, the specification describes two possible filter locations. By also not requiring a length in the metadata, this makes it difficult slash impossible to read all Bloom filters for a Parquet file in a single range request. It is also not clear why the Bloom filters needed to *each* be prefixed with a Thrift metadata blob, when this information could also be contained in the ColumnMetaData. Or perhaps, god forbid, the filters could have become part of the main Parquet metadata itself. In the end we end up with a lot of additional reads to find and read the Bloom filter bytes, in principle requiring a careful trade-off between reading the filters and "just" reading the column brute-force.

DuckDB Bloom Filters

As of the last feature release (1.2.0), DuckDB supports *both reading and writing* of Parquet Bloom filters. This happens completely transparently to users, no additional action or configuration is required.

Writing

Currently, Bloom filters are supported for the following types:

- Integer types (`TINYINT`, `UTINYINT`, `SMALLINT`, `USMALLINT`, `INTEGER`, `UINTEGER`, `BIGINT`, `UBIGINT`)
- Floating point types (`FLOAT`, `DOUBLE`)
- String types (`VARCHAR` and `BLOB`)

Nested types (lists, structs, arrays) are currently *not* supported, but they might be added in a future release. In general, Bloom filters will be written if DuckDB decides to use dictionary encoding for a particular column (chunk) within the row group. There is a `COPY` parameter that controls the maximum dictionary size (`DICTIONARY_SIZE_LIMIT`), but this parameter is by default set to 10% of the row group size (`ROW_GROUP_SIZE`), which is 122,880 rows by default. Those values have been found to be reasonable first approximations for most use cases but users are of course encouraged to experiment with both parameters if they encounter performance issues. As the number of distinct values in a Bloom filter grows, its size needs to be improved to maintain a certain false positive ratio. By default, filters size is chosen using an "acceptable" false positive ratio of 1% or 0.01. The new `BLOOM_FILTER_FALSE_POSITIVE_RATIO` `COPY` parameter controls the acceptable ratio. In general, false positive hurt more the slower the read path is.

Reading

During reading, DuckDB will automatically use constant-comparison filter predicates in the query (e.g., `WHERE a = 42`) to probe the Bloom filter (if present) and skip row groups where the Bloom filter can guarantee there are no matching rows in the group. Again, this happens transparently to users and there is no configuration that needs to be set.

Users can find out if a given Parquet file contains Bloom filters: the `parquet_metadata` function was extended with two new columns, `bloom_filter_offset` and `bloom_filter_length`. Furthermore, to actually find out which row groups would be eliminated by Bloom filters for a given file and column, we have added the `parquet_bloom_probe` function. For example, `parquet_bloom_probe('file.parquet', 'col1', 42)` returns a table for each row group in `file.parquet` that indicates whether the value `42` can guaranteed not to be in each row group or not for column `col1`. Most users will not need to use these functions however, they just help with debugging (and testing).

Example Use Case

Let's showcase the Parquet Bloom filters in DuckDB with an example. First, we create an example file `filter.parquet` that *will* contain Bloom filters:

```
COPY (
    FROM range(10) r1, range(10_000_000) r2
    SELECT r1.range * 100 AS r
    ORDER BY random()
)
TO 'filter.parquet'
(FORMAT parquet, ROW_GROUP_SIZE 10_000_000);
```

```
SELECT r, count(*)
FROM 'filter.parquet'
GROUP BY r
ORDER BY r;
```

The file contains 10 distinct values `(0, 100, 200 ... 900)`, each repeated ten million times. So in total, 100 million rows. The resulting Parquet file weighs in at 88 MB.

We will also create an equivalent file but *without* Bloom filters. We achieve this by setting the `DICTIONARY_SIZE_LIMIT` to 1:

```
COPY 'filter.parquet' to 'nofilter.parquet'
(FORMAT parquet, DICTIONARY_SIZE_LIMIT 1, ROW_GROUP_SIZE 10_000_000);
```

The contents of both files will be equivalent, but `nofilter.parquet` will not use dictionary encoding and thus not use Bloom filters. As a result, the file is also larger, weighing in at 181 MB. However, there is a much larger difference when querying for non-existing values, in our example `501`:

```
.timer on
SELECT sum(r) FROM 'filter.parquet'   WHERE r = 501;
SELECT sum(r) FROM 'nofilter.parquet' WHERE r = 501;
```

The first query completes in ca. than 0.002 s, where the second query takes 0.1 s. This large difference can be explained by Bloom filters! Since `501` does not actually occur in the query, DuckDB can automatically exclude all row groups and not actually read any data besides the Bloom filters. We can explore this further using the `parquet_metadata` function:

```
FROM parquet_metadata('filter.parquet')
SELECT row_group_id, stats_min, stats_max,
    bloom_filter_offset, bloom_filter_length
ORDER BY row_group_id;
```

row_group_id	stats_min	stats_max	bloom_filter_offset	bloom_filter_length
0	0	900	92543967	47
…				
9	0	900	92544390	47

We can see that there are ten row groups, and that there is a quite compact Bloom filter for reach row group with a length of 47 bytes each. That's ca. 500 bytes added to fairly large file, so rather irrelevant for file size.

If we run the query on the other file, we can see the lack of Bloom filters:

```
FROM parquet_metadata('nofilter.parquet')
SELECT row_group_id, stats_min, stats_max,
       bloom_filter_offset, bloom_filter_length
ORDER BY row_group_id;
```

row_group_id	stats_min	stats_max	bloom_filter_offset	bloom_filter_length
0	0	900	NULL	NULL
…				

We can further explore the Bloom filters in the file using the `parquet_bloom_probe` function. For example, for the value of 500 (which exists in the data), the function shows the following:

```
FROM parquet_bloom_probe('filter.parquet', 'r', 500);
```

file_name	row_group_id	bloom_filter_excludes
filter.parquet	0	false
…	…	…
filter.parquet	9	false

So the Bloom filter cannot exclude any row group because the value `500` is contained in all row groups. But if we try a *non-existent* value, the Bloom filter strikes:

```
FROM parquet_bloom_probe('filter.parquet', 'r', 501);
```

file_name	row_group_id	bloom_filter_excludes
filter.parquet	0	true
...	...	...
filter.parquet	9	true

Here, we can confidently skip all row groups because the Bloom filter guarantees that there can be no matching values in those row groups. All that with 47 bytes per row group.

Conclusion

DuckDB's new Bloom filter support for Parquet files can greatly reduce the amount of data to be read in certain scenarios, greatly improving query performance. This is particularly useful if files are read over a slow network connection or if row groups are particularly large with few distinct yet non-clustered values.

The DuckDB Local UI

Publication date: 2025-03-12

Authors: Jeff Raymakers, Gábor Szárnyas

TL;DR: The DuckDB team and MotherDuck are excited to announce the release of a local UI for DuckDB shipped as part of the `ui` extension.

The DuckDB project was built to make it simple to leverage modern database technology. DuckDB can be used from many popular languages and runs on a wide variety of platforms. The included Command Line Interface (CLI) provides a convenient way to interactively run SQL queries from a terminal window, and several third-party tools offer more sophisticated UIs.

The DuckDB CLI provides advanced features like interactive multi-line editing, auto-complete, and progress indicators. However, it can be cumbersome for working with lengthy SQL queries, and its data exploration tools are limited. Many of the available third party UIs are great, but selecting, installing, and configuring one is not straightforward. Using DuckDB through a UI should be as simple as using the CLI. And now it is!

> The DuckDB UI is the result of a collaboration between DuckLabs and MotherDuck and is shipped as part of the `ui` extension.

Introducing the DuckDB UI

Starting with DuckDB v1.2.1, a full-featured local web UI is available out-of-the-box! You can start it from the terminal by launching the DuckDB CLI client with the `-ui` argument:

```
duckdb -ui
```

You can also run the following SQL command from a DuckDB client (e.g., CLI, Python, Java, etc.):

```
CALL start_ui();
```

Both of these approaches install the `ui` extension (if it isn't installed yet), then open the DuckDB UI in your browser:

The DuckDB UI uses interactive notebooks to define SQL scripts and show the results of queries. However, its capabilities go far beyond this. Let's go over its main features.

> The DuckDB UI runs all your queries locally: your queries and data never leave your computer. If you would like to use MotherDuck through the UI, you have to opt-in explicitly.

Features

Databases

Your attached databases are shown on the left. This list includes in-memory databases plus any files and URLs you've loaded. You can explore tables and views by expanding databases and schemas.

Table Summary

Click on a table or view to show a summary below. The UI shows the number of rows, the name and type of each column, and a profile of the data in each column.

Select a column to see a more detailed summary of its data. You can use the *"Preview data"* button near the top right to inspect the first 100 rows. You can also find the SQL definition of the table or view here.

Notebooks

You can organize your work into named notebooks. Each cell of the notebook can execute one or more SQL statements. The UI supports syntax highlighting and autocomplete to assist with writing your queries.

You can run the whole cell, or just a selection, then sort, filter, or further transform the results using the provided controls.

Column Explorer

The right panel contains the Column Explorer, which shows a summary of your results. You can dive into each column to gain insights.

MotherDuck Integration

If you would like to connect to MotherDuck, you can sign into MotherDuck to persist files and tables to a cloud data warehouse crafted for using DuckDB at scale and sharing data with your team.

...And More!

The DuckDB UI is under active development. Expect additions and improvements!

Footprint

Like the DuckDB CLI, the DuckDB UI creates some files in the `.duckdb` directory in your home directory. The UI puts its files in a sub-directory, `extension_data/ui`:

- Your notebooks and some other state are stored in a DuckDB database, `ui.db`.
- When you export data to the clipboard or a file (using the controls below the results), some tiny intermediate files (e.g., `ui_export.csv`) are generated. Your data is cleared from these files after the export is completed, but some near-empty files remain, one per file type.

Internals

Support for the UI is implemented in a DuckDB extension. The extension embeds a localhost HTTP server, which serves the UI browser application, and also exposes an API for communication with DuckDB. In this way, the UI leverages the native DuckDB instance from which it was started, enabling full access to your local memory, compute, and file system.

Results are returned in an efficient binary form closely matching DuckDB's in-memory representation (DataChunk). Server-sent events enable prompt notification of updates such as attaching databases. These techniques and others make for a low-latency experience that keeps you in your flow.

See the UI extension documentation for more details.

Summary

In this blog post, we presented the new DuckDB UI, a powerful web interface for DuckDB.

The DuckDB UI shares many of its design principles with the DuckDB database. It's simple, fast, feature-rich, and portable, and runs locally on your computer. The DuckDB UI extension is also open source: visit the `duckdb/duckdb-ui` repository if you want to dive in deeper into the extension's code.

> The repository does not contain the source code for the frontend, which is currently not available as open-source. Releasing it as open-source is under consideration.

For help or to share feedback, please file an issue, join the `#ui` channel in either the DuckDB Discord or the MotherDuck Community Slack.

Happy quacking!

Update (2025-03-20)

We are very happy to see the overwhelmingly positive response to the announcement of the DuckDB Local UI. In this short update, we'd like to address some of the questions raised by the community:

- Regarding the **licensing of the frontend code** (HTML, JavaScript, etc.), MotherDuck is currently reviewing licensing options for the UI code and assets and will clarify shortly.
- Regarding **offline ("air-gapped") operations,** we are working towards supporting offline usage in a future version, and are currently exploring how to provide version notifications as upgrades become available.
- Regarding the **client-server protocol for the UI,** we're going to work on opening up the client-server protocol so that other user interfaces can use the "back-end" part of the UI extension to run queries in DuckDB.

Preview: Amazon S3 Tables in DuckDB

Publication date: 2025-03-14

Authors: Sam Ansmink, Tom Ebergen, Gábor Szárnyas

TL;DR: We are happy to announce a new preview feature that adds support for Apache Iceberg REST Catalogs, enabling DuckDB users to connect to Amazon S3 Tables and Amazon SageMaker Lakehouse with ease.

> The AWS Storage Blog also published a post on this feature, see Streamlining access to tabular datasets stored in Amazon S3 Tables with DuckDB.

Iceberg Ahead!

In recent years, the Iceberg open table format has become increasingly popular. Major data warehouse platforms such as Databricks, Snowflake, Google BigQuery and AWS have all announced or already implemented support for Iceberg tables. These platforms also support Iceberg catalogs, which are responsible for tracking current metadata for a collection of Iceberg tables grouped by namespaces.

DuckDB has supported reading Iceberg tables since September 2023 via the `iceberg` extension. Today, we are happy to introduce a new preview feature in this extension, which allows attaching to Iceberg REST catalogs. This preview release coincides with two AWS announcements yesterday: support for Iceberg tables in Amazon S3 Tables and the GA release of the integration between S3 Tables and SageMaker Lakehouse (AWS Glue Data Catalog). In practice, these developments mean that DuckDB now provides an end-to-end solution for reading Iceberg tables in S3 Tables and SageMaker Lakehouse.

> DuckDB's support for Iceberg REST Catalog endpoints in Amazon S3 Tables is the result of a collaboration between AWS and DuckLabs.

Using Apache Iceberg REST Catalogs in DuckDB

Steps for Installing

To connect to Apache Iceberg REST Catalogs in DuckDB, make sure you are running the **latest stable** DuckDB release (version 1.2.1). For our example steps, we'll use the DuckDB CLI client. You can obtain this client from the installation page and start it with:

```
duckdb
```

Next, we need to install the "bleeding edge" versions of the required extensions from the `core_nightly` repository.

```
FORCE INSTALL aws FROM core_nightly;
FORCE INSTALL httpfs FROM core_nightly;
FORCE INSTALL iceberg FROM core_nightly;
```

> For more information on using the `core_nightly` repository, please see the notes at the end of the post.

With these extensions installed, your DuckDB is now capable of using Apache Iceberg REST Catalogs. Let's find some data.

Setting up an Amazon S3 Table Bucket

(If you already have Iceberg tables in Amazon S3 Tables, you can skip to the "Reading Iceberg Catalogs with DuckDB" section.)

In this post, we demonstrate how to read data from Amazon S3 Tables. To follow along, make sure that your account has `s3tables` permissions and create a new S3 table bucket. Note that Amazon S3 Tables is currently only supported in selected AWS regions.

Populating an Amazon S3 Table Bucket

If you don't have an S3 table bucket with tables already, we found the easiest way to get going is to create a table using Amazon Athena. See their instructions. For our example, we created a simple table with three columns using the Athena query editor:

```
CREATE TABLE duck_species (
    id INT,
    english_name STRING,
    latin_name STRING
) TBLPROPERTIES ('table_type' = 'ICEBERG');
```

Let's insert some data to the table:

```
INSERT INTO duck_species VALUES
    (0, 'Anas nivis', 'Snow duck');
```

Reading Amazon S3 Tables with DuckDB

Querying S3 Tables with DuckDB is really easy. The first step is to get your AWS credentials into DuckDB. You can achieve this in two ways. First, you can let DuckDB detect your AWS credentials and configuration based on the default profile in your `~/.aws` directory by creating the following secret using the Secrets Manager:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
```

Alternatively, you can set the AWS key, secret, and region values manually. For example:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

> **Tip.** To see the secrets in your session, run `FROM duckdb_secrets();`{:.language-sql .highlight}

Next, point DuckDB to your S3 table bucket. You can do so by copy-pasting the S3 Tables ARN value directly from the AWS Management Console and using it in the ATTACH command:

```
ATTACH 'arn:aws:s3tables:<us-east-1>:<111122223333>:bucket/<bucket-name>'
    AS my_s3_tables_catalog (
        TYPE iceberg,
        ENDPOINT_TYPE s3_tables
    );
```

And that's all! Now, DuckDB is connected to Amazon S3 Tables. To show the available tables, run:

```
SHOW ALL TABLES;
```

database varchar	schema varchar	name varchar	column_names varchar[]	column_types varchar[]	temporary boolean
my_s3_tables_catalog	ducks	duck_species	[__]	[INTEGER]	false

You can query tables as if they were ordinary DuckDB tables:

```
FROM my_s3_tables_catalog.ducks.duck_species;
```

id int32	english_name varchar	latin_name varchar
0	Anas nivis	Snow duck

You also have an alternative option to connect to S3 Tables using the Amazon SageMaker Lakehouse (AWS Glue Data Catalog) Iceberg REST Catalog endpoint. To do so, run:

```
ATTACH '<account_id>:s3tablescatalog/<namespace_name>'
AS (
    TYPE iceberg,
    ENDPOINT_TYPE glue
);
```

> **Tip.** If you need basic read access to tabular data in a single S3 table bucket, use the `s3_tables` endpoint type. If you want a unified view across all of your tabular data in AWS, use the `glue` endpoint type.

Schema Evolution

A key feature of the Iceberg format is schema evolution, i.e., the ability to follow changes in the table's schema. To demonstrate this, we go back to the Athena query editor and add a new column to the `duck_species` table:

```
ALTER TABLE duck_species
    ADD COLUMNS (conservation_status STRING);
```

Then, we insert a few more duck species:

```
INSERT INTO duck_species VALUES
    (1, 'Anas eatoni', 'Eaton''s pintail', 'Vulnerable'),
    (2, 'Histrionicus histrionicus', 'Harlequin duck', 'Least concern');
```

Let's run the query again from DuckDB:

```
FROM my_s3_tables_catalog.ducks.duck_species;
```

The query now returns a table with the additional fourth column, which has a `NULL` value in the row inserted before the change in the schema – as expected.

id int32	english_name varchar	latin_name varchar	conservation_status varchar
1	Anas eatoni	Eaton's pintail	Vulnerable
2	Histrionicus histrionicus	Harlequin duck	Least concern
0	Anas nivis	Snow duck	NULL

Conclusion

The latest preview release of the DuckDB `iceberg` extension enables directly reading tables using Iceberg REST endpoints. This allows you to query Amazon S3 Tables and Amazon SageMaker Lakehouse (AWS Glue Data Catalog) with ease. As of today, the extension is in an experimental state and is under active development. We will publish a stable release later this year.

Footnotes

Cleaning Up

If you created a new S3 table bucket to follow the examples, don't forget to clean up by deleting your S3 table bucket.

Using the `core_nightly` Repository

The extensions used in this blog post are currently experimental, and hence they are distributed through the `core_nightly` repository. If you want to switch back to using extensions from the `core` repository, follow the extension documentation.

Note that DuckDB does not support reloading extensions. Therefore, if you experience any issues, try restarting DuckDB after updating the extensions.

Maximizing Your Delta Scan Performance in DuckDB

Publication date: 2025-03-21

Author: Sam Ansmink

TL;DR: We released a new version of the `delta` extension, which includes several new features and performance improvements. In this blog post, we'll put the `delta` extension through its paces with some benchmarks and take a deep dive into some of the new performance-related features.

Overview

In our previous post, we talked about what the Delta Lake table format is all about, and how DuckDB's `delta` extension leverages the Delta Kernel library to offer native support. In this blog post, we will focus on how to get the most performance out of reading Delta tables from DuckDB. We'll start with a small recap of Delta, then demonstrate the performance gains that were made over the past few releases. Finally, we demonstrate three key features available in the latest Delta release that will ensure you get the most out of your Delta reading performance: metadata caching, file skipping and partition information pushdown.

The Delta Open Table Format

Let's start off with a small recap of Delta to get back up to speed. Delta Lake is an Open Table Format similar to Apache Iceberg and Apache Hudi. Open Table Formats are best understood as "a collection of data and metadata files" that aim to provide the flexibility of data lakes while providing some of the consistency guarantees of traditional data warehouses. In the case of Delta, the format consists of Parquet files for data and a mix of Parquet, JSON and binary files for metadata. Besides providing an improved level of consistency, the additional metadata provided by Open Table Formats allows various types of performance optimizations through things like column statistics and file skipping. For a more in-depth explanation, we refer to the previous Delta blog post.

The `delta` Extension

DuckDB natively supports reading Delta Tables through the `delta` extension. This extension is one of the core DuckDB extensions with >70k weekly downloads. Using this extension to read from a Delta Table is really simple. Since DuckDB v1.2.0, the `delta` extension will be automatically installed upon first use and loaded when invoking the `delta_scan` function.

So for example, to read a local Delta table, simply open any DuckDB client and run:

```
SELECT * FROM delta_scan('./<path_to_your_delta_table>');
```

Is your Delta table on someone else's machine, perhaps in AWS? DuckDB can also query straight from S3! To have DuckDB automatically load your AWS credentials and query a remote Delta table, run:

```
CREATE SECRET (TYPE s3, PROVIDER credential_chain);
SELECT * FROM delta_scan('s3://<your-bucket>/<your_delta_table>');
```

For other cloud providers such as Azure or Google Cloud, check the extension's documentation page.

Performance Improvements between `delta` v0.1.0 and 0.3.0

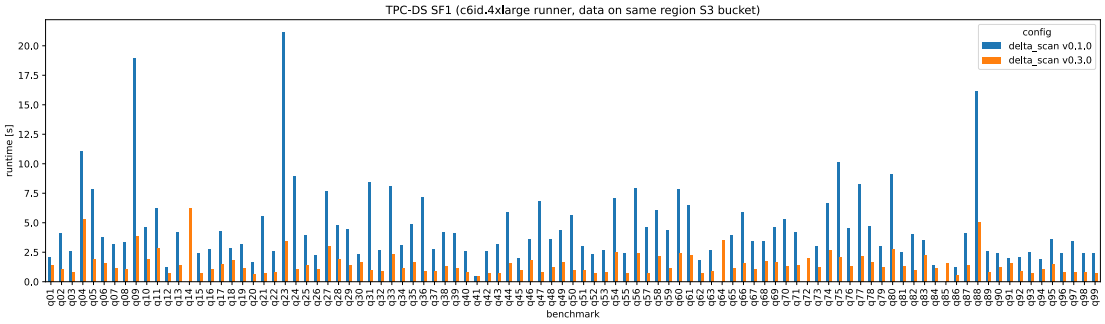
While the first release (v0.1.0) of the `delta` extension already came with various performance-related features such as projection pushdown and constant filter pushdown, the features added since then have massively improved the performance of `delta_scan`. To illustrate this, our first benchmark will use the industry standard TPC-DS benchmark with the scale factor 1 data set (SF1).

Benchmark Setup

For this benchmark, we started up an AWS c6id.4xlarge instance (16 vCPUs, 32 GB RAM) and wrote the TPC-DS SF1 dataset to an S3 bucket in the same region (eu-west-1) using PySpark. Each benchmark is run a total of 6 times with the result being the median runtime of the last 5 runs with the first being considered a cold run. The aggregated results are shown in the following table.

Result	Total runtime	Min runtime	Max runtime	Median runtime	Queries timed out
`delta` extension v0.1.0	444.76	0.48	21.31	3.63	4
`delta` extension v0.3.0	151.06	0.46	6.15	1.22	0

The detailed results of the benchmark are shown in the foldout: Detailed TPC-DS SF1 benchmark results, `delta` extension v0.1.0 vs. v0.3.0



Analysis

We can see from the results that there has been a significant performance improvement all around. In v0.1.0, 4 out of 99 queries hit the benchmark timeout of 30 sec and were excluded from the results. In v0.3.0, all 99 queries completed well within the timeout. Comparing total runtimes (excluding the queries that timed out for v0.1.0) we find a speedup of more than 3×!

Now, without going into too much detail, an important part of the speedup here can be attributed to the **cardinality information propagation** that was added in PR #77. Having accurate cardinality estimation is essential for DuckDB's query optimizer to work well and produce efficient query plans. Specifically, DuckDB's Join Optimizer uses the cardinality estimates to change the order in which joins are performed. The join order can massively impact the cardinality of the intermediate tuples, which has a big impact on query performance. Especially in workloads like the TPC-DS benchmark, which has many queries that contain a lot of joins, the Join Order Optimizer plays a crucial role. For (a lot) more details, check out this thesis.

Further Optimizations

Attaching Delta Tables

Besides the general performance improvements like cardinality information propagation, several performance-related features have been added to the `delta` extension as well. One of those is the ability to attach Delta tables. Using ATTACH to query a Delta table has multiple advantages. For starters, by using ATTACH, your queries can look a little cleaner when querying the same table multiple times since you don't have to repeat the full Delta table path every time it is queried. More importantly, though, using ATTACH will allow DuckDB to cache/reuse certain parts of the Delta metadata, which can improve query performance. To attach the local Delta table, run:

```
ATTACH './<path_to_your_delta_table>' AS <your_table> (TYPE delta);
```

After attaching the Delta table, you can query the table just by using the alias:

```
SELECT * FROM <your_table>;
```

By default, DuckDB will automatically cache Delta metadata within the same transaction. That means if a Delta table is scanned multiple times in that transaction, DuckDB can reuse parts of the Delta metadata between the different scans. For example, the following query will only read the Delta metadata once:

```
SELECT * FROM t1
UNION ALL
SELECT * FROM t1;
```

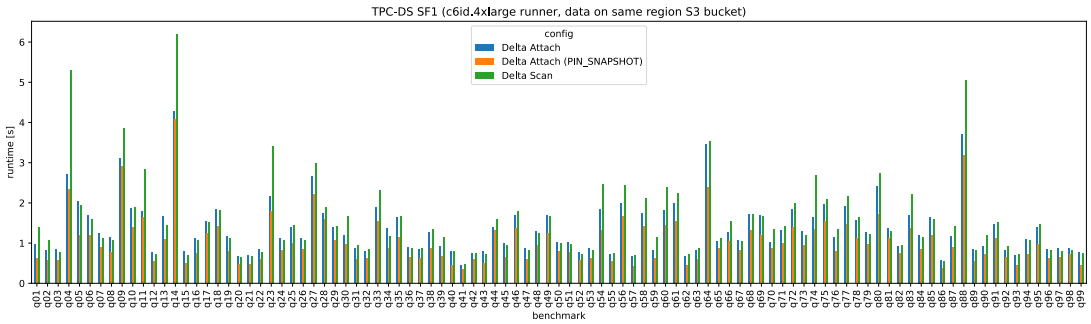
For even more performance improvements, DuckDB also supports **persisting this cached Delta metadata *between* different queries**. To do this, the Delta table can be attached using the `PIN_SNAPSHOT` option. With this option enabled, subsequent queries can reuse the metadata such as in the following code block:

```
ATTACH 's3://<your-bucket>/<your_delta_table>' AS t2 (
    TYPE delta,
    PIN_SNAPSHOT
);

-- First scan (metadata not yet cached)
SELECT * FROM t1;

-- Second scan (metadata is now cached)
SELECT * FROM t2;
```

Metadata caching can have a significant performance impact, especially in situations where the data is relatively small and the latency high. To illustrate this, we will rerun our TPC-DS experiment to compare three different ways of scanning the Delta table: using `delta_scan`, using `ATTACH`, and using `ATTACH ... (PIN_SNAPSHOT)`. The rest of the benchmark setup is identical to the one in the previous section.

result	total runtime	min runtime	max runtime	median runtime
`delta_scan`	151.06	0.46	6.15	1.22
`ATTACH`	134.26	0.43	4.28	1.19
`ATTACH (PIN_SNAPSHOT)`	102.80	0.36	4.04	0.87

The detailed results of the benchmark are shown in the foldout: Detailed TPC-DS SF1 benchmark results with different configurations



The results show that for many TPC-DS queries, using `ATTACH` instead of `delta_scan` can already slightly improve performance for several queries, with overall runtime seeing a 1.13× speedup. When the metadata is fully in cache due to `PIN_SNAPSHOT`, we see an even greater speedup of 1.47×. This, however, comes at the tradeoff of missing any updates to the table that occur after the `ATTACH` statement.

A keen eye looking at the full results will also spot a few cases where the `ATTACH` results are actually slightly worse than the results with raw `delta_scan`. This is something we will explain in the section on the pushdown / ATTACH interplay.

File Skipping

Another key performance feature of scanning Delta tables is file skipping. As explained in the introduction, Delta tables contain metadata that contains all sorts of statistics of the data files of the table. These statistics can be used by engines like DuckDB to decide which Parquet

files need to be scanned and which ones can be skipped altogether. File skipping is something that is done automatically by DuckDB. File skipping will work for both constant filters and dynamic filters (filters that are calculated during query execution):

```
-- constant filter
FROM delta_scan('...')
WHERE col_a > 'some_value';
-- dynamic filter
FROM delta_scan('...')
WHERE col_a > (SELECT max(col_z) FROM other_tbl);
```

In the previous benchmarks, file skipping has only a very limited effect. The overall data is simply not big enough and many queries actually touch large parts of the data anyway. However, when only a relatively small part of the data is touched in a query, file skipping can have a huge impact on performance. To demonstrate this, we will first generate some test data. We'll use the same PySpark-based test data generation script as before.

The table has 100 million rows and has a very basic schema with an incrementing `id` column of `INTEGER` type and a `value` column of VARCHAR type. If we query the data using DuckDB we will see something like:

```
FROM delta_scan('s3://<your-bucket>/<your_delta_table>');
```

id int64	value varchar
49950000	val-49950000
49950001	val-49950001
49950002	val-49950002
49950003	val-49950003
·	·
·	·
·	·
49996	val-49996
49997	val-49997
49998	val-49998
49999	val-49999

100000000 rows
(8 shown)

Now, let's say we are only interested in a specific range of `id`s: maybe we only want `id`s below 100. We will now construct two queries.

For the first query, we will directly read all the Parquet files stored in the table using a glob pattern:

```
FROM parquet_scan('s3://<your-bucket>/<your_delta_table>/*.parquet')
WHERE id < 100;
```

> We are doing this for illustrative purposes to show us the benefits of file skipping, scanning the raw Parquet files in a Delta table like this only works here because we have no updates, deletes or checkpoints in this table.

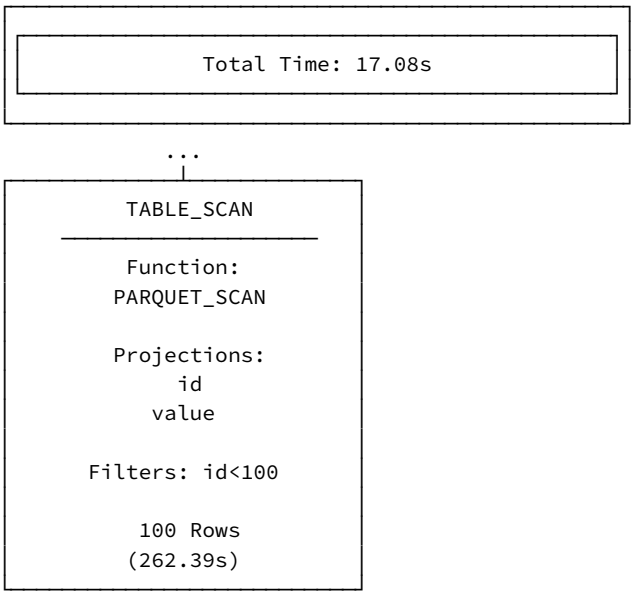
For the second query, we directly scan the table using the `delta_scan` table function, selecting only the `id`s we are interested in using a `WHERE` clause:

```
FROM delta_scan('s3://<your-bucket>/<your_delta_table>')
WHERE id < 100;
```

Now when running these queries from a c6id.4xlarge AWS instance on the S3 bucket in the same region, we can see that they perform wildly differently. The `delta_scan` only requires ≈0.5 seconds to complete, whereas the `parquet_scan` takes ≈17 seconds. So what's going on here exactly?

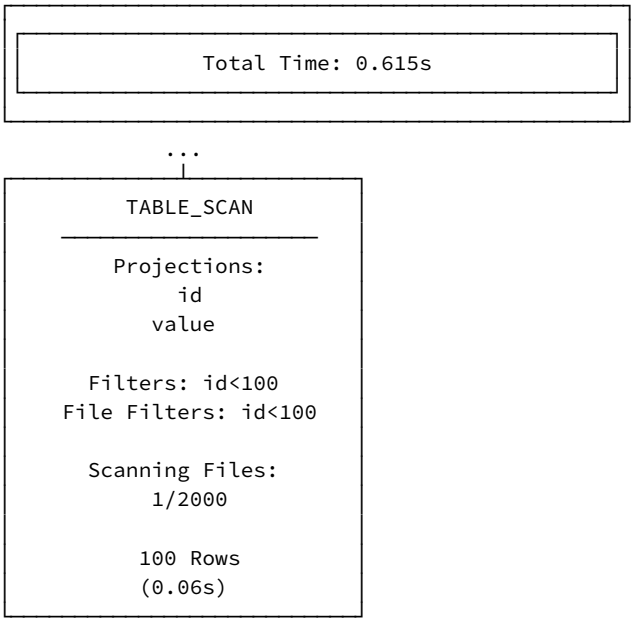
We can use DuckDB's `EXPLAIN ANALYZE` statement to get more details here. Let's start by analyzing the `parquet_scan`:

```
EXPLAIN ANALYZE
FROM parquet_scan('s3://<your-bucket>/<your_delta_table>/*.parquet')
WHERE id < 100;
```



We can see in the `EXPLAIN ANALYZE` output that our filter was properly pushed down, and the scan correctly only produced 100 rows. This all seems fine, right? Well, let's compare it with the output of `EXPLAIN ANALYZE` for the `delta_scan`:

```
EXPLAIN ANALYZE
FROM delta_scan('s3://<your-bucket>/<your_delta_table>');
```



For the `delta_scan` function's `EXPLAIN ANALYZE` output, we can see two new fields: `File Filters` and `Scanning Files`. This shows us clearly what's going on. The `id<100` predicate is now used for two things: it's pushed down into the scans on the individual Parquet files, just like the `parquet_scan`, but it also shows up as a file filter, which is used to reduce the list of files to be scanned altogether! This leads to a **2,000× reduction** of the amount of Parquet metadata to be read, which results in a huge performance boost.

Partition Information Pushdown

The final DuckDB Delta performance feature is partition information pushdown. Partition information pushdown and the partition-aware aggregation operator are relatively new features introduced in DuckDB v1.2.0. In the v0.3.0 release of the `delta` extension this feature was also added, which means that DuckDB can now use the partitioning information to create query plans that can utilize the fact that the data

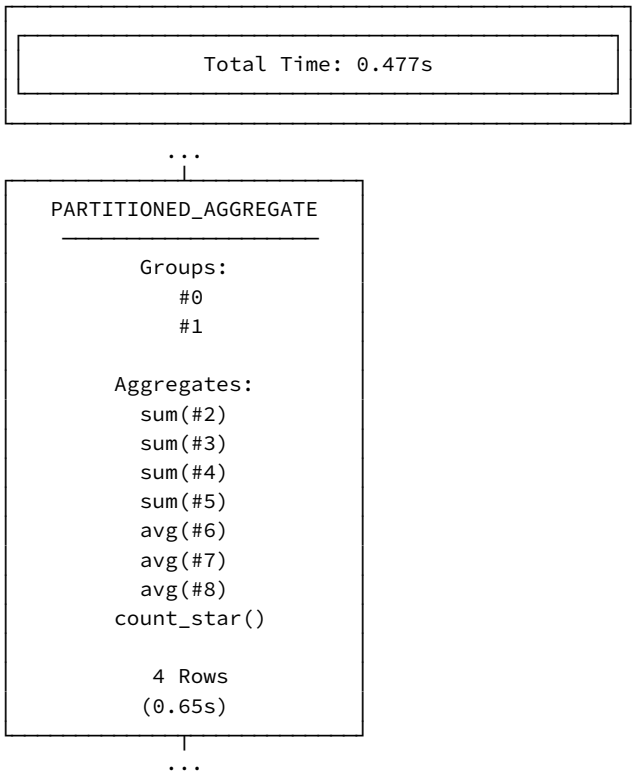
scanned is already partitioned. To show the performance benefit of partition information, we will, *surprise,* run another benchmark! This time, we chose the TPC-H dataset at scale factor 10 and ran the experiment on a 32 GB MacBook Pro M1 Max. We partitioned the `lineitem` table by the `l_returnflag` and `l_linestatus` columns. We then run Q1 which looks roughly like this:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) AS sum_qty,
    ...
FROM
    lineitem
    ...
GROUP BY
    l_returnflag,
    l_linestatus
    ...;
```

Note that the query contains a GROUP BY statement, which lists the exact columns by which our dataset is already partitioned. Making DuckDB use partitioning-aware operators is done all automatically, so in this case simply running:

```
ATTACH './<path_to_partitioned_directory>/lineitem_sf10' AS lineitem (
    TYPE delta
);
PRAGMA tpch(1);
```

will fire off the TPC-H Q1 on the partitioned Delta dataset. To inspect what's happening we will again use `EXPLAIN ANALYZE`:

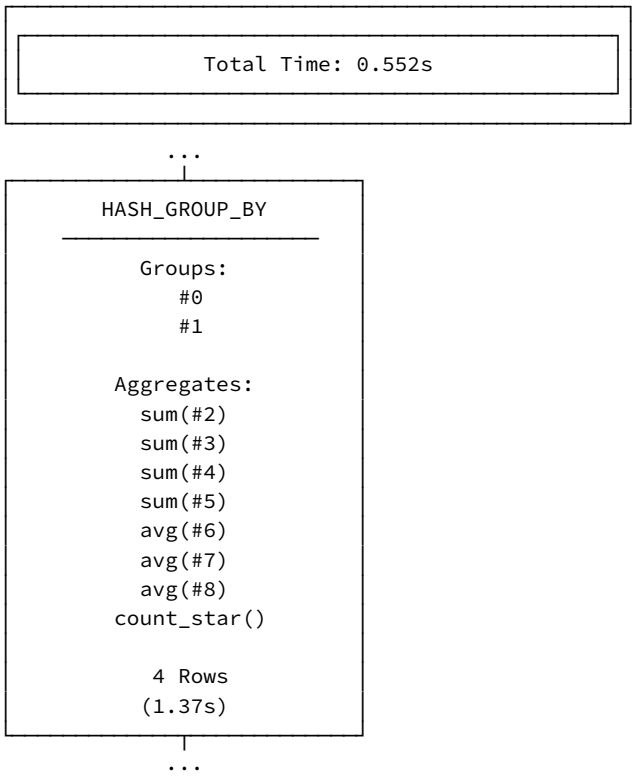


We can see that DuckDB has detected the partitioning information correctly and is using the `PARTITIONED_AGGREGATE` operator to do the GROUP BY efficiently.

Now, as a baseline, we will rerun the same query, but now with partition information pushdown disabled:

```
ATTACH './<path_to_partitioned_directory>/lineitem_sf10' AS lineitem (
    TYPE delta,
    PUSHDOWN_PARTITION_INFO 0
);
PRAGMA tpch(1);
```

Again, using the `EXPLAIN ANALYZE`, we can see that DuckDB will now use a regular `HASH_GROUP_BY` operator, since the partition information from Delta was not available during query planning.



Now looking at the performance differences between these two queries, we can see that the overall speedup is only a modest 1.16×, however the aggregation operation itself was sped up by 2.11×! This means that when queries that regularly do heavy group by operations, partitioning the data by these columns can definitely be a very useful tool to have in your performance tuning toolbox.

A Note on Pushdown / ATTACH Performance Interplay

While features such as filter pushdown and partition information pushdown will improve performance for many workloads, it is useful to be aware that there is a somewhat intricate interplay between the metadata caching mechanism from using `ATTACH`, and the pushdown of filters and partition information. At the end of the section on the `ATTACH` feature, we already saw that for some queries, using `ATTACH` is actually slightly slower than using the raw `delta_scan`. Without diving into too much detail, pushing down filters and partitioning information can negatively affect the effectiveness of metadata caching for some queries. This means that on some queries, you may, somewhat counterintuitively, benefit from partially disabling filter pushdown when using `ATTACH`:

```
ATTACH './<your_delta_table_directory>' AS dt (
    TYPE delta,
    PIN_SNAPSHOT,
    PUSHDOWN_PARTITION_INFO 0,
    PUSHDOWN_FILTERS 'none'
);
```

This should be considered an advanced use case, though, and only relevant when optimizing for specific queries. The default settings of `ATTACH` should provide the best overall performance and are recommended for most cases. Furthermore, there is ongoing work in the underlying `delta-kernel-rs` library used by DuckDB Delta that aims to reduce this effect by exposing mechanisms to cleverly refresh metadata objects held by DuckDB. As soon as these mechanisms are available, we will add them to the DuckDB `delta` extension, likely making these flags obsolete for anything but testing.

Conclusion

In this blog post, we've taken a look at the latest version of the DuckDB `delta` extension and put it to the test with some benchmarks. We've run queries from the industry standard TPC benchmarks to demonstrate the large performance improvements that were made over the past releases of the `delta` extension.

Furthermore, we've taken a look at three specific techniques that can be used while working with Delta tables to improve performance even further:

- Using metadata caching with ATTACH
- Using filters and data layout to reduce the number of files that need to be scanned
- Utilizing partitioning information to speed up aggregations

All in all, we think with the v0.3.0 release of the `delta` extension, DuckDB can read Delta tables with excellent performance for many different workloads and highly encourage everyone to give the latest version a shot!

Using DuckDB in Streamlit

Publication date: 2025-03-28

Author: Petrica Leuca

TL;DR: We use a real-world railway dataset to demonstrate the integration of DuckDB and Streamlit, including the database connection management, the DuckDB Python relational API and responsiveness in interactive map charts.

Introduction

In the article Analyzing Railway Traffic in the Netherlands, Gábor has shown how to analyze data with SQL and reading remote files with DuckDB, by using open data from the Rijden de Treinen *(Are the trains running?)* application.

Using the above open data, in this post we will build an application, in which a user can:

- Understand the railway traffic data;
- Visualize the railway network traffic in 2024, by using Plotly;
- Visualize the five closest train stations on any point selected on the map, by using the Streamlit Folium component.

The application will be developed with Streamlit, an open source framework in which one can implement data applications easily with Python. The application code is available on GitHub.

Connecting to DuckDB in Streamlit

To load the railway traffic data, we write a function, which creates the two tables we will be working with, `services` and `stations`:

```
def prepare_duckdb(duckdb_conn):

    duckdb_conn.sql("""
        create table if not exists services as
        from 'https://blobs.duckdb.org/nl-railway/services-2024.csv.gz'
    """)

    duckdb_conn.sql("""
        create table if not exists stations as
        from 'https://blobs.duckdb.org/nl-railway/stations-2023-09.csv'
    """)
```

From a Streamlit application there are three ways to connect to DuckDB:

- In-memory connection;
- Persisted local file connection;
- Attaching an external database.

In-Memory Connection

Connecting to an in-memory database involves loading the data into memory, whenever a DuckDB connection is established. Depending on the resources of the application server and the amount of data, creating a new in-memory DuckDB connection and loading the required data on every request will slow down the application. In Streamlit this behavior is addressed by caching the resource, which can be done as:

- caching the DuckDB connection as a global shared connection in the application;
- caching the DuckDB connection for the user session.

```
@st.cache_resource(ttl=datetime.timedelta(hours=1), max_entries=2)
def get_duckdb_memory(session_id):
    """
    Set a caching resource which will be refreshed
     - either at each hour
     - either at each third call
     - either when the connection is established for a new session_id
    """

    duckdb_conn = duckdb.connect()
    prepare_duckdb(duckdb_conn=duckdb_conn)

    return duckdb_conn
```

Persisted Local File Connection

A DuckDB connection can be established against a persisted database file on the server of the application. The database file can be created:

- during the deployment, before the application starts;
- a standalone process, which creates the database and copies it to the application server. Depending on the freshness requirement of the application, one might schedule a process to refresh the data.

```
duckdb_conn = duckdb.connect(
    "train_stations_and_services.duckdb",
    read_only=True
)
```

Attaching an External Database

Another way to connect to DuckDB is to establish a **read only** connection to an instance over HTTPS or S3 compatible APIs, e.g., DUCKDB_EXTERNAL_LOCATION = "https://blobs.duckdb.org/nl-railway/train_stations_and_services.duckdb":

```
duckdb_conn = duckdb.connect()
duckdb_conn.execute(f"attach '{DUCKDB_EXTERNAL_LOCATION}' as ext_db")
duckdb_conn.execute("use ext_db")
```

More details about attaching an external database can be found in the DuckDB documentation.

Considerations

1. DuckDB works best if you have allocated 1-4GB of memory per thread, therefore configure the appropriate resources when deploying the Streamlit application.

2. The recommendation is to use a new connection for each database interaction or at least one at user session level, and not a global connection at application level. Using a global connection might result in unexpected behaviour, depending on the number of users using the application at the same time.

3. When using a persisted local database, take into account that when there is a database connection established no other process can write to it. One workaround is to have a process which creates and replaces the database file on the server.

4. When using an attached database, take into account that the application can have a performance impact depending on the storage solution chosen.

Starting from this point, we will work with the external database.

Analyzing Dutch Railway Data

To analyze the data within Streamlit, we will make use of the Python relational API, by chaining queries. We start by defining a relation, which will select the needed data from `stations`:

```
stations_selection = duckdb_conn.sql("""
    select name_long as station_name, geo_lat, geo_lng, code
    from stations
""").set_alias("stations_selection")
```

Then we define a relation for the `services` selection:

```
services_selection = (
    duckdb_conn.sql("from services")
    .aggregate("""
        station_code: "Stop:Station code",
        service_date: "Service:Date",
        service_date_format: strftime(service_date, '%d-%b (%A)'),
        num_services: count(*)
    """)
    .set_alias("services")
)
```

We then join the two relations to select the station name, the geolocation and other details we will be using further, bringing the above code together in a function.

```
def get_stations_services_query(duckdb_conn):

    # create a relation for the station selection
    stations_selection = ...

    # create a relation for the services selection
    services_selection = ...

    # return the query and the duckdb_conn
    return (
        stations_selection
        .join(
            services_selection,
            "services.station_code = stations_selection.code"
        )
        .select("""
            service_date,
            service_date_format,
            station_name,
            geo_lat,
            geo_lng,
            num_services
        """)
    ), duckdb_conn
```

When we run `get_stations_services_query(get_duckdb_conn())`, there is yet no data retrieved from the database. This is because the query evaluation is lazy, which means that the query is not executed against the database until an execution command is encountered.

The query will be executed when one of the following methods is encountered:

- `stations_query.df()` to extract to a Pandas dataframe;
- `stations_query.fetchall()` to extract in a list;
- `stations_query.write_to()` to export the data in a file;
- any other calculation method, such as `.sum`, `.row_number` etc. The relational methods can be found in the DuckDB documentation.

Top 5 Busiest Train Stations per Month

To extract the 5 busiest train stations per month, we start from the above query and add a `row_number` calculation with a final filter of the ranking to be less or equal to 5:

```
stations_query, _ = get_stations_services_query(get_duckdb_conn())

top_5_query = (
    stations_query.aggregate("""
            station_name,
            service_month: monthname(service_date),
            service_month_id: month(service_date),
            num_services: sum(num_services)
        """)
    .select("""
            station_name,
            service_month,
            service_month_id,
            num_services,
            rn: row_number()
                over (
                    partition by service_month
                    order by num_services desc
                )
        """)
    .filter("rn <= 5")
    .order("service_month_id, station_name")
)
```

Streamlit provides a series of charts, such as line and bar, but, unfortunately, they do not offer many personalization possibilities. Therefore, the above chart was generated with Plotly, an open source graphing library, integrated with Streamlit by calling `st.plotly_chart`.

Railway Network Utilization Analysis

The data about the train rides through the train stations of The Netherlands, at day level, is a perfect input for analyzing the utilization of the railway network across the country and during the year.

Railway Network Utilization across the Country To analyze the network utilization across the country, we use `density_map` chart from Plotly, which will generate a heat map on a map chart. Because heat maps work best with pre-aggregated data, we first aggregate the number of train services and the geolocation of the station:

```
stations_df = stations_query.aggregate(
    "geo_lat, geo_lng, num_services: sum(num_services)"
).df()
```

It is interesting to observe that the north-east of the country is barely covered by railway and that, while Groningen and The Hague appear to be used during the year, neither of their train stations was in the top 5 busiest train stations per month.

Animating the Heat Map From the above static chart, we can easily generate an animation with Plotly, by providing the animation frame parameter. In our case, we generate an animation with the railway network utilization at service day level during July 2024:

Railway Network Utilization during the Year From a year point of view, we are curious to discover how busy the network is in each month and each day of the week, which is better displayed through a heat map. We prepare the input for the chart, by pivoting the number of services by the month name of the service date and by grouping by the ISO day name of the week. At the moment, there is no pivot method in the Python relational API, but we can use the SQL query generated by the relational API as source in the pivot raw SQL:

```
@st.cache_data(ttl=3600, max_entries=100)

def get_stations_services_data(_duckdb_conn):
    query = _duckdb_conn.sql("from services").aggregate("""
            service_day: dayname("Service:Date"),
            service_day_isodow: isodow("Service:Date"),
            service_month: monthname("Service:Date"),
            num_services: count(distinct "Service:RDT-ID")
        """)

    return (
        _duckdb_conn.sql(f"""
            pivot ({query.sql_query()})
            on service_month
            using sum(num_services)
            group by service_day, service_day_isodow
            order by service_day_isodow
        """)
        .select(
            "January",
            "February",
            "March",
            "April",
            "May",
            "June",
            "July",
            "August",
            "September",
            "October",
            "November",
            "December",
            "service_day",
        )
        .df()
    )
```

We have decided to create a function, in order to highlight `cache_data` functionality in Streamlit. While the above query is executed in 300 ms on a MacBook Pro with 12 GB memory, we want to cache the results because the result of the query does not change often. `cache_data` is used in Streamlit to cache the results of a function and just as `cache_resource`, it has the attributes time to live and max entries. The two caching mechanisms require that the input parameters passed to the function to be hashable. Since the DuckDB connection object is not hashable, we can tell Streamlit to ignore it by adding the prefix _.

> It is important to notice that the order of columns and values matters in a heat map chart and that we need to reset the index of the pivoted data before plotting.

The code of implementing interactive maps with Plotly in Streamlit is available on GitHub.

Finding the Closest 5 Stations with Folium

One other use-case for the railway network data is to find the closest train stations to a location selected by the user on the map. While Streamlit and Plotly have the functionality to register click events on the chart, they work only when the user is selecting points which are part of the chart data. Therefore, we will use `streamlit-folium`, a Streamlit component wrapped around Folium, which can register the clicks on any point of the map.

To work with geo data in DuckDB we have to install and load the spatial extension:

```
duckdb_conn.sql("install spatial;")
duckdb_conn.sql("load spatial;")
```

We then define a function to retrieve the query, which returns the closest five train stations to a location, composed of latitude and longitude.

```
def get_closest_stations_query(duckdb_conn, lat, lng):

    stations_selection = duckdb_conn.sql("""
        select name_long as station_name, geo_lat, geo_lng, code
        from stations st
        where exists (
                select count(*)
                from services sv
                where st.code = sv."Stop:Station code"
                having count(*) > 100
            )
    """)

    return (
        stations_selection.project(f"""
            code as station_code,
            station_name,
            geo_lat,
            geo_lng,
            station_geo_point: st_point(geo_lng, geo_lat),
            clicked_geo_point: st_point({lng}, {lat}),
            distance_in_m: st_distance_sphere(
                            st_point(geo_lng, geo_lat),
                            clicked_geo_point
                        ),
            distance_in_km: round(distance_in_m/1000,2)
        """)
        .order("distance_in_km")
        .limit(5)
    )
```

In the above query we use `st_point` to create a point type field and `st_distance_sphere` to get the distance, in meters, between two points.

> It is important to mention that while we usually refer to a geo point as `[latitude, longitude]`, when working with spatial extensions we usually create the point as `[longitude, latitude]`. This applies to PostGIS as well.

When a user clicks on the map, we store the clicks in the session state and rerun Streamlit to display the map with the new user selection. When rerun happens, Streamlit will rerun the entire application for the current session, therefore understanding the session state and caching mechanisms is essential when building Streamlit applications.

The code of implementing interactive maps with Folium in Streamlit is available on GitHub.

Conclusion

In this post, we illustrated how DuckDB can be used in Streamlit, how the Python relational API can make our code cleaner by chaining queries and how to implement interactive maps with Plotly and Folium, while leveraging the Streamlit caching mechanism.

Fully Local Data Transformation with dbt and DuckDB

Publication date: 2025-04-04

Author: Petrica Leuca

TL;DR: In this post, we implement data transformation and reverse ETL pipelines with DuckDB and dbt using the `dbt-duckdb` adapter.

Introduction

The Data Build Tool, `dbt`, is an open-source transformation framework, which enables data teams to adopt software engineering best practices in the code they deliver, such as Git workflow and unit testing. Other notable features of `dbt` are data lineage, documentation and data testing as part of the execution pipeline.

In this article we will demonstrate how to perform fully local data transformations with dbt and DuckDB. To this end, we'll use the `dbt-duckdb` adapter, which integrates the `dbt` standard and with DuckDB's processing power.

Data Model

We use two open datasets in this post: railway services, provided by the team behind the Rijden de Treinen *(Are the trains running?)* application and cartography information about the Netherlands provided by cartomap. The datasets are organized into:

- a persisted DuckDB database, in which we store the railway services data from 2024, hosted on Cloudflare;
- a provinces GeoJSON file, containing the geographic information about the Dutch provinces, hosted on GitHub;
- a municipalities GeoJSON file, containing the geographic information about Dutch municipalities, stored together with the code.

After doing an initial exploration of the above data, we can observe that a province can have one or many municipalities, a municipality can have none or many train stations and a train service record is connected to one and only one train station. Therefore we decide on the following data model:

- a dimension table, `dim_nl_provinces`, to hold information about the Dutch provinces;
- a dimension table, `dim_nl_municipalities`, to hold information about the Dutch municipalities, linked to `dim_nl_provinces`;
- a dimension table, `dim_nl_train_stations`, to hold information about the train stations in the Netherlands, linked to `dim_nl_municipalities`;
- a fact table, `fact_services`, to hold information about the train services, linked to `dim_nl_train_stations`.

The purpose of processing the train services and the Netherlands cartography data is to organize the data in a structure which can easily be used in future railway data analysis use cases. Such curated data structures are often called *data marts*.

> **Tip.** The following code is available on GitHub.

Processing Data with DuckDB and dbt

After we have initialized our project, we configure the connection details for DuckDB in the `profiles.yml` file. Along with specifying if the database should be in memory or persisted to disk, we also specify:

- which extensions are required for the data processing, e.g., spatial;
- external databases, attached from the local disk or other storage solutions.

```yaml
dutch_railway_network:

  outputs:
    dev:
      type: duckdb
      path: data/dutch_railway_network.duckdb
      extensions:
        - spatial
        - httpfs
      threads: 5
      attach:
        - path: 'https://blobs.duckdb.org/nl-railway/train_stations_and_services.duckdb'
          type: duckdb
          alias: external_db
  target: dev
```

We then configure the `sources.yml` file under the `models` directory by specifying external sources (such as files) and table definitions from the attached database(s):

```yaml
version: 2
sources:
  - name: geojson_external
    tables:
      - name: nl_provinces
        config:
          external_location: "https://cartomap.github.io/nl/wgs84/provincie_2025.geojson"
      - name: nl_municipalities
        config:
          external_location: "seeds/gemeente_2025.geojson"
  - name: external_db
    database: external_db
    schema: main
    tables:
      - name: stations
      - name: services
```

> The `external_location` can point to CSV, Parquet or JSON files. Both the local file system and remote endpoints (e.g., HTTP or S3) are supported.

With both the profile and source defined, we can now load the data.

Loading Data to DuckDB

In `dbt` the way of storing the data in the target system is called `materialization`. The `dbt-duckdb` adapter provides the following materialization options:

- `table`, replacing the target table at each run;
- `incremental`, with append and `delete+insert` options, modifying the data in the table, but not the table itself (if it exists);
- `snapshot`, implementing a Slowly Changing Dimension Type 2 table with time validity intervals;
- `view`, replacing the target view at each run.

Another feature of using the above adapter is that the SQL dialect used in the data processing scripts has all the friendly SQL extensions DuckDB has to offer.

To refresh the data in the `dim_nl_provinces` table, we use the `st_read` spatial function, which reads and parses automatically the `nl_provinces` GeoJSON file, defined in `sources.yml`.

```
{{ config(materialized='table') }}

SELECT
    {{ dbt_utils.generate_surrogate_key(['id']) }} AS province_sk,
    id AS province_id,
```

```
    statnaam AS province_name,
    geom AS province_geometry,
    {{ common_columns() }}
FROM st_read({{ source("geojson_external", "nl_provinces") }}) AS src;
```

Similarly, we fully refresh the data for `dim_nl_municipalities` and `fact_services`.

To build the relation between a train station location and a municipality, we use the `st_contains` spatial function, which returns `true` when a geometry contains another geometry:

```
{{ config(materialized='table') }}

SELECT
    {{ dbt_utils.generate_surrogate_key(['tr_st.code']) }} AS station_sk,
    tr_st.id AS station_id,
    tr_st.code AS station_code,
    tr_st.name_long AS station_name,
    tr_st.type AS station_type,
    st_point(tr_st.geo_lng, tr_st.geo_lat) AS station_geo_location,
    coalesce(dim_mun.municipality_sk, 'unknown') AS municipality_sk,
    {{ common_columns() }}
FROM {{ source("external_db", "stations") }} AS tr_st
LEFT JOIN {{ ref ("dim_nl_municipalities") }} AS dim_mun
       ON st_contains(
           dim_mun.municipality_geometry,
           st_point(tr_st.geo_lng, tr_st.geo_lat)
       )
WHERE tr_st.country = 'NL';
```

> To read from the external sources, we reference the source by providing the source and table names.

Exporting Data from DuckDB

One major benefit of using DuckDB for data processing is the ability to export data to files (such as CSV, JSON and Parquet) and to refresh data directly in PostgreSQL or MySQL databases.

External Files

The feature of exporting data to files is enabled by the `dbt-duckdb` adapter with the `external` materialization. With the `external` materialization, we are able to export data to CSV, JSON and Parquet file types to a specified storage location (local or external). The load type is `full refresh`, therefore existing files are overwritten.

In the following processing step, we export aggregated train service data at month level, to a Parquet file, partitioned by year and month:

```
{{
    config(
        materialized='external',
        location="data/exports/nl_train_services_aggregate",
        options={
            "partition_by": "service_year, service_month",
            "overwrite": True
        }
    )
}}

SELECT
    year(service_date) AS service_year,
    month(service_date) AS service_month,
    service_type,
    service_company,
    tr_st.station_sk,
    tr_st.station_name,
```

```
    m.municipality_sk,
    m.municipality_name,
    p.province_sk,
    p.province_name,
    count(*) AS number_of_rides
FROM {{ ref ("fact_services") }} AS srv
INNER JOIN {{ ref("dim_nl_train_stations") }} AS tr_st
        ON srv.station_sk = tr_st.station_sk
INNER JOIN {{ ref("dim_nl_municipalities") }} AS m
        ON tr_st.municipality_sk = m.municipality_sk
INNER JOIN {{ ref("dim_nl_provinces") }} AS p
        ON m.province_sk = p.province_sk
WHERE service_year = {{ var('execution_year') }}
GROUP BY ALL
```

The exported files are placed in Hive partitioned directory structure.

```
./service_year=2024/service_month=1:
49255 Apr  2 14:54 data_0.parquet

...

./service_year=2024/service_month=12:
48031 Apr  2 14:54 data_0.parquet
```

PostgreSQL

After we process the data into our railway services data mart, we can generate from it a daily aggregate at train station level, organized into a star schema model, by having the dimension keys part of the data:

```
{{
    config(
        materialized='incremental',
        incremental_strategy='delete+insert',
        unique_key="""
            service_date,
            service_type,
            service_company,
            station_sk
        """
    )
}}

SELECT
    service_date,
    service_type,
    service_company,
    srv.station_sk,
    mn.municipality_sk,
    province_sk,
    count(*) AS number_of_rides,
    {{ common_columns() }}
FROM {{ ref ("fact_services") }} AS srv
INNER JOIN {{ ref("rep_dim_nl_train_stations") }} AS tr_st
        ON srv.station_sk = tr_st.station_sk
INNER JOIN {{ ref("rep_dim_nl_municipalities") }} AS mn
        ON tr_st.municipality_sk = mn.municipality_sk
WHERE NOT service_arrival_cancelled

  {% if is_incremental() %}
    AND srv.invocation_id = (
        SELECT invocation_id
        FROM {{ ref("fact_services") }}
        ORDER BY last_updated_dt DESC
```

```
        LIMIT 1
    )
  {% endif %}
GROUP BY ALL
```

Thanks to DuckDB's ability to connect and write to a PostgreSQL database, we can add the above processing step to our `dbt` project, under the directory `models/reverse_etl`.

To connect to a PostgreSQL database, we need to specify in `profiles.yml`:

- the `postgres` extension;
- the PostgreSQL connection string in the `attach` section.

```
dutch_railway_network:

  outputs:
    dev:
      type: duckdb
      path: data/dutch_railway_network.duckdb
      extensions:
        - ...
        - postgres
      threads: 5
      attach:
        - ...
        - path: "postgresql://postgres:{{ env_var('DBT_DUCKDB_PG_PWD') }}@localhost:5466/postgres"
          type: postgres
          alias: postgres_db
  target: dev
```

We also have to configure the model's database details in `dbt_project.yml`:

```
models:
  dutch_railway_network:
    transformation:
      schema: main
      +docs:
        node_color: 'silver'
    reverse_etl:
      database: postgres_db
      schema: public
      +docs:
        node_color: '#d5b85a'
```

With this configuration, all the models from the `transformation` directory will be executed on the `main_main` schema, while the models from `reverse_etl` will be executed on the `main_public` schema.

After executing the models with `dbt run --model +reverse_etl`, the data is available to query from PostgreSQL:

```
psql -U postgres

SELECT count(*), sum(number_of_rides)
FROM main_public.rep_fact_train_services_daily_agg;

 count  |   sum
--------+----------
 240826 | 17438151
```

> It is important to mention that while PostgreSQL is the target database and `dbt` provides `merge` as an incremental strategy for it; the execution of the above pipeline is happening in DuckDB, therefore the incremental load can be done only with `append` or `delete+insert` strategies.

Execution Details

The above implementation consists of 10 models and 20 data tests, processing 400 MB of data from the attached DuckDB database, together with small data organized in GeoJSON files. The total execution time, on a single thread and on a MacBook Pro of 12 GB, is between 40 and 45 seconds. From the total execution time, approximately 30 seconds are spent to process the train services data and 4 seconds to write the aggregated data to PostgreSQL:

```
05:48:07  Running with dbt=1.9.3
05:48:08  Registered adapter: duckdb=1.9.2
05:48:08  Found 10 models, 20 data tests, 4 sources, 565 macros
05:48:08
05:48:08  Concurrency: 1 threads (target='dev')
...
05:48:45  19 of 30 OK created sql table model main_main.fact_services ..................... [OK in 32.60s]
...
05:48:50  26 of 30 OK created sql incremental model postgres_db.main_public.rep_fact_train_services_daily_
agg  [OK in 3.74s]
...
05:48:51  Finished running 2 external models, 1 incremental model, 7 table models, 20 data tests in 0 hours
0 minutes and 42.63 seconds (42.63s).
05:48:51
05:48:51  Completed successfully
05:48:51
05:48:51  Done. PASS=30 WARN=0 ERROR=0 SKIP=0 TOTAL=30
```

Conclusion

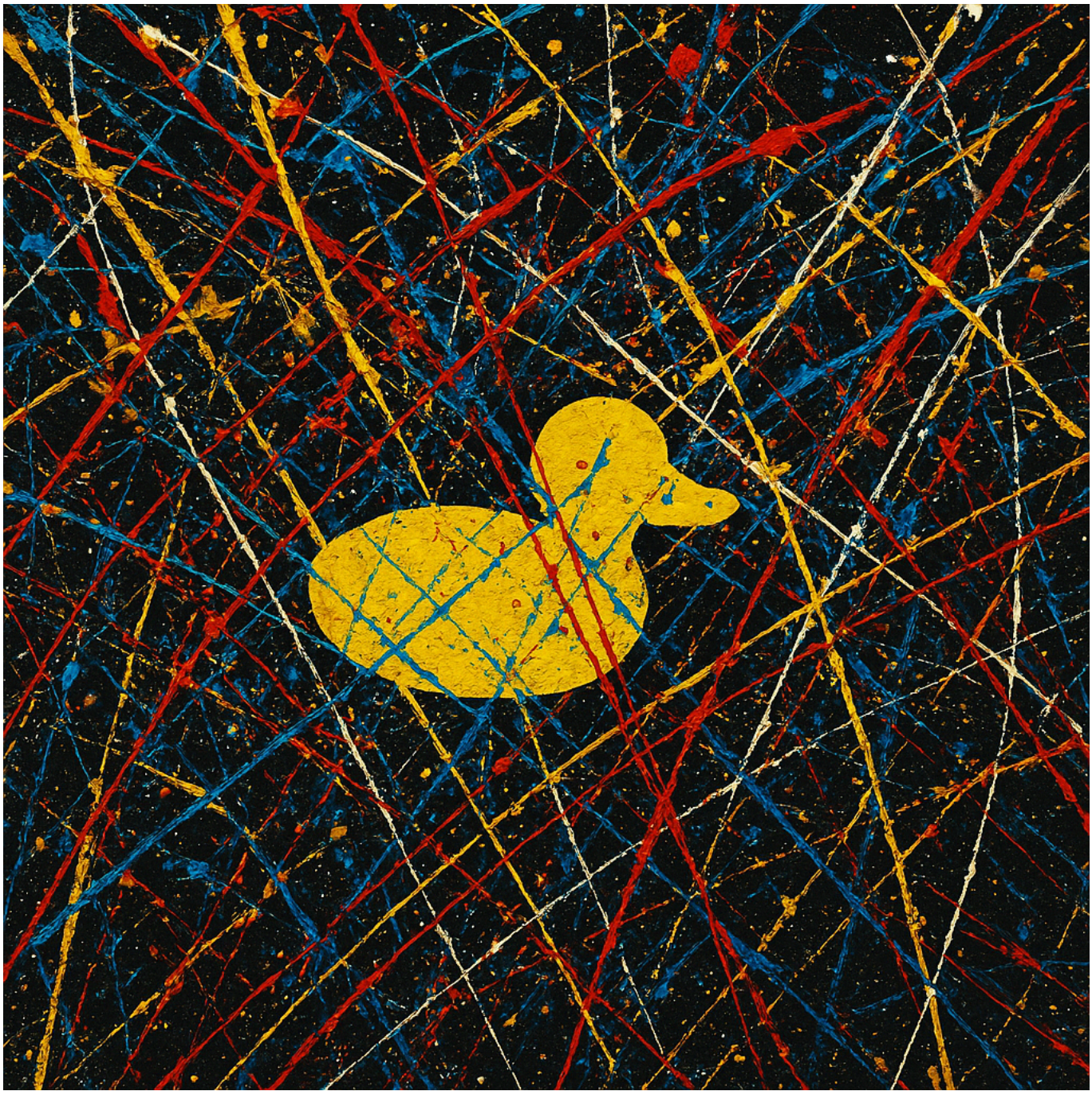
In this post we have demonstrated how DuckDB integrates with `dbt` and how it is part of the data processing ecosystem by showcasing data mart creation, file exports and reverse ETL to a PostgreSQL database.

DuckDB's CSV Reader and the Pollock Robustness Benchmark: Into the CSV Abyss

Publication date: 2025-04-16

Authors: Pedro Holanda, Gábor Szárnyas, Gerardo Vitagliano

TL;DR: DuckDB ships with a fast and robust CSV reader, which – we believe – can consume most CSV files found in the wild. To empirically evaluate this, we used the Pollock Benchmark, a state-of-the-art test suite designed to measure how well CSV readers can operate on non-standard files, and found that DuckDB ranks #1.



The Sorry State of the CSV Landscape

It is a well-known fact that CSV files come in all shapes and forms. While there is a well-defined standard, it is common for systems to export data without following even the basic rules of CSV construction. For example, Rabobank, one of the biggest banks in the Netherlands, exports the financial data of its clients with unescaped quotes in quoted values.

If a big financial institute cannot write proper CSV files, imagine the errors occurring in CSV files exported by decades-old legacy software, in hand-written CSV files, files exported from spreadsheet software, and so on! These sorts of errors are so prevalent that spreadsheet software, dataframe libraries and specialized CSV tools go above and beyond to salvage these files and hammer them into tables that users can work with.

DuckDB's CSV Parser

When we designed the DuckDB CSV parser, we strived to ensure that it's not only easy to use and fast, but also extremely reliable – to the point it can read most CSVs thrown at it. In this blog post, we study this from two angles. First, we explore the main options that can help you read non-standard CSV files and show their effects on parsing with some simple examples. Second, we evaluate how robust DuckDB's CSV parser is by putting it to the test with the Pollock Benchmark.

> There is a lot of discourse on the internet about whether a system should go above and beyond to read files that do not follow the standard. And yes, non-standard files are frustrating. We can all complain about them on the internet or decide to ignore them. But we strongly believe that ignoring reality comes at a cost. Being able to read non-standard files – even if it might result in ambiguity – should be the user's decision.

Running Example

As our running example, we'll use a CSV file called `cafes.csv`:

```
ZIP,Name,Country
1014,"Cafe ""Gedoogt""",Netherlands
1015,"Cafe "De Tuin" Amsterdam",Netherlands
1095,Joost
1872,Cafe Gezellig,Netherlands,5
```

Despite its small size, the file contains many different errors. Let's inspect it line by line:

Line	Row	Comment
1	`ZIP,Name,Country`	The header of the file.
2	`1014,"Cafe ""Gedoogt""",Netherlands`	A well-defined line, with a quoted value that contains escaped quotes ("").
3	`1015,"Cafe "De Tuin" Amsterdam",Netherlands`	The value for `Name` contains unescaped quotes.
4	`1095,Joost`	This line is missing the value for the `Country`.
5	`1872,Cafe Gezellig,Netherlands,5`	Has an extra column with the value `5`.

Loading the Example Naïvely

If we naïvely try to read our example, we get a table with only a single line:

```
FROM read_csv('cafes.csv');
```

column0 int64	column1 varchar	column2 varchar	column3 int64
1872	Cafe Gezellig	Netherlands	5

We can see that the reader ignored all lines except the last one. Let's query the CSV sniffer to see the dialect it detected:

```
SELECT Delimiter, Quote, Escape, SkipRows
FROM sniff_csv('cafes.csv');
```

Delimiter varchar	Quote int16	Escape varchar	SkipRows uint16
,	\0	\0	4

We can see that the sniffer decided that quotes and escape characters are not present in the file (represented as `\0` values). Consequently, it decided to skip the first 4 rows (including the header) in order to read the file according to the dialect.

> **Tip.** DuckDB allows users to retrieve all errors that occur when reading a CSV file using the `reject_errors` table. This is beyond the scope of this blog post, but you can find more details in the "Reading Faulty CSV Files" documentation page.

Loading the Example with a User-Defined CSV Dialect

Let's attempt to read the faulty CSV files with a user-defined CSV dialect. To this end, we'll set the `auto_detect` option to `false` and manually specify the presence of the `header`, the values of the `delim`(iter), `quote`, and `escape` options, as well as the schema with the `columns` option.

```
FROM read_csv('cafes.csv',
    auto_detect = false,
    header = true,
    delim = ',',
    quote = '"',
    escape = '"',
    columns = {'ZIP': 'INT16', 'Name': 'VARCHAR', 'Country': 'VARCHAR'}
);
```

Running this SQL statement will result in an error due to unescaped quotes in line 3: the parser can't determine whether the first quote in `"De Tuin"` marks the end of the quoted value or not:

```
Invalid Input Error:
CSV Error on Line: 3
Original Line: 1015,"Cafe "De Tuin" Amsterdam",Netherlands
Value with unterminated quote found.

Possible fixes:
* Disable the parser's strict mode (strict_mode=false) to allow reading rows that do not comply with the
CSV standard.
* Enable ignore errors (ignore_errors=true) to skip this row
* Set quote to empty or to a different value (e.g., quote='')
```

The error message informs us that there was a value with an *unterminated quote,* caused by the unescaped quote character. It also suggests some possible fixes, which will guide our attempts in the next sections.

Disabling Strict Mode

As the error message informed us, one of the ways to relax the constraints of the DuckDB CSV reader to read this file is by utilizing the `strict_mode` option. Namely, disabling the `strict_mode` option will allow the CSV Reader to go over the following common errors:

1. Unescaped quoted values, such as the one in line 3.
2. Rows that have too many columns. For example, for the row `1872,Cafe Gezellig,Netherlands,5`, the last column is ignored to fit our schema of three columns.
3. A mix of newline delimiters, e.g., files having both `\n` and `\r\n` as newline delimiters. This error does not occur in our running example.

By default, `strict_mode` is set to `true`, which implies that DuckDB will not attempt to parse rows that do not fit the dialect. However, setting it to `false` allows the CSV reader to attempt to read the file even if it cannot be correctly read under the given configuration. Let's go back to our example and turn off strict mode:

```
FROM read_csv('cafes.csv',
    auto_detect = false,
    header = true,
    strict_mode = false,
    delim = ',',
    quote = '"',
    escape = '"',
    columns = {'ZIP': 'INT16', 'Name': 'VARCHAR', 'Country': 'VARCHAR'}
);
```

We can see that the reader now only errors at line 4, indicating the previous lines were correctly read up to that point:

```
Invalid Input Error:
CSV Error on Line: 4
Original Line: 1095,Joost
Expected Number of Columns: 3 Found: 2
Possible fixes:
* Enable null padding (null_padding=true) to replace missing values with NULL
* Enable ignore errors (ignore_errors=true) to skip this row
```

> It is important to note two things for the `strict_mode = false` option:
>
> - When using this option, there is no guarantee that the result is correct. This is because it's impossible to define what constitutes a "correct result" when parsing a non-standard CSV file.
> - DuckDB runs a best-effort parser, as it will try only fail if it encounters an error to which it can't find any reasonable guess of what the correct data should look like (e.g., imagine a column of integers, where some integers are represented as a spelled-out string, this will cause a casting error). While it is possible that, with a given combination of errors, our parser may still fail, we are striving to make it even more robust as we face the abyss.

Ignoring Errors

The other option suggested by the error message was to set `ignore_errors = true`. This option simply means that any rows which do not fit the chosen dialect or schema will be skipped from the result. In practice, if we add it to our query:

```
FROM read_csv('cafes.csv',
    auto_detect = false,
    header = true,
    strict_mode = false,
    delim = ',',
    quote = '"',
    escape = '"',
    columns = {'ZIP': 'INT16', 'Name': 'VARCHAR', 'Country': 'VARCHAR'},
    ignore_errors = true
);
```

We get an almost complete result! Only missing line number 4 `1095,Joost` since it is missing a column.

ZIP int16	Name varchar	Country varchar
1014	Cafe "Gedoogt"	Netherlands
1015	Cafe De Tuin Amsterdam	Netherlands
1872	Cafe Gezellig	Netherlands

> The `ignore_errors` option also affects the sniffer's behavior, as it will ignore lines that produce errors during sniffing. In general, the configuration that results in the fewest errors will be preferred.

Null Padding for Missing Values

The last option that can help when reading non-standard CSV files is the `null_padding` option. This option is useful when lines in the CSV file have an inconsistent number of values, with some missing values for given columns. Let's read the file with the `null_padding = true` option.

```
FROM read_csv('cafes.csv',
    auto_detect = false,
    header = true,
    strict_mode = false,
    delim = ',',
    quote = '"',
    escape = '"',
    columns = {'ZIP': 'INT16', 'Name': 'VARCHAR', 'Country': 'VARCHAR'},
    null_padding = true
);
```

This will produce the following result:

ZIP int16	Name varchar	Country varchar
1014	Cafe "Gedoogt"	Netherlands
1015	Cafe De Tuin Amsterdam	Netherlands
1095	Joost	NULL
1872	Cafe Gezellig	Netherlands

Minimal Configuration to Read the Example File

Whew. We loaded the file but the `read_csv` call is about 200 characters long! Luckily, we can do better than that: if we set both `strict_mode = false` and `null_padding = true`, we only need to specify the `quote` and `escape` values. This call only requires about 80 characters and gets a valid result from this CSV file:

```
FROM read_csv('cafes.csv',
    strict_mode = false,
    null_padding = true,
    quote = '"',
    escape = '"'
);
```

ZIP int64	Name varchar	Country varchar	column3 int64
1014	Cafe "Gedoogt"	Netherlands	NULL
1015	Cafe De Tuin Amsterdam	Netherlands	NULL
1095	Joost	NULL	NULL
1872	Cafe Gezellig	Netherlands	5

With this configuration, the sniffer actually creates a new column (`column3`) to accommodate the extra value in the last line. This example shows that even on a CSV file exhibiting several issues, DuckDB's CSV sniffer can still make reasonably good decisions on the file's dialect and schema.

> **Tip.** You can drop the last column by e.g. using `SELECT #1, #2, #3 FROM ...` in the query.

The Pollock Benchmark

The Pollock Benchmark is a CSV data loading benchmark designed to evaluate how robust a CSV system is when reading non-standard CSV files. The benchmark was published at VLDB 2023 and is completely open-source, with a DuckDB entry recently added to its repository.

The authors of the Pollock benchmark analyzed over 245,000 public CSV datasets to understand the most common ways the RFC-4180 standard is violated in CSV files in the wild. After identifying the most common error types, they created a CSV file generator with a pollution mechanism, which inserts these errors. In total, the benchmark tool generates over 2,200 polluted files. It also generates the correct dialect configuration that should be used to read these files, along with a clean version of the file.

To evaluate the robustness of systems, the benchmark tool reads the polluted file with the configuration set for a given system-under-test. It then uses the system to write a new file with the answer and compares it with the clean version. This comparison yields a score that reflects how accurately the system has read the file. There are mainly two scores that the benchmark produces: the simple score and the weighted score, with the weighted score accounting for how commonly that type of error appears in real life.

In this section, we will describe the most common errors depicted in the paper and share the results of the benchmark after adding DuckDB.

Common Errors

The paper depicts many of the common errors that appear in CSV files. In this section, we will briefly discuss some of the most common errors, but refer to the paper for a complete overview:

1. **Inconsistent Number of Cells in Rows:** RFC-4180 requires that all rows, including the header, have a consistent number of columns, but many files have too many or too few delimiters per row.
2. **Non-Standard Newline Sequences:** RFC-4180 requires that files use a carriage return followed by a line feed (`\r\n`), but many files use only line feeds, carriage returns, or a mix of them.
3. **Multiple Header Lines:** The standard allows for an optional single header line but many files have multiple headers.
4. **Incorrectly Quoted or Unescaped Cells:** This error occurs when quoted values have unescaped quotes in them. See the `file_escape_char_0x00.csv` for an example. Note how GitHub's table renderer, which can read the clean version of this file fails to parse it.
5. **Files with Multibyte Delimiters:** The standard requires files to be comma-separated. Although single-byte delimiters are widely accepted, multibyte delimiters are common.
6. **Files with Multiple Tables:** When a CSV file stores multiple tables with different schemas in the same file.

Methodology

The benchmark includes a wide variety of systems being tested, such as CSV parsing frameworks (e.g., CleverCSV), relational database systems (e.g., PostgreSQL), spreadsheet software (e.g., LibreOffice Calc), data visualization tools (e.g., Dataviz), and dataframe libraries (e.g., Pandas).

An important aspect of the benchmark is that it provides the dialect and schema for each file. Without this, systems that do not have a sniffer (e.g., PostgreSQL) would not be able to read the file. There is no specific rule or differentiation in how much each system utilizes that information. For example, Pandas only takes partial advantage of these settings. To incorporate these differences in our evaluation of DuckDB, we decided to add two different configurations:

1. **DuckDB (benchmark config).** Under this configuration, all the options in the configuration file that are relevant to DuckDB – such as CSV dialect and schema – are passed to the reader. In addition, we also set all the options described in the previous section (i.e., `null_padding = true, strict_mode = false, ignore_errors = true`). This essentially tells us how much we can read from these files if the users manually set the necessary options.
2. **DuckDB (auto-detect only).** Under this configuration, we do not take advantage of providing custom configuration files. The only options set are those that allow for non-standard file reading (i.e., `null_padding = true, strict_mode = false, ignore_errors = true`). Hence, this option also evaluates the full power of our sniffer in scenarios of uncertainty.

Pollock Scores

The following table presents the results with DuckDB included. We also limited the original table to show only the systems with the best scores in each category (e.g., parsing frameworks, relational systems, and so on). For both the simple and weighted scores, DuckDB with all options set (the default configuration for the Pollock benchmark) is the clear winner. In terms of the simple score, this means that DuckDB correctly read `99.61%` of the data from all the files. It also correctly handles the most common errors, as reflected in the weighted score.

Pollock scores sorted by weighted score (out of 10):

System under test	Pollock score (weighted)	Pollock score (simple)
DuckDB 1.2 (benchmark config)	9.599	9.961
“SpreadDesktop”	9.597	9.929
Pandas 1.4.3	9.431	9.895

System under test	Pollock score (weighted)	Pollock score (simple)
“SpreadWeb”	9.431	9.721
SQLite 3.39.0	9.375	9.955
DuckDB 1.2 (auto-detect only)	8.439	9.075
UniVocity 2.9.1	7.936	9.939
LibreOffice Calc 7.3.6	7.833	9.925
Dataviz	5.152	5.003

> According to the benchmark's website, “SpreadDesktop” is a desktop-based commercial spreadsheet software and “SpreadWeb” is a web-based spreadsheet software. Their real names were omitted due to licensing.

As expected, running DuckDB in full auto mode scores lower, since the sniffer must detect dialects and schemas by itself. Some files even have a multibyte delimiter, which is supported by DuckDB, but is not part of the sniffer's search space, yielding a lower score. In these cases, DuckDB still managed to read about `90.75%` of the data correctly, reaching in 9.075 as the total score and 8.439 as the total weighted score. Once again, this result comes from simply calling `read_csv('file_path', null_padding = true, strict_mode = false, ignore_errors = true)` with zero user input regarding the actual data configuration, demonstrating that DuckDB's CSV reader can indeed read most non-standard CSV files, even with minimal configuration!

Contributing to the Benchmark

It's entirely possible that your favorite CSV reader system is not yet included in the benchmark results. DuckDB, for example, was initially not included, but we found that adding it was quite easy! We hope that the DuckDB pull request serves as inspiration for those who wish to add their favorite systems to it. The reproduction of the results is quite simple too. Unlike most benchmarks that focus on performance and correctness, the Pollock Benchmark measures *accuracy,* making it easy to reproduce independently of the machine used.

Happy hacking!

Temporal Analysis with Stream Windowing Functions in DuckDB

Publication date: 2025-05-02

Authors: Petrica Leuca, Gábor Szárnyas

TL;DR: DuckDB can perform time-based analytics using windows with different semantics (e.g., tumbling, hopping and sliding windows). In this post, we demonstrate these by detecting trends and anomalies in the railway service at Amsterdam Centraal Station.

Introduction

In data platforms, we usually categorize the data into dimension and fact data. While dimensions contain information about entities (name, address, serial number, etc.), facts contain events related to such entities (clicks, sales, bank transactions, readings from IoT devices, etc.). In general, fact data includes a timestamp attribute, denoting the moment when the event happened (or was observed).

When the timestamped data is processed on a streaming platform, it is often processed with *stream windowing functions* in order to organize the data into time windows. In this post, we will show how to apply stream windows on static timestamped fact data in DuckDB, as part of a data analysis task to compute train service summaries, trends and interruptions at Amsterdam Centraal Station.

> In a future post, we'll cover streaming design patterns with DuckDB.

For the current implementation, we will use the DuckDB database created in the dbt project detailed in the article "Fully Local Data Transformation with dbt and DuckDB", based on the open data from the Rijden de Treinen *(Are the trains running?)* application. We start by attaching (in any DuckDB session) the database from our storage location.

```
ATTACH 'https://blobs.duckdb.org/data/dutch_railway_network.duckdb';
USE dutch_railway_network.main_main;
```

> **Warning.** The database is rather big (approx. 1.2 GB), therefore make sure to have a stable internet connection. Instead of attaching the database, you can also download the database file and connect to it from the command line:
>
> ```
> duckdb dutch_railway_network.duckdb -cmd 'USE main_main'
> ```

Tumbling Windows

Tumbling windows are fixed-size **[left-closed, right-open)** time intervals, used to calculate summaries at a certain time unit level (year, day, hour, etc.). Tumbling windows are also used to transform (irregular) fact data into time series data, by aggregating it at a regular time interval.

One way of implementing tumbling windows is to use the `date_trunc` function, which will truncate the timestamp to the specified precision. For example, in the following, we retrieve the number of services for each hour and each day in 2024:

```
SELECT
    date_trunc('hour', station_service_time) AS window_start,
    window_start + INTERVAL 1 HOUR AS window_end,
    count(*) AS number_of_services
FROM ams_traffic_v
WHERE year(station_service_time) = 2024
GROUP BY ALL
ORDER BY 1;
```

window_start timestamp	window_end timestamp	number_of_services int64
2024-01-01 01:00:00	2024-01-01 02:00:00	2
2024-01-01 02:00:00	2024-01-01 03:00:00	3
2024-01-01 03:00:00	2024-01-01 04:00:00	4
.	.	.
.	.	.
.	.	.
2024-12-31 20:00:00	2024-12-31 21:00:00	9
2024-12-31 21:00:00	2024-12-31 22:00:00	1
2024-12-31 23:00:00	2025-01-01 00:00:00	2
8781 rows (6 shown)		3 columns

Another approach is to use the `time_bucket` function, which will truncate the timestamp to the bucket width provided, starting from the specified offset. For example, we calculate the number of services each quarter of an hour, starting with `00`:

```
SELECT
    time_bucket(
        INTERVAL 15 MINUTE, -- bucket width
        station_service_time,
        INTERVAL 0 MINUTE -- offset
    ) AS window_start,
    window_start + INTERVAL 15 MINUTE as window_end,
    count(*) AS number_of_services
FROM ams_traffic_v
WHERE year(station_service_time) = 2024
GROUP BY ALL
ORDER BY 1;
```

window_start timestamp	window_end timestamp	number_of_services int64
2024-01-01 01:30:00	2024-01-01 01:45:00	1
2024-01-01 01:45:00	2024-01-01 02:00:00	1
2024-01-01 02:15:00	2024-01-01 02:30:00	2
.	.	.
.	.	.
.	.	.
2024-12-31 20:45:00	2024-12-31 21:00:00	2
2024-12-31 21:00:00	2024-12-31 21:15:00	1
2024-12-31 23:45:00	2025-01-01 00:00:00	2
32932 rows (6 shown)		3 columns

> The time bucket function is generating the buckets from the timestamp column itself, there could be gaps in the time series data. As seen in the above result, the first record is `2024-01-01 01:30:00`, because there are no records before that timestamp.

Given that tumbling windows are non-overlapping intervals, we can calculate summaries, such as the average number of services during a 15 minute interval. It is interesting to observe that the number of train services is quite stable during the day but it's much lower during the night – even in Amsterdam.

Hopping Windows

Hopping windows are fixed-size time intervals, but, contrary to tumbling windows, are overlapping. A hopping window is defined by:

- how much time should elapse between the window start time, called **hopping size**;

- how much time should a window contain, called **window size**.

One use case for hopping windows is to identify the five busiest 15-minute periods (window size) during 2024, starting every 5 minutes (hopping size). We start by generating artificial hopping windows for all the dates we are interested in:

```
WITH time_range AS (
    SELECT
        range AS window_start,
        window_start + INTERVAL 15 MINUTE AS window_end
    FROM range(
        '2024-01-01 00:00:00'::TIMESTAMP,
        '2025-01-01 00:00:00'::TIMESTAMP,
        INTERVAL 5 MINUTE -- hopping size
    )
)
```

window_start timestamp	window_end timestamp
2024-01-01 00:00:00	2024-01-01 00:15:00
2024-01-01 00:05:00	2024-01-01 00:20:00
2024-01-01 00:10:00	2024-01-01 00:25:00
.	.
.	.
.	.
2024-12-31 23:45:00	2025-01-01 00:00:00
2024-12-31 23:50:00	2025-01-01 00:05:00
2024-12-31 23:55:00	2025-01-01 00:10:00
105408 rows (6 shown)	2 columns

We then join the above intervals with the train service data in order to calculate the number of services for each **[left-closed, right-open)** interval:

```
SELECT
    window_start,
    window_end,
    count(service_sk) AS number_of_services
FROM ams_traffic_v
INNER JOIN time_range AS ts
    ON station_service_time >= ts.window_start
        AND station_service_time < ts.window_end
GROUP BY ALL
ORDER BY 3 DESC, 1 ASC
LIMIT 5;
```

resulting in:

window_start timestamp	window_end timestamp	number_of_services int64
2024-02-17 10:25:00	2024-02-17 10:40:00	28
2024-02-17 11:25:00	2024-02-17 11:40:00	28
2024-02-17 16:25:00	2024-02-17 16:40:00	28
2024-02-17 09:25:00	2024-02-17 09:40:00	27
2024-02-17 12:25:00	2024-02-17 12:40:00	27

Can you imagine how it must have been like in the control room when within 15 minutes, 28 trains were arriving or departing in a station with 15 tracks?

> By applying a `RIGHT OUTER JOIN` in the above query, gaps are filled with 0 number of services.

Sliding Windows

Sliding windows are overlapping intervals, but, compared to hopping windows, they are dynamically generated from the time column analyzed, therefore changing when new records are inserted. Sliding windows can be implemented by using the RANGE window framing:

```
SELECT
    station_service_time - INTERVAL 15 MINUTE AS window_start, -- window size
    station_service_time                      AS window_end,
    count(service_sk) OVER (
        ORDER BY station_service_time
            RANGE
                BETWEEN INTERVAL 15 MINUTE PRECEDING -- window size
                AND CURRENT ROW
    ) AS number_of_services
FROM ams_traffic_v
ORDER BY 3 DESC, 1
LIMIT 5;
```

window_start timestamp	window_end timestamp	number_of_services int64
2024-02-17 11:25:00	2024-02-17 11:40:00	29
2024-02-17 10:24:00	2024-02-17 10:39:00	28
2024-02-17 11:18:00	2024-02-17 11:33:00	28
2024-02-17 11:18:00	2024-02-17 11:33:00	28
2024-02-17 11:23:00	2024-02-17 11:38:00	28

> Because the current row is included in the calculation, the sliding windows are **[left-closed, right-closed]**.

Session Windows

A *session window* groups events that happen close together in time, separated by inactivity gaps. A new session starts when the time between two events exceeds a defined timeout. The most common use case of session windows is to detect gaps in the timestamped data.

We continue the data analysis by identifying the days in which there were periods of time larger than 10 minutes in which no train was arriving/departing in/from the Amsterdam Centraal Station. In this context, a session window is the period of time in which train services run without a service inactivity gap longer than 10 minutes.

We start by calculating, for each record, the previous service time, by using the `lag` window function. We observed above that there is almost no traffic during the night, therefore we include only services between 6 AM and 11 PM:

```
SELECT
    service_sk,
    station_service_time,
    lag(station_service_time) OVER (
        PARTITION BY station_service_time::DATE
        ORDER BY station_service_time
    ) AS previous_service_time,
    date_diff('minute', previous_service_time, station_service_time) AS gap_minutes
FROM ams_traffic_v
WHERE hour(station_service_time) BETWEEN 6 AND 23
```

In the above query we also calculate the gap, in minutes, between the current service and the previous service, with `date_diff`. If there is no previous service, the column will be `NULL`, depicting the first service session in the day:

station_service_time timestamp	previous_service_time timestamp	gap_minutes int64
2024-01-09 06:00:00	NULL	NULL
2024-01-16 06:00:00	NULL	NULL
2024-01-22 06:00:00	NULL	NULL
.	.	.
.	.	.
.	.	.
2024-11-28 06:01:00	NULL	NULL
2024-12-05 06:01:00	NULL	NULL
2024-12-23 06:00:00	NULL	NULL
366 rows (6 shown)		3 columns

> **Tip.** Because `gap_minutes` is computed based on a window function, we can filter on it with `QUALIFY`, e.g.: `QUALIFY gap_minutes IS NULL`

We then mark if the current record is in the same session as the previous one, by comparing the minutes elapsed to a timeout, in our case 10 minutes:

```
IF(gap_minutes >= 10 OR gap_minutes IS NULL, 1, 0) AS new_session
```

By applying a moving sum, at day level, over the `new_session` attribute, we assign an identifier to the session:

```
sum(new_session) OVER (
    PARTITION BY station_service_date
    ORDER BY station_service_time ROWS UNBOUNDED PRECEDING
) AS session_id_in_day
```

Bringing it all together, we can now retrieve the dates which had at least one inactivity gap of 10 minutes during the 18 hours day service time (the number of hours between 6 AM and 11 PM):

```
WITH ams_daily_traffic AS (
    SELECT
        service_sk,
        station_service_time,
        lag(station_service_time) OVER (
            PARTITION BY station_service_time::DATE
            ORDER BY station_service_time
        ) AS previous_service_time,
        date_diff('minute', previous_service_time, station_service_time) AS gap_minutes
    FROM ams_traffic_v
    WHERE hour(station_service_time) BETWEEN 6 AND 23
), window_calculation AS (
        SELECT
            service_sk,
            station_service_time,
            station_service_time::DATE AS station_service_date,
            gap_minutes,
            IF(gap_minutes >= 10 OR gap_minutes IS NULL, 1, 0) new_session,
            sum(new_session) OVER (
                PARTITION BY station_service_date
                ORDER BY station_service_time ROWS UNBOUNDED PRECEDING
            ) AS session_id_in_day
       FROM ams_daily_traffic
), session_window AS (
    SELECT
        station_service_date,
        session_id_in_day,
        max(gap_minutes)          AS gap_minutes,
        min(station_service_time) AS window_start,
```

```
        max(station_service_time) AS window_end,
        count(service_sk)         AS number_of_services
    FROM window_calculation
    GROUP BY ALL
)
SELECT
    station_service_date,
    max(ceil(date_diff('minute', window_start, window_end) / 60)) AS number_of_hours_without_gap,
    count(*) AS number_of_sessions,
    sum(number_of_services) as number_of_services,
FROM session_window
GROUP BY ALL
HAVING number_of_hours_without_gap < 18
ORDER BY 2, 1;
```

station_service_date date	number_of_hours_without_gap double	number_of_sessions int64	number_of_services int128
2024-04-29	7.0	12	521
2024-12-31	14.0	6	946
2024-01-01	16.0	6	847
2024-04-30	16.0	7	645
2024-04-14	17.0	3	1289
2024-05-01	17.0	5	788
2024-05-02	17.0	3	729
2024-05-03	17.0	5	699
2024-05-04	17.0	3	907
2024-05-19	17.0	3	837
2024-10-28	17.0	2	748
2024-10-29	17.0	2	785
2024-10-30	17.0	2	783
2024-11-02	17.0	2	654
14 rows			4 columns

Something must have happened on 29 April 2024! We observe that, during the 18 hours of service, there were 12 session windows, which means that, for at least 10 times, no train arrived or departed during a period of 10 minutes. A reason for this could be that a regular train service was not running on that day. And indeed maintenance work started between Amsterdam and Utrecht.

> **Tip.** Time windows are visualized with Plotly timeline charts, a type of Gantt charts.

Conclusion

In this post we have demonstrated how stream windowing functions can be implemented on historical timestamped data in DuckDB, offering a starting point in time (series) data analysis. We also recommend "Catching up with Windowing", a post about DuckDB's windowing features, which can be adopted in the functions presented herein.

Sorting on Insert for Fast Selective Queries

Publication date: 2025-05-14

Author: Alex Monahan

TL;DR: Sorting data when loading can speed up selective read queries by an order of magnitude, thanks to DuckDB's automatic min-max indexes (also known as zone maps). This approach applies to most columnar file formats and databases as well. This post unpacks the DuckDB file structure as an example of a columnar data format and gives practical advice for using sorting to improve the speed of queries.

The fastest way to read data is to not read data. It's as simple as that! This post is all about how to read as little data as possible to answer selective read queries.

> Feel free to skip straight to the best practices!

Use Cases

The cases where these techniques are most useful are when read performance is more critical than write performance. In these cases, we can deliberately add ordering steps that will slow down our writes, but will greatly speed up our reads. This is often the case when data is pre-processed in the background, but read queries are customer facing in a dashboard or app.

Using sorting will be helpful in any of these situations:

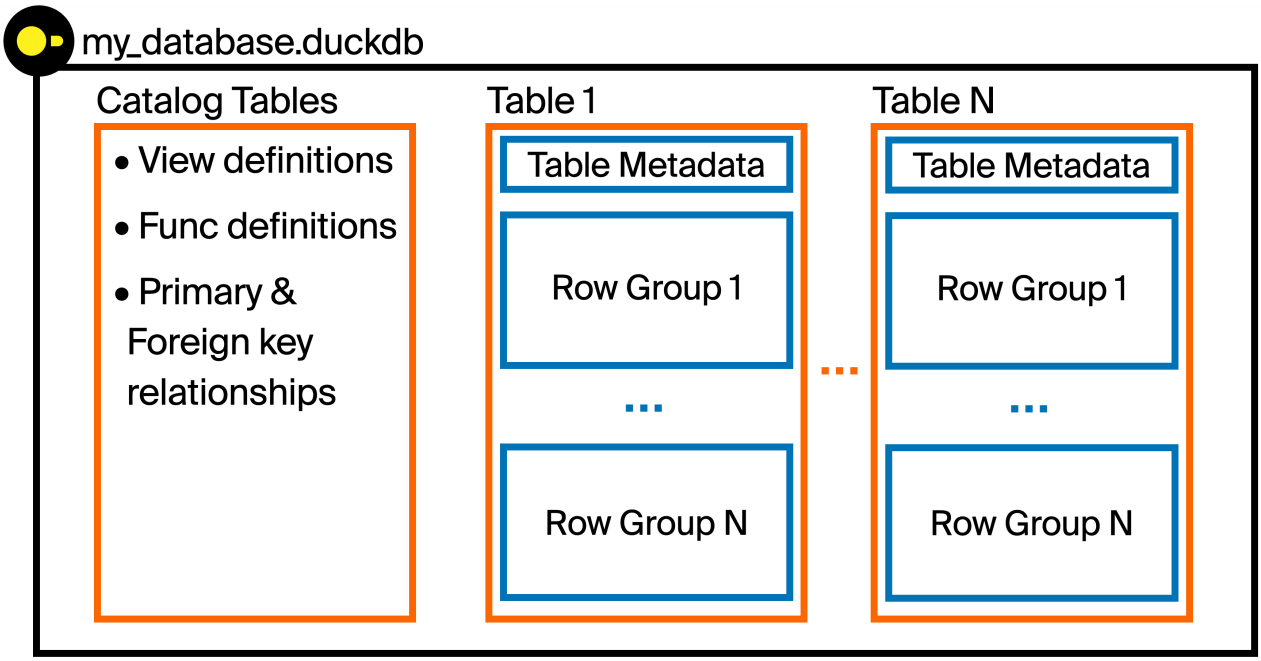
- Your dataset is large and doesn't fit entirely in memory
- You only want to read a portion of your dataset for each query
- You access your data via HTTP(S)
- Your data lives in the cloud on object storage such as AWS S3

This overview describes the DuckDB file format, but thanks to DuckDB's partial reading support, these techniques can be generally applied to nearly any columnar file format or database. This is a great way to speed up querying Apache Parquet files on remote endpoints – including data lakes!

> Stay tuned for a subsequent post covering advanced multi-column sorting!

How Does the DuckDB Format Help?

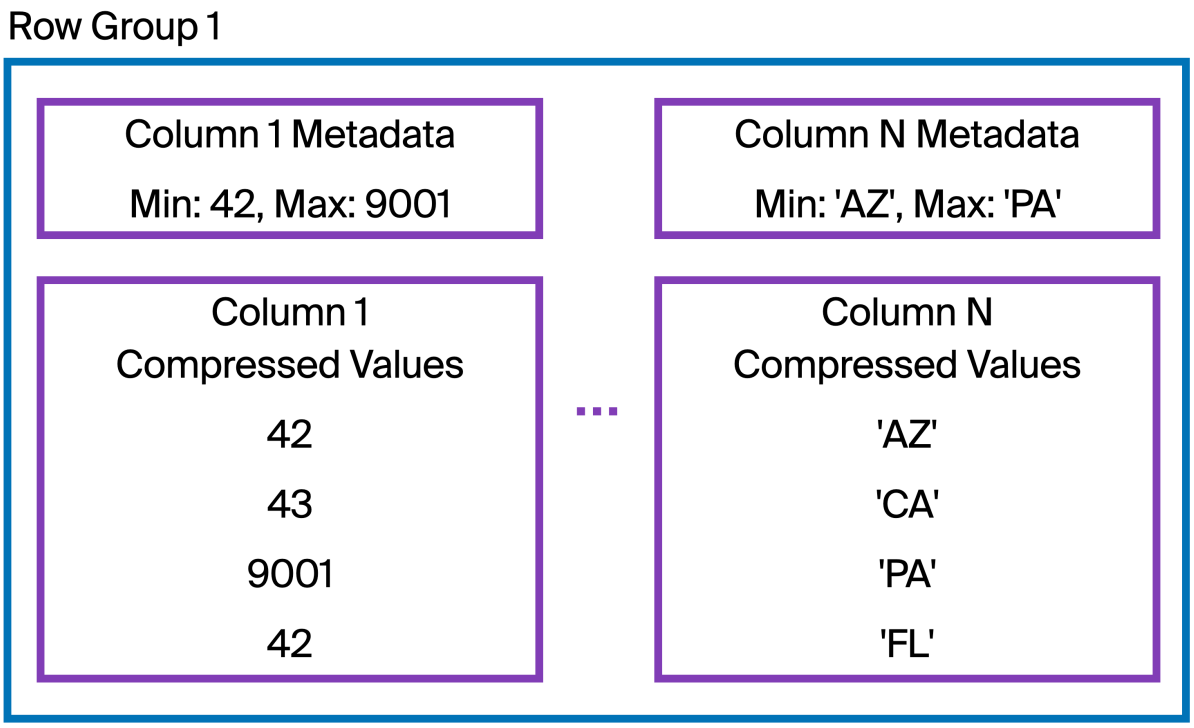
Let's build up some intuition around the structure of the DuckDB file format! DuckDB stores data in a single file. Each file is called a *database* (the DuckDB library is the *database engine*).



Each database file can store multiple tables, views, functions, indexes, and primary/foreign key relationships all in the same file. This has some key advantages for portability, but it also allows for the storage of more than just data – metadata can be stored as well. This metadata allows for the DuckDB engine to selectively read just portions of the DuckDB file as they are requested, which is key for handling larger than memory datasets and for the performance optimizations in this post!

DuckDB stores data in a columnar fashion (meaning that values within a column are stored together in the same set of blocks). However, columnar does not mean storing the entire column contiguously! Before storing data, DuckDB breaks tables up into chunks of rows called row groups. Each row group is 122 880 rows by default.

Next, we zoom in on `Row Group 1` from `Table 1` in the first diagram:



Within each row group, the data related to a single column is stored contiguously together on disk in one or more blocks. DuckDB com-

presses this data to reduce file size.

Using Zone Maps to Skip Reading Data

So, how does storing data in columns help with selective read queries? By itself, it does not! However, at the start of the column data stored within each row group, DuckDB also stores metadata about the column data being stored. This includes the minimum and maximum values of that column within that row group. We call these *zone maps* or *min-max indexes.*

When DuckDB receives a SQL query that contains a filter, before reading the column segments off of disk, it checks the metadata first. Could this filter value possibly fall within the minimum/maximum range of this column segment? If it is not possible, then DuckDB can skip reading the data in that entire row group.

For example, `Column N` in this example is storing the abbreviations for US states. If we are querying data for the state of VA (Virginia), could this row group contain any data for VA? Our query could be:

```
FROM "Table 1"
WHERE "Column N" = 'VA';
```

> DuckDB's friendly SQL makes the traditional SQL `SELECT *` optional!

DuckDB first checks the zone map (labeled `Column N Metadata` in the diagram). Does VA fall within the range (alphabetically) of AZ to PA? No it does not! We can skip this entire row group.

If instead we were filtering for the state of NM (New Mexico), the zone map indicates that it is possible that NM data exists within this row group. As a result, DuckDB will retrieve the entire column of data (possibly from a remote location) and check each row to see if any NM data exists. According to the diagram, there is no NM data in this row group, so retrieving it and checking each row for a match would be unnecessary effort. All 122 880 rows would need to be checked! If only the zone map index did not cover such a large range of state abbreviations...

Strategically Skipping Data

Our goal is to sort our data so that the zone maps are as selective as possible for the columns we are interested in filtering. If the min-max index of this row group only extended from AZ to CA, then the data would be far more likely to be skipped. Said another way, each subset of the data we want to retrieve should only be stored in a few row groups. We don't want to pull every row group and check every row! Hopefully states that are early in the alphabet are all grouped together in the same row group, for example.

When executing selective queries, you could aim for pulling just a single row group! However, since DuckDB's multithreading model is based on row groups, you should still see high performance as long as the number of row groups is less than the number of threads (≈CPU cores) DuckDB is using.

Sorting Best Practices

There are many rules of thumb for how to best utilize min-max indexes, since their effectiveness is highly data and workload dependent. These are useful both in DuckDB as well as in other columnar formats like Apache Parquet or other columnar databases. Here are several approaches to consider!

When deciding which columns to sort, it is critical to examine the `WHERE` clauses of any read workloads. Often the most important factor is which columns are used to filter most often. A basic approach would be to sort by all columns that are used as filters, beginning with the columns used most often.

Another option if the workload involves filtering by multiple different columns is to consider sorting first by columns with the lowest cardinality (the fewest unique values). For example, it may be helpful to sort by a broad `customer_type` prior to a unique `customer_id`.

It is often useful to only read recent data, so ensuring recent data is not spread throughout the table can improve performance. However, when ordering by timestamps, be mindful that timestamps are frequently very high cardinality. As a result, sorting by lower cardinality

columns first may be helpful. You may be tempted to sort by timestamp first – **resist that temptation!** (or at least test out alternatives). It may be more beneficial to sort by a timestamp rounded to the nearest week, month, or year and then by other columns.

> To benefit from min-max indexes, a `WHERE` clause must filter directly on a specific column, not on a calculated expression. If an expression is used, it must be evaluated for each row, so no row groups can be skipped!

Additional Techniques

There are more ways to take full advantage of the min-max indexes in DuckDB!

Avoid Small Inserts

If a workload is inserting data in small batches or a single row at a time, there is not an opportunity to effectively sort the data when it is being inserted. Instead, the data will be sorted largely by insertion time, which will only provide effective pruning for time-based filters. If possible, bulk inserts or batching will allow the sorting to work more effectively for other columns. As an alternative, there can be a periodic re-sorting job, which is analogous to a re-indexing task in transactional systems.

Sort in Chunks

Sorting can be a computationally expensive operation for large tables. One way to reduce the amount of memory (or disk spill) required when sorting is to process the table in pieces by looping through multiple SQL statements, each filtered to a specific chunk. Since SQL does not have a looping construct, this would be handled by a host language (Python, Jinja templating, etc.). The pseudocode would be to:

```
CREATE OR REPLACE TABLE sorted_table AS
    FROM unsorted_table
    WITH NO DATA;

for chunk in chunks:
    INSERT INTO sorted_table
        FROM unsorted_table
        WHERE chunking_column = chunk
        ORDER BY other_columns...;
```

This will have the effect of sorting initially by the chunking column, and then by the `other_columns`. It may also take longer to run (since the data must be scanned once per chunk), but memory use is likely to be much lower.

Sort the First Few Characters of Strings

Approximate sorting works well for improving read performance. In the zone maps of VARCHAR columns, DuckDB stores just the first 8 bytes of the min and max string values. As a result, there is no need to sort more than the first 8 bytes (8 ASCII characters)!

This has an added benefit of faster sorting, as the runtime of DuckDB's radix sort algorithm is sensitive to the length of strings (by design!). The time complexity of the algorithm is $O(nk)$, where n is the number of rows, and k is the width of the sorting key. Sorting by just the first few characters of a VARCHAR can be quicker and less compute intensive while achieving similar read performance. DuckDB's VARCHAR data type also inlines strings when they are under 12 bytes, so working with short strings is faster for that reason as well. For example:

```
CREATE OR REPLACE TABLE sorted_table AS
    FROM unsorted_table
    ORDER BY varchar_column_to_sort[:8];
```

> DuckDB's friendly SQL allows bracket notation for string slicing!

Filter by More Columns

Adding filters to a `WHERE` clause can be helpful if those columns being filtered have any kind of approximate order. For example, instead of just filtering by `customer_id`, if the table is sorted by `customer_type`, include that in the query also. Often, if the `customer_id` is known at query time, it is possible to know other metadata as well.

Adjust the Row Group Size

One parameter that can be tuned for specific workloads is the number of rows within a row group (the `ROW_GROUP_SIZE`). If there are many unique values within a column being filtered on, then a smaller number of rows per row group could reduce the total number of rows that must be scanned. However, there is an overhead of checking metadata more often when row groups are smaller, so there is a tradeoff.

A larger row group size may actually be preferable if a table is particularly large and queries are very selective. For example, if querying a large fact table with years of history, but filtering to only the last week of data. Larger row group sizes reduce the number of metadata checks that are necessary to reach the recent data. However, each row group is larger, so there is a tradeoff there as well.

To adjust the row group size, pass in a parameter when attaching a database. Note that a row group size should be a power of 2. The minimum row group size is 2 048, the vector size of DuckDB.

```
ATTACH './smaller_row_groups.duckdb' (ROW_GROUP_SIZE 8_192);
```

Sneak Peak of the Next Post in the Series

A subsequent post in this series will cover advanced sorting techniques. These can boost performance if multiple reader workloads with different filters are querying the same dataset. For example, what if I need to filter by US state for some queries, but by customer last name for others? Additionally, sorting by rounded time buckets (day, month, or year) and then by other columns can be helpful when only recent data is needed.

Microbenchmarks will show the 10× performance benefits that were mentioned at the start of this post! The post will also provide ways to measure how effectively a table is sorted along various columns.

Conclusion

Ordering data upon insert can significantly speed up read queries that include filters. Once your dataset becomes large or you are storing it remotely, consider applying these techniques. Plus, these approaches can be used in nearly any columnar file format or database!

Happy analyzing!

Machine Learning Prototyping with DuckDB and scikit-learn

Publication date: 2025-05-16

Author: Petrica Leuca

TL;DR: In this post, we prototype a machine learning workflow using DuckDB for data handling and scikit-learn for modeling.

Introduction

Machine learning prototyping often involves juggling datasets, preprocessing steps, and performance constraints, which can make the process both complex and time-consuming. scikit-learn is one of Python's most popular and powerful libraries for machine learning, offering a large set of utilities for building and evaluating models. In this post, we will explore how DuckDB complements `scikit-learn` in the model development life cycle, by implementing a penguin species prediction model on Penguins observations in the Palmer Archipelago.

> The following implementation is executed in a marimo Python notebook, which is available on GitHub, in our examples repository.

Data Preparation

We start by loading the Palmer Penguins dataset and use DuckDB's `COLUMNS(*)` expression to filter out any record which contains `NA` or `NULL`:

```
duckdb_conn.read_csv(
    "https://blobs.duckdb.org/data/penguins.csv"
).filter(
    "columns(*)::text != 'NA'"
).filter(
    "columns(*) is not null"
).select(
    "*, row_number() over () as observation_id"
).to_table(
    "penguins_data"
)
```

Even if the `NA` values are removed from the dataset, DuckDB has inferred the column type as `VARCHAR` at the moment of `read_csv`. Therefore, we modify the column type to a numeric type, `DECIMAL(5, 2)`, using the `ALTER TABLE` statement:

```
duckdb_conn.sql(
    "alter table penguins_data alter bill_length_mm set data type decimal(5, 2)"
)
```

> **Tip.** One can also define the schema at import time.

We can now plot the data and check for species specific clusters. For example, using a scatter plot we identify clusters at the combination between bill depth and bill length:

We also observe that there are a few descriptive columns in the dataset, such as `species` and `island`. In a machine learning workflow, one common preprocessing step is to transform such values into numerical values, assigning them a unique identifier; a process called *label encoding*. While scikit-learn offers a LabelEncoder utility, this process is very similar to how we work with reference tables in a data warehouse. Therefore, we define a function which will create, for each column (feature), a reference table, leveraging the DuckDB Python relational API:

```
def process_reference_data(duckdb_conn):
    for feature in ["species", "island"]:
        duckdb_conn.sql(f"drop table if exists {feature}_ref")
        (
            duckdb_conn.table("penguins_data")
            .select(feature)
            .unique(feature)
            .row_number(
                window_spec=f"over (order by {feature})",
                projected_columns=feature
            )
            .select(f"{feature}, #2 - 1 as {feature}_id")
            .to_table(f"{feature}_ref")
        )
        duckdb_conn.table(f"{feature}_ref").show()
```

After the execution of the above function, two tables which will contain the distinct values of the category and a unique identifier are created, e.g., `species_ref`:

species varchar	species_id int64
Adelie	0
Chinstrap	1
Gentoo	2

The last step in our data preparation step is to define a selection query variable, which selects the data from the initial dataset and the reference tables:

```
selection_query = (
    conn.table("penguins_data")
    .join(conn.table("island_ref"), condition="island")
    .join(conn.table("species_ref"), condition="species")
)
```

> `selection_query` is lazy evaluated, therefore, no data is retrieved at the definition time.

Model Training

Our scope is to predict a penguin's specie based on its characteristics (features), such as bill length and depth, flipper length, body mass and island. Such a model is called a classification model because it predicts the category to which the input data belongs.`scikit-learn` offers multiple classification models. For our dataset, we have chosen to implement a random forest classifier, based on decision trees:

We start by splitting the data into train and test data, with the `train_test_split` utility from `scikit-learn`:

```
def train_split_data(selection_query):
    X_df = selection_query.select("""
        bill_length_mm,
        bill_depth_mm,
        flipper_length_mm,
        body_mass_g,
        island_id,
        observation_id,
        species_id
    """).order("observation_id").df()
    y_df = [
        x[0]
        for x in selection_query.order("observation_id").select("species_id").fetchall()
    ]

    num_test = 0.30
```

```
    return train_test_split(X_df, y_df, test_size=num_test)

X_train, X_test, y_train, y_test = train_split_data(selection_query)
```

Splitting the data is a common step in a machine learning workflow and it returns:

- `X_train`, containing the input data based on which we want to assign a species category;
- `y_train`, containing the species for each record in `X_train`;
- `X_test`, containing the input data based on which we will test the model;
- `y_test`, containing the species for each record in `X_test`.

We then define the RandomForestClassifier and fit the `X_train` and `y_train` data to it. We save the model in a pickle file, such that we can use the model without retraining:

```
model = RandomForestClassifier(n_estimators=1, max_depth=2, random_state=5)

model.fit(X_train.drop(["observation_id", "species_id"], axis=1).values, y_train)

pickle.dump(model, open("./model/penguin_model.sav", "wb"))
```

We can now check the model accuracy score:

```
model.score(
    X_test.drop(["observation_id", "species_id"], axis=1).values, y_test
)
```

```
0.98
```

> We are working with `pickle` due to the small size of our model. Other persistence methods are detailed on the `scikit-learn` documentation page.

Inference with DuckDB

Inference is the process of using the model to get predictions on (new) data:

```
model = pickle.load(open("./model/penguin_model.sav", "rb"))

model.predict(...)
```

When working with DuckDB there are three approaches to retrieve the predicted category from the data. We can use Pandas, or a DuckDB Python UDF with or without batching. In the following, we present these approaches.

With Pandas

The most common method of retrieving predictions is to load the predictions into a new column in a Pandas dataframe:

```
predicted_df = selection_query.select("""
    bill_length_mm,
    bill_depth_mm,
    flipper_length_mm,
    body_mass_g,
    island_id,
    observation_id,
    species_id
""").df()

predicted_df["predicted_species_id"] = model.predict(
    predicted_df.drop(["observation_id", "species_id"], axis=1).values
)
```

With DuckDB, the dataframe can be queried with SQL:

```
(
    duckdb_conn.table("predicted_df")
    .select("observation_id", "species_id", "predicted_species_id")
    .filter("species_id != predicted_species_id")
)
```

resulting in:

observation_id int64	species_id int64	predicted_species_id int64
13	1	2
15	1	2
39	1	2
44	1	2
68	1	2
70	1	2
76	1	2
90	1	2
94	1	2
104	1	2
106	1	2
110	1	2
124	1	2
126	1	2
243	3	2
296	2	1
16 rows		3 columns

> **Warning.** If there is a table which has the same name as the dataframe, then `register` can be used to give another table name to the dataframe, e.g., `duckdb_conn.register("table_name", predicted_df)`

DuckDB Python UDF, Row by Row

DuckDB offers the possibility to register User Defined Functions (UDFs) from Python functions. Because the UDF is executed row by row, our prediction function will return the predicted species id for each row:

```
def get_prediction_per_row(
    bill_length_mm: Decimal,
    bill_depth_mm: Decimal,
    flipper_length_mm: int,
    body_mass_g: int,
    island_id: int
) -> int:
    model = pickle.load(open("./model/penguin_model.sav", "rb"))
    return int(
        model.predict(
            [
                [
                    bill_length_mm,
                    bill_depth_mm,
                    flipper_length_mm,
                    body_mass_g,
                    island_id,
                ]
            ]
        )[0]
    )
```

In the above Python function we provide the required features as input for the model and return the predicted value (an `integer`). With this information, we create the DuckDB function:

```
duckdb_conn.create_function(
    "predict_species_per_row", get_prediction_per_row, return_type=int
)
```

We can now use the UDF in the SQL SELECT clause:

```
selection_query.select("""
    observation_id,
    species_id,
    predict_species_per_row(
        bill_length_mm,
        bill_depth_mm,
        flipper_length_mm,
        body_mass_g,
        island_id
    ) as predicted_species_id
""").filter("species_id != predicted_species_id")
```

This returns the same results as above.

DuckDB Python UDF, Batch Style

While the row-by-row prediction can be useful when the scope is to retrieve the prediction for a limited number of rows, it will be less performant when working with more data. Therefore, by aggregating the data into arrays (which preserve order), we can simulate a mass retrieval (or batch style) of the prediction.

We first create a Python function to get the predictions for a JSON object, which contains the columnar representation of the features needed as model input:

```
def get_prediction_per_batch(input_data: dict[str, list[Decimal | int ]]) -> np.ndarray:
    """
    input_data example:
        {
            "bill_length_mm": [40.5],
            "bill_depth_mm": [41.5],
            "flipper_length_mm: [250],
            "body_mass_g": [3000],
            "island_id": [1]
        }
    """
    model = pickle.load(open("./model/penguin_model.sav", "rb"))

    input_data_parsed = orjson.loads(input_data)

    input_data_converted_to_numpy = np.stack(tuple(input_data_parsed.values()), axis=1)

    return model.predict(input_data_converted_to_numpy)


duckdb_conn.create_function(
    "predict_species_per_batch",
    get_prediction_per_batch,
    return_type=duckdb.typing.DuckDBPyType(list[int]),
)
```

> While DuckDB has MAP and STRUCT data types, which convert automatically to a dict, they have a slower execution time compared to `json_object` (including the `orjson` deserialization time).

With `json_object` we extract the columnar representation of the features, in the format `'feature_name': array[feature]`:

```
json_object(
    'bill_length_mm', array_agg(bill_length_mm),
    'bill_depth_mm', array_agg(bill_depth_mm),
    'flipper_length_mm', array_agg(flipper_length_mm),
    'body_mass_g', array_agg(body_mass_g),
    'island_id', array_agg(island_id)
) as input_data,
```

We then pack in a STRUCT, the predictions together with other columns we are interested in:

```
struct_pack(
    observation_id := array_agg(observation_id),
    species_id := array_agg(species_id),
    predicted_species_id := predict_species_per_batch(input_data)
) as output_data
```

Last, we unnest the results, in order to flatten the lists into tables:

```
.select("""
    unnest(output_data.observation_id) as observation_id,
    unnest(output_data.species_id) as species_id,
    unnest(output_data.predicted_species_id) as predicted_species_id
""")
```

The above query is wrapped into a Python function, named `get_selection_query_for_batch`, with which we can chain queries, e.g., for mass retrieval of incorrect predictions:

```
get_selection_query_for_batch(selection_query).filter("species_id != predicted_species_id")
```

A batch approach can be implemented by using LIMIT and OFFSET to loop through the data:

```
for i in range(4):
    (
        get_selection_query_for_batch(
            selection_query
            .order("observation_id")
            .limit(100, offset=100*i)
            .select("*")
        )
        .filter("species_id != predicted_species_id")
    ).show()
```

> LIMIT and OFFSET are executed last, therefore, they should be applied before the prediction selection.

Performance Considerations

To obtain performance data on a larger scale dataset, we have generated a dummy dataset with approximately 59 million records. On a sample of 10%, on a 16 GB MacBook Pro, the batch processing ranges between 3 and 4 seconds, while the Pandas implementation is executed under 1 second. This is because the Python UDF includes multiple conversion steps which affect performance:

- parsing the input data as JSON;
- converting the JSON to numpy array;
- unnesting the array to rows.

We have still chosen to show-case the Python UDFs, because they are a powerful asset when working within a Python environment and the performance difference is negligible when we speak of small data.

Conclusion

In this post, we have demonstrated how DuckDB complements `scikit-learn` in the machine learning development life cycle, with a focus on the data preparation and inference phases. The inference results on the dummy data are poor; is this because of the model or the dummy data? We leave this question open to our readers, as a means to explore how DuckDB can be used in the model evaluation and model optimization phases.

The Lost Decade of Small Data?

Publication date: 2025-05-19

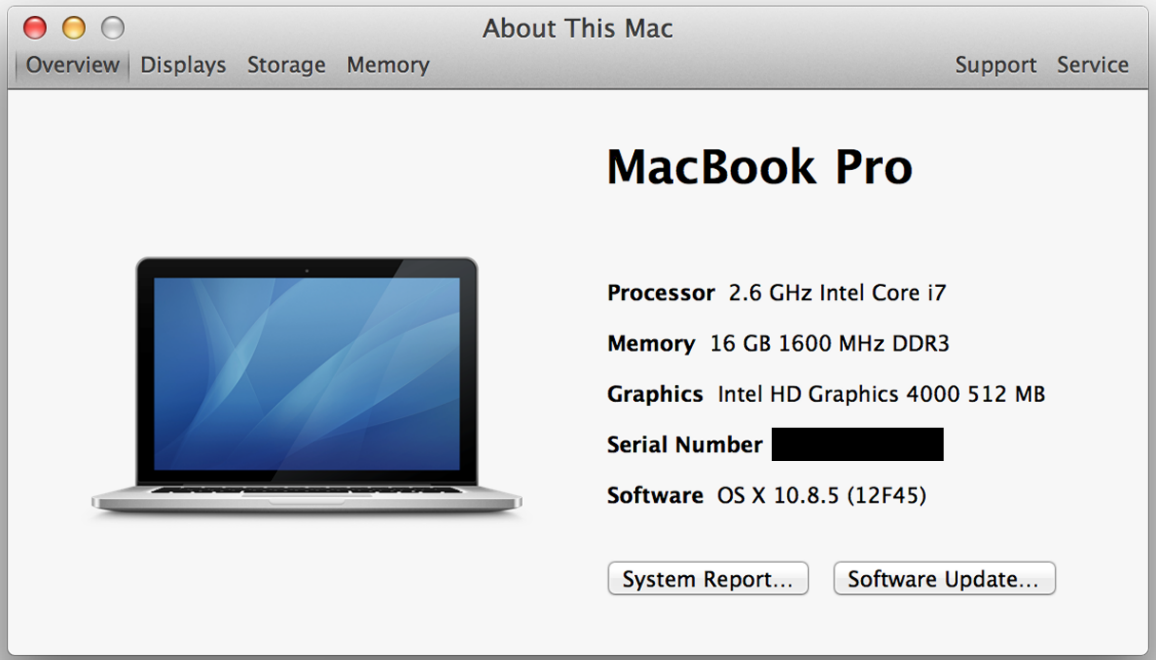
Author: Hannes Mühleisen

TL;DR: We benchmark DuckDB on a 2012 MacBook Pro to decide: did we lose a decade chasing distributed architectures for data analytics?

Much has been said, not in the very least by ourselves, about how data is actually not that "Big" and how the speed of hardware innovation is outpacing the growth of useful datasets. We may have gone so far to predict a data singularity in the near future, where 99% of useful datasets can be comfortably queried on a single node. As it was recently shown, the median scan in Amazon Redshift and Snowflake reads a doable 100 MB of data, and the 99.9-percentile reads less than 300 GB. So the singularity might be closer than we think.

But we started wondering, when did this development really start? When did personal computers like the ubiquitous MacBook Pro, usually condemned to running Chrome, become the data processing powerhouses that they really are today?

Let's turn our attention to the **2012** Retina MacBook Pro, a computer many people (myself included) bought at the time because of its gorgeous "Retina" display. Millions were sold. Despite being unemployed at the time, I had even splurged for the 16 GB RAM upgrade. But there was another often-forgotten revolutionary change in this machine: it was the first MacBook with a built-in Solid-State Disk (SSD) *and* a competitive 4-core 2.6 GHz "Core i7" CPU. It's funny to watch the announcement again, where they *do* stress the performance aspect of the "all-flash architecture" as well.

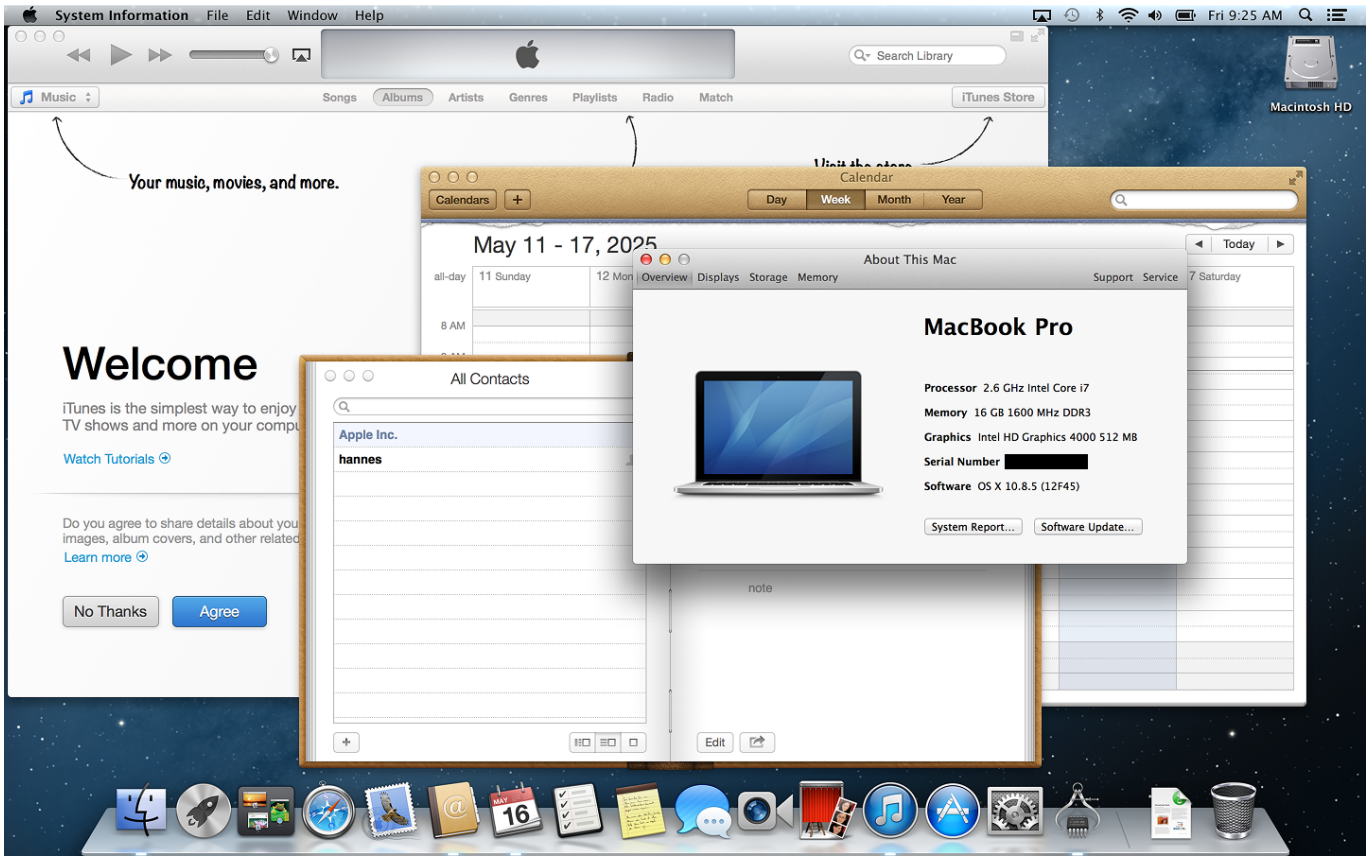


Side note: the MacBook Air was actually the first MacBook with an (optional) built-in SSD already back in 2008. But it did not have the CPU firepower of the Pro, sadly.

Coincidentally, I still have this laptop in the DuckLabs office, currently used by my kids to type their names in a massive font size or watch Bluey on YouTube when they're around. But can this relic still run modern-day DuckDB? How will its performance compare to modern MacBooks? And could we have had the data revolution that we are seeing now already back in 2012? Let's find out!

Software

First, what about the operating system? In order to make the comparison fair(er) to the decades, we actually *downgraded* the operating system on the Retina to OS X 10.8.5 "Mountain Lion", the operating system version that shipped just a few weeks after the laptop itself in July 2012. Even though the Retina can actually run 10.15 (Catalina), we felt a true 2012 comparison should also use an operating system from the era. Below is a screenshot of the user interface for those of us who sometimes feel a little old.



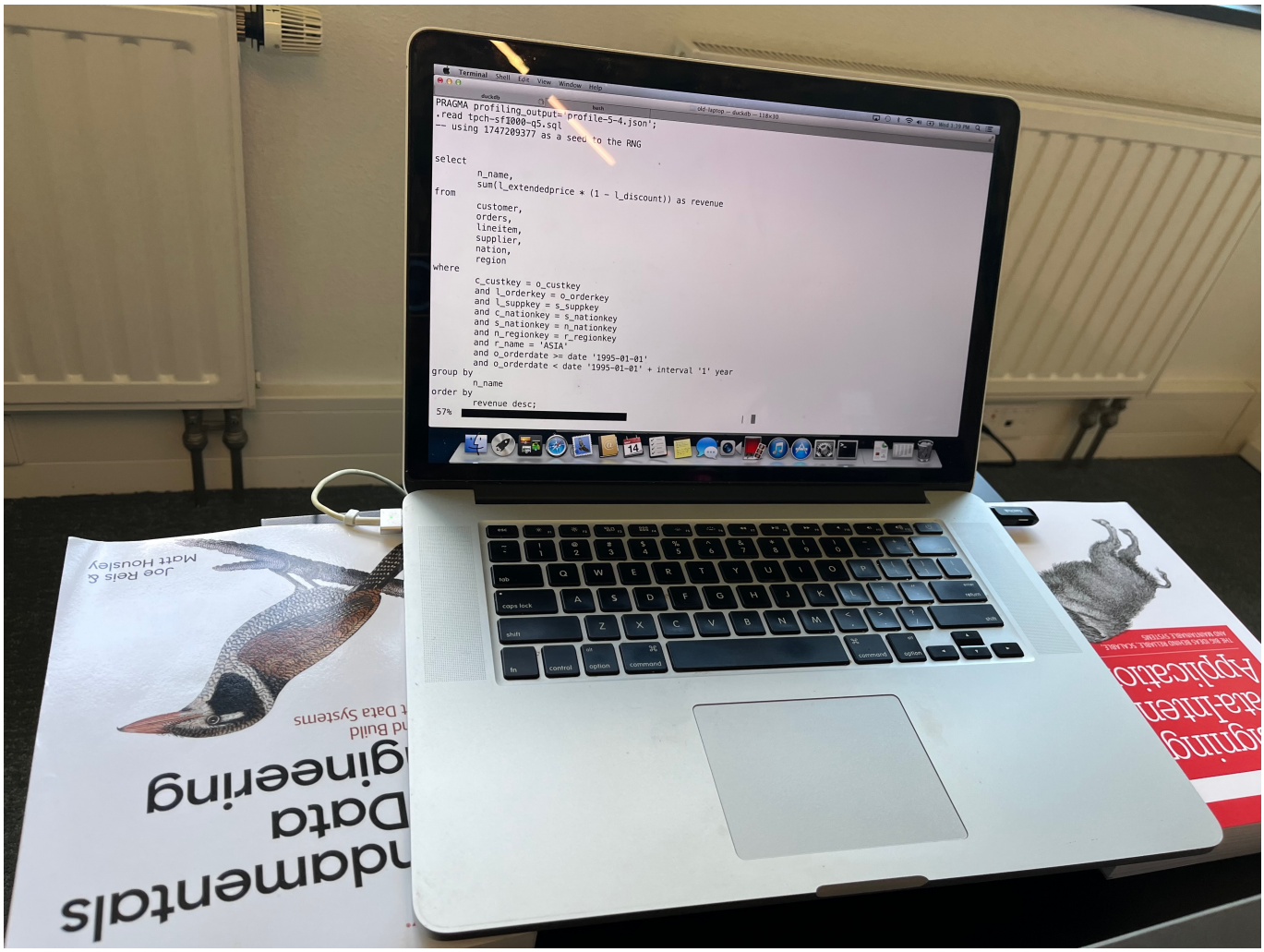
Moving on to DuckDB itself: here at DuckDB we are more than a little religious about portability and dependencies – or rather the lack thereof. This means that very little had to happen to make DuckDB run on the ancient Mountain Lion: the stock DuckDB binary is built with by default with backwards-compatibility to OS X 11.0 (Big Sur), but simply changing the flag and recompiling turned out to be enough to make DuckDB 1.2.2 run on Mountain Lion. We would have loved to also use a 2012 compiler to build DuckDB, but, alas, C++ 11 was unsurprisingly simply *too new* in 2012 to be fully supported by compilers. Either way, the binary runs fine and could have been also produced by working around the compiler bugs. Or we could have just hand-coded Assembly like others have done.

Benchmarks

But we're not interested in synthetic CPU scores, we're interested in synthetic SQL scores instead! To see how the old machine is holding up when performing *serious* data crunching, we used the *at this point rather tired but well-known TPC-H benchmark* at scale factor 1000. This means that the two main tables, `lineitem` and `orders` contain 6 and 1.5 billion rows, respectively. When stored as a DuckDB database, the database has a size of ca. 265 GB.

From the audited results on the TPC website, we can see that running the benchmark on this scale factor on a single node seems to require hardware costing hundreds of thousands of Dollars.

We ran each of the 22 benchmark queries five times, and took the median runtime to remove noise. However, because the amount of RAM (16 GB) is very much less than the database size (265 GB), no significant amount of the input data can be cached in the buffer manager, so those are not really what people sometimes call “hot” runs.



Below are the per-query results in seconds:

query	latency
1	142.2
2	23.2
3	262.7
4	167.5
5	185.9
6	127.7
7	278.3
8	248.4
9	675.0
10	1266.1
11	33.4
12	161.7

query	latency
13	384.7
14	215.9
15	197.6
16	100.7
17	243.7
18	2076.1
19	283.9
20	200.1
21	1011.9
22	57.7

But what do those cold numbers actually mean? The hidden sensation is that we actually *have* numbers, this old computer could actually complete all benchmark queries using DuckDB! If we look at the time a bit closer, we see the queries take anywhere between a minute and half an hour. Those are not unreasonable waiting times for analytical queries on that sort of data in any way. Heck, you would have been waiting way longer back in 2012 for Hadoop YARN to pick up your job in the first place only to spew stack traces at you at some point.

2023 Improvements

But how do those results stack up against a modern MacBook? As a comparison point, we used a modern ARM-based M3 Max MacBook Pro, which happened to be sitting on the same desk. But between them, the two MacBooks represent more than a decade of hardware development.

Looking at GeekBench 5 benchmark scores alone, we see a ca. 7× difference in raw CPU speed when using all cores, and ca. factor 3 difference in single-core speed. Of course there are also big differences in RAM and SSD speeds. Funnily, the display size and resolution are almost unchanged.

Here are the results side-by-side:

query	latency_old	latency_new	speedup
1	142.2	19.6	7.26
2	23.2	2.0	11.60
3	262.7	21.8	12.05
4	167.5	11.1	15.09
5	185.9	15.5	11.99
6	127.7	6.6	19.35
7	278.3	14.9	18.68
8	248.4	14.5	17.13
9	675.0	33.3	20.27
10	1266.1	23.6	53.65
11	33.4	2.2	15.18
12	161.7	10.1	16.01
13	384.7	24.4	15.77

query	latency_old	latency_new	speedup
14	215.9	9.2	23.47
15	197.6	8.2	24.10
16	100.7	4.1	24.56
17	243.7	15.3	15.93
18	2076.1	47.6	43.62
19	283.9	23.1	12.29
20	200.1	10.9	18.36
21	1011.9	47.8	21.17
22	57.7	4.3	13.42

We do see significant speedups, from 7 up to as much as 53. The geometric mean of the timings improved from 218 to 12, a ca. 20× improvement.

Reproducibility

The binary, scripts, queries, and results are available on GitHub for inspection. We also made the TPC-H SF1000 database file available for download so you don't have to generate it. But be warned, it's a large file.

Discussion

We have seen how the decade-old MacBook Pro Retina has been able to complete a complex analytical benchmark. A newer laptop was able to significantly improve on those times. But absolute speedup numbers are a bit pointless here. The difference is purely *quantitative*, not *qualitative*.

From a user perspective, it matters much more that those queries complete in somewhat reasonable time, not if it took 10 or 100 seconds to do so. We can tackle almost the same kind of data problems with both laptops, we just have to be willing to wait a little longer. This is especially true given DuckDB's out-of-core capability, which allows it to spill query intermediates to disks if required.

What is perhaps more interesting is that back in 2012, it would have been completely feasible to have a single-node SQL engine like DuckDB that could run complex analytical SQL queries against a database of 6 billion rows in manageable time – and we did not even have to immerse it in dry ice this time.

History is full of "what if"s, what if something like DuckDB had existed in 2012? The main ingredients were there, vectorized query processing had already been invented in 2005. Would the now somewhat-silly-looking move to distributed systems for data analysis have ever happened? The dataset size of our benchmark database was awfully close to that 99.9% percentile of input data volume for analytical queries in 2024. And while the retina MacBook Pro was a high-end machine in 2012, by 2014 many other vendors shifted to offering laptops with built-in SSD storage and larger amounts of memory became more widespread.

So, yes, we really did lose a full decade.

Announcing DuckDB 1.3.0

Publication date: 2025-05-21

Author: The DuckDB team

TL;DR: The DuckDB team is happy to announce that today we're releasing DuckDB version 1.3.0, codenamed “Ossivalis”.

> To install the new version, please visit the installation guide. Note that it can take a few hours to days to release some client libraries (e.g., Go, R, Java) and extensions (e.g., the UI) due to the extra changes and review rounds required.

We are proud to release DuckDB 1.3.0. This release of DuckDB is named “Ossivalis” after Bucephala Ossivalis, an ancestor of the Goldeneye duck that lived millions of years ago.

In this blog post, we cover the most important features of the new release. DuckDB is moving rather quickly, and we could cover only a small fraction of the changes in this release. For the complete release notes, see the release page on GitHub.

Breaking Changes and Deprecations

Old Linux glibc Deprecation

Now that all mainstream Linux distributions use glibc 2.28 or newer, DuckDB's official Linux binaries require at least glibc 2.28 or newer. The release is built using the `manylinux_2_28` image from Python, which combines an older glibc with a newer compiler. This change also implies that extensions are no longer distributed for the `linux_amd64_gcc4` platform.

We highly value portability, so it is of course still possible to build DuckDB from source for older versions of glibc.

Lambda Function Syntax

Previously, lambda functions in DuckDB could be specified using the single arrow syntax: `x -> x + 1`. The single arrow operator is also used by the JSON extension to express JSON extraction using the syntax `->'field'`. The two meanings of the single arrow operator are treated the same by the binder, thus they share the same (low) precedence, necessitating extra parentheses in JSON expression with equality checks:

```
SELECT (JSON '{"field": 42}')->'field' = 42;
-- throws a Binder Error:
-- No function matches the given name and argument types 'json_extract(JSON, BOOLEAN)

SELECT ((JSON '{"field": 42}')->'field') = 42;
-- return true
```

This often caused confusion among users, therefore, the new release deprecates the old arrow lambda syntax and replaces it with Python-style lambda syntax:

```
SELECT list_transform([1, 2, 3], lambda x: x + 1);
```

To make the transition smoother, the deprecation will happen in several steps over the next year. First, DuckDB 1.3.0 introduces a new setting to configure the lambda syntax:

```
SET lambda_syntax = 'default';
SET lambda_syntax = 'enable_single_arrow';
SET lambda_syntax = 'disable_single_arrow';
```

Currently, `default` enables both syntax styles, i.e., the old single arrow syntax and the Python-style syntax. We will gradually phase out the old syntax. DuckDB 1.5.0 will be the last release supporting the single arrow syntax without explicitly enabling it. Following versions will first disable the single arrow syntax by default, then ultimately remove the `lambda_syntax` flag and fully deprecate the single arrow syntax, so the old behavior will no longer be accessible.

Serializing Strings in List with Escapes

Starting with the new version, DuckDB escapes characters such as `'` in strings serialized in nested data structures to allow round-tripping between the serialized string and the nested representation. For example:

```
SELECT ['hello ''my'' world'] AS s;
```

DuckDB version 1.2.2 returns `[hello 'my' world]` while DuckDB 1.3.0 returns `['hello \'my\' world']`.

To serialize a list of strings with the old behavior, use the `array_to_string` function:

```
SELECT printf('[%s]', array_to_string(
        ['hello ''my'' world', 'hello ''cruel'' world'], ', '
    )) AS s;
```

```
┌──────────────────────────────────────────┐
│                    s                     │
│                varchar                   │
├──────────────────────────────────────────┤
│ [hello 'my' world, hello 'cruel' world]  │
└──────────────────────────────────────────┘
```

Minor SQL Parser Changes

- The term AT now needs quotes to be used as an identifier as it is used for time travel in Iceberg.
- `lambda` is now a reserved keyword due to the change in lambda syntax.
- GRANT is no longer a reserved keyword.

New Features

The new DuckDB release again contains a lot of exciting new features:

External File Cache

DuckDB is used a lot to read from remote files, e.g., Parquet files stored on HTTP servers or blob storage. Previous versions would always fully re-read file data. With this release, we added a cache for data from external files. This cache is subject to the overall DuckDB memory limit. If space is available, it will be used to dynamically cache data from external files. This should greatly improve re-running queries on remote data. For example:

```
.timer on
.mode trash -- do not show query result
FROM 'https://blobs.duckdb.org/data/shakespeare.parquet';
Run Time (s): real <1.456> user 0.037920 sys 0.028510
FROM 'https://blobs.duckdb.org/data/shakespeare.parquet';
Run Time (s): real <0.360> user 0.029188 sys 0.007620
```

We can see that the query is much faster the second time around due to the cache. In previous versions, the runtime would have been the same.

The cache contents can be queried using the `duckdb_external_file_cache()` table function like so:

```
.mode duckbox -- re-enable output
FROM duckdb_external_file_cache();
```

path varchar	nr_bytes int64	location int64	loaded boolean
https://blobs.duckdb.org/data/shakespeare.parquet	1697483	4	true
https://blobs.duckdb.org/data/shakespeare.parquet	16384	1681808	true

The cache is enabled by default but can be disabled with:

```
SET enable_external_file_cache = false;
```

Directly Query Data Files with the CLI

DuckDB's command line interface (CLI) gained the capability to directly query Parquet, CSV or JSON files. This works by just using e.g. a Parquet file instead of the database file. This will expose a view that can be queried. For example, say we have a Parquet file called `region.parquet`, this will work:

```
duckdb region.parquet -c 'FROM region;'
```

r_name varchar
AFRICA
AMERICA
ASIA
EUROPE
MIDDLE EAST

When using the CLI like this, what actually happens is that we launch a temporary in-memory DuckDB database, and create two views over the given file:

- `file` – this view is always named the same, regardless of the name of the file.
- `[base_file_name]` – this view depends on the name of the file, e.g., for `region.parquet` this is `region`.

Both views can be queried and will give the same result.

The main advantage of this feature is usability: we can use the regular shell to navigate to a file, and then use DuckDB to open that file without having to refer to the path of the file at the SQL level.

TRY Expression

DuckDB already supported TRY_CAST, which was trying to cast a value but did not fail the query if this was not possible. For example:

```
SELECT TRY_CAST('asdf' AS INTEGER);
```

returns NULL. This release generalizes this functionality beyond casting to arbitrary expressions that can error using TRY. For example, the logarithm of 0 is undefined, and `log(0)` will throw an exception and tell you that it "cannot take logarithm of zero". With the new TRY, this will return NULL instead, e.g.:

```
SELECT TRY(log(0));
```

```
NULL
```

Again, this will work for arbitrary expressions. We recommend to use TRY sparingly however if an error is expected often because there is going to be a performance impact. If any *batch* of rows causes an error, we switch to row-by-row execution of the expression to figure out exactly which row had an error and which did not. This is slower.

Updatings Structs

Starting with the new release, it's possible to update the sub-schema of structs using the `ALTER TABLE` clause. You can add, drop, and rename fields:

```
CREATE TABLE test (s STRUCT(i INTEGER, j INTEGER));
INSERT INTO test VALUES (ROW(1, 1)), (ROW(2, 2));
ALTER TABLE test DROP COLUMN s.i;
ALTER TABLE test ADD COLUMN s.k INTEGER;
ALTER TABLE test RENAME COLUMN s.j TO l;
```

```
┌──────────────────────────────────┐
│                s                 │
│ struct(l integer, k integer)     │
├──────────────────────────────────┤
│ {'l': 1, 'k': NULL}              │
│ {'l': 2, 'k': NULL}              │
└──────────────────────────────────┘
```

Altering structs is also supported inside `LIST` and `MAP` columns.

Swapping in New Databases

The `ATTACH OR REPLACE` clause allows you to replace a database, so you can swap a database on the fly. For example:

```
ATTACH 'taxi_v1.duckdb' AS taxi;
USE taxi;
ATTACH OR REPLACE 'taxi_v2.duckdb' AS taxi;
```

This feature was implemented by external contributor `xevix`.

UUID v7 Support

> **Warning. Update (2025-05-23).** The UUID v7 implementation in DuckDB v1.3.0 is not consistent with the UUID standard, causing the timestamp values to be off. This means that the timestamps in the generated UUID v7 values will only be correct if they are used exclusively within DuckDB. Importing or exporting UUID v7 values from/to systems will yield incorrect timestamps. We patched this bug and the fix will soon be available in the preview builds and the upcoming 1.3.1 patch release.

DuckDB now supports UUID v7, which is a newer version of UUIDs. `UUIDv7` combines a Unix timestamp in milliseconds and random bits, offering both uniqueness and sortability. This is useful to e.g. order UUIDs by age or to combine the ubiquitous `ID` and `TIMESTAMP` columns in many tables into a single `UUIDv7` column.

New UUIDs can be created using the `uuidv7()` scalar function. For example:

```
SELECT uuidv7();
```

```
┌──────────────────────────────────────┐
│               uuidv7()               │
│                 uuid                 │
├──────────────────────────────────────┤
│ 8196f1f6-e3cf-7a74-bc0e-c89ac1ea1e19 │
└──────────────────────────────────────┘
```

There are also additional functions to determine the UUID version (`uuid_extract_version()`) and to extract the internal timestamp (`uuid_extract_timestamp()`), for example:

```
SELECT uuid_extract_version(uuidv7());
```

```
┌──────────────────────────────────┐
│ uuid_extract_version(uuidv7())   │
│              uint32              │
├──────────────────────────────────┤
```

```
┌──────────────────────────────┐
│              7               │
└──────────────────────────────┘
```

```
SELECT uuid_extract_timestamp(uuidv7());
```

```
┌──────────────────────────────────────┐
│ uuid_extract_timestamp(uuidv7())     │
│    timestamp with time zone          │
├──────────────────────────────────────┤
│ 2025-05-21 08:32:14.61+00            │
└──────────────────────────────────────┘
```

This feature was implemented by external contributor dentiny.

Expression Support in CREATE SECRET

DuckDB has an internal "secret" management facility for things like S3 credentials. With this release, it is possible to use scalar expressions in the creation of the secret. This allows for secret contents to not be specified in query text, which makes them easier to keep out of log files, etc. For example:

```
SET VARIABLE my_bearer_token = 'hocus pocus this token is bogus';
CREATE SECRET http (
    TYPE http,
    BEARER_TOKEN getvariable('my_bearer_token')
);
```

You can see that the BEARER_TOKEN field in the secret is set from the `getvariable` function in CREATE SECRET. In the CLI, this is also possible through *environment variables* using `getenv()`. For example, this is now possible:

```
MY_SECRET=asdf duckdb -c \
    "CREATE SECRET http (TYPE http, BEARER_TOKEN getenv('MY_SECRET'))"
```

Unpacking Columns

DuckDB v1.3.0 brings a further boost to the popular `COLUMNS(*)` expression. Previously, *unpacking* the entities into a list was possible by adding a leading `*` character:

```
CREATE TABLE tbl AS SELECT 21 AS a, 1.234 AS b;
SELECT [*COLUMNS(*)] AS col_exp FROM tbl;
```

```
┌──────────────────┐
│     col_exp      │
│ decimal(13,3)[]  │
├──────────────────┤
│ [21.000, 1.234]  │
└──────────────────┘
```

However, this syntax could not be used in tandem with other expressions such as casting:

```
SELECT [*COLUMNS(*)::VARCHAR] AS col_exp FROM tbl;
```

```
Binder Error:
*COLUMNS() can not be used in this place
```

The new UNPACK keyword removes this limitation. The following expression

```
SELECT [UNPACK(COLUMNS(*)::VARCHAR)] AS col_exp FROM tbl;
```

is equivalent to:

```
SELECT [a::VARCHAR, b::VARCHAR] AS col_exp FROM tbl;
```

```
┌──────────────┐
│   col_exp    │
│ varchar[]    │
├──────────────┤
│ [21, 1.234]  │
└──────────────┘
```

Spatial `JOIN` Operator

We added a new specialized join operator as part of the `spatial` extension, which greatly improves the efficiency of *spatial joins*, that is, queries that `JOIN` two geometry columns using specific spatial predicate functions, such as `ST_Intersects` and `ST_Contains`.

Similarly to a `HASH_JOIN`, the `SPATIAL_JOIN` builds a temporary lookup data-structure for the smaller side of the join, except it's an R-Tree, instead of a hash table. What this means for you is that you don't need to create an index first, or do any other pre-processing to optimize your spatial joins. It's all handled by the join operator internally.

While the query optimizer will try to instantiate this new operator for `LEFT`, `OUTER`, `INNER` and `RIGHT` spatial joins, one limitation currently is that the join can only contain a single join condition, or the optimizer will fall back to use a less efficient join strategy.

The following example illustrates how the `SPATIAL_JOIN` operator becomes part of the query plan. It's a relatively small query, but on my machine it executes almost 100× faster than it used to do in DuckDB v1.2.2!

```
LOAD spatial;

-- generate random points
CREATE TABLE points AS
  SELECT
      ST_Point(x, y) AS geom,
      (y * 50) + x // 10 AS id
  FROM
      generate_series(0, 1000, 5) r1(x),
      generate_series(0, 1000, 5) r2(y);

-- generate random polygons
CREATE TABLE polygons AS
  SELECT
      ST_Buffer(ST_Point(x, y), 5) AS geom,
      (y * 50) + x // 10 AS id
  FROM
      generate_series(0, 500, 10) r1(x),
      generate_series(0, 500, 10) r2(y);

-- inspect the join plan
EXPLAIN
    SELECT *
    FROM polygons
    JOIN points ON ST_Intersects(points.geom, polygons.geom);
```

```
            ...
┌─────────────┴─────────────┐
│       SPATIAL_JOIN        │
│  ─────────────────────    │
│     Join Type: INNER      │
│       Conditions:         │
│ ST_Intersects(geom, geom) │
│       ~40401 Rows         │
└─────────────┬─────────────┘
┌─────────────┴─────────────┐┌─────────────┴─────────────┐
│         SEQ_SCAN          ││         SEQ_SCAN          │
│  ─────────────────────    ││  ─────────────────────    │
│      Table: points        ││      Table: polygons      │
│  Type: Sequential Scan    ││  Type: Sequential Scan    │
│       ~40401 Rows         ││       ~2601 Rows          │
└───────────────────────────┘└───────────────────────────┘
```

└──────────────────────────┘ └──────────────────────────┘

For the curious, there are more details in the PR.

Internal Changes

For this release there have also been a large number of internal changes.

We have completed an almost complete re-implementation of DuckDB's *Parquet reader and writer*. This should greatly improve Parquet performance and reliability, and also expanded Parquet feature support for obscure logical types like `UNKNOWN` and `FLOAT16`.

We have also done a large amount of internal changes around the reading of multiple files (e.g., a folder of Parquet files) in an API called the `MultiFileReader`. We have unified the handling of multiple files across many of our file readers, e.g., Parquet, CSV, JSON, Avro, etc. This allows DuckDB to handle e.g. schema differences between multiple files in a unified way.

We have also added a new string compression method, `DICT_FSST`. Before, DuckDB supported *either* dictionary encoding *or* FSST compression ("Fast Static Symbol Table") for strings. Those compression methods could not be mixed within a storage block (265 kB by default). However, we observed a lot of real-world data where part of the block would benefit from dictionary encoding, and another part would benefit from FSST. FSST does not by default duplicate-eliminate strings. This release combines both methods into a new compression method, `DICT_FFST`. This *first* runs dictionary encoding and *then* compresses the dictionary using FSST. Dictionary encoding and FSST-only encoding are also still available. We have also optimized storing validity masks ("which rows are NULL?") in this release, some compression methods (like the new `DICT_FSST`) can handle NULLs internally and this obviates the need for a separate validity mask. Combined, those new features should greatly reduce the storage space required especially for strings. Note that the compression method is automatically picked by DuckDB based on actually observed compression ratios so users will not have to explicitly set this.

Final Thoughts

These were a few highlights – but there are many more features and improvements in this release. There have been over 3,000 commits by over 75 contributors since we released v1.2.2. The full release notes can be found on GitHub. We would like to thank our community for providing detailed issue reports and feedback. And our special thanks goes to external contributors, who directly landed features in this release!

Arrow IPC Support in DuckDB

Publication date: 2025-05-23

Authors: Pedro Holanda, Ian Cook, Dewey Dunnington, Bryce Mecum

TL;DR: DuckDB now supports consuming and producing the Arrow IPC Serialization Format through the `arrow` community extension.

DuckDB and Arrow

In recent years, the Apache Arrow project has gained a lot of traction in the data world, thanks to its columnar format that allows for easy interchange of data between different systems – mostly with a zero-copy approach. Apache Arrow powers, among others, the integration between DuckDB and Polars. In practice, when DuckDB produces or consumes a Polars DataFrame, it is actually using the Arrow columnar format underneath.

The importance of having such a format is also one of the main reasons DuckDB was among the pioneers in integrating with Arrow and implementing an Arrow Database Connectivity (ADBC) interface – particularly because Arrow makes this possible with no additional dependencies, thanks to its C data interface.

But one limitation of Arrow's C data interface is that it exchanges data using pointers (memory addresses). This limits the possibilities if you want to exchange Arrow data between different processes or systems. To overcome this limitation, the Arrow project also specifies the Arrow IPC format, which allows users to efficiently serialize Arrow columnar data and pass it between processes or over a network. This data can be consumed as a stream, either directly from a memory buffer or from a file.

We're thrilled to announce that DuckDB is now able to consume and produce these Arrow streams via the new `arrow` community extension. In this post, we will describe the Arrow IPC serialization format in more detail, show how to install the new `arrow` community extension for DuckDB, and give a demo showing how to use it.

Arrow Interprocess Communication (Arrow IPC)

The Arrow IPC format provides a way of serializing (and optionally compressing) Arrow-formatted data, enabling you to transfer data over a network or store it on disk while keeping it in Arrow format, avoiding the overhead of converting it to a different format. Arrow IPC supports LZ4 and Zstd compression, and when stored as a file, it also supports a file footer that can be used to speed up retrieval and processing by allowing parts of the data to be skipped (similar to the approach used by the Parquet format). When compared to Parquet, the Arrow IPC format has two main benefits:

1. **Ease of implementation:** Writing a low-level Arrow IPC consumer/producer is less complex than writing a Parquet one, especially if the system already integrates with the Arrow format.
2. **Faster encoding and decoding:** The process of encoding and decoding (serializing and deserializing) Arrow data is much simpler and faster than with Parquet. This can yield faster processing times—especially if you are streaming data that does not need to be stored on disk afterwards.

Arrow and Parquet are complementary formats. Parquet's sophisticated system of encoding and compression options typically yields much smaller files, making Parquet a better choice for archival storage. Arrow's ability to eliminate encoding and decoding overheads typically yields faster and more efficient data interchange, making Arrow a better choice for query result transfer and ephemeral caching.

To give you an illustration of just how simple the Arrow IPC format is, consider the following illustration. In the Arrow IPC format, a table is serialized as a sequence of record batches (a collection of records organized in a columnar layout), preceded by their shared schema:

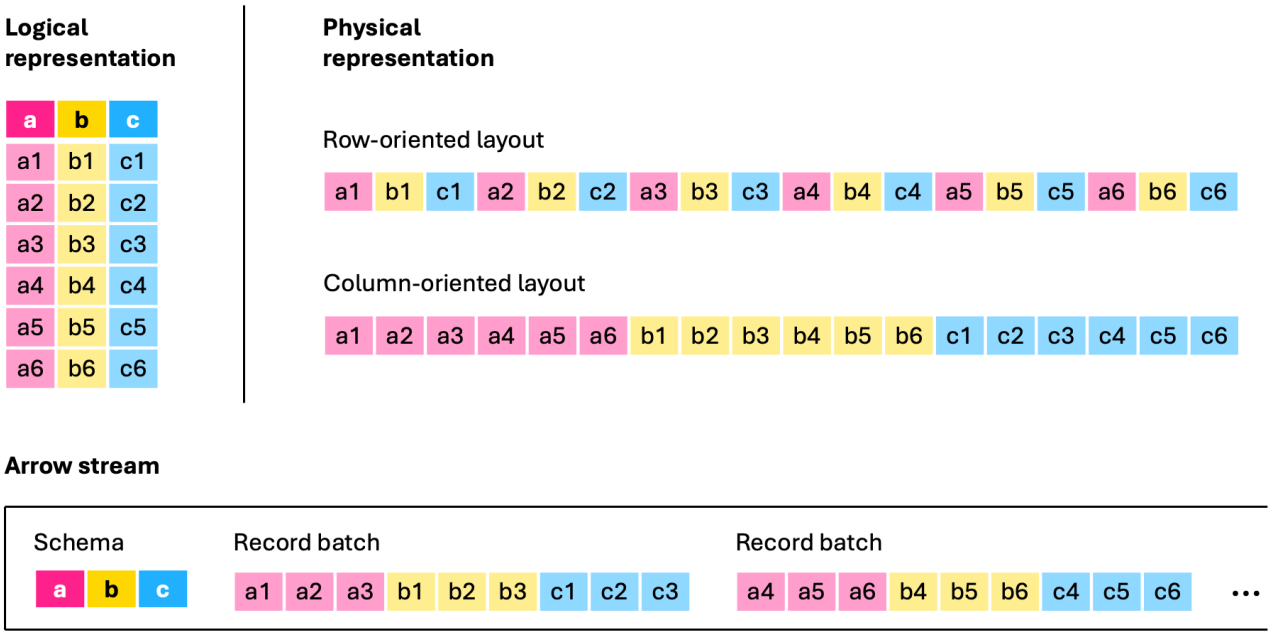


Figure from Apache Arrow Blog: How the Apache Arrow Format Accelerates Query Result Transfer.

Note that in realistic scenarios, record batches are much larger and the figure above is simplified for illustrative purposes.

The Arrow Community Extension

DuckDB has included an integration with the Arrow IPC format for many years, via the (now-deprecated) Arrow core extension. However, the main purpose of this support was to allow DuckDB interoperability with JavaScript, hence it was designed only to read in-memory serialized buffers, and not Arrow IPC files. The extension's code complexity and maintainability were very high, because working with Arrow IPC required having the entire Arrow C++ library as a dependency, as we did not want to write our own serialization and deserialization code for the Arrow IPC format.

More recently, a much smaller Arrow C++ implementation started to gain popularity as a way to interact with Arrow IPC data: the nanoarrow library. Using nanoarrow, we completely redesigned the old DuckDB Arrow extension to have a much smaller dependency, a cleaner codebase, and the ability to scan Arrow IPC files. We also took the opportunity to shift the Arrow DuckDB extension from a core extension to a community extension. This change was made for two main reasons. The first is to enable the Arrow developer and user community to be more involved in building and supporting the extension. The second is to have a release schedule that is not tied to the DuckDB release schedule. In practice, this means that members of the core Arrow developer community can decide when a new version of the extension will be released.

Installing and loading the new Arrow extension is very simple:

```
INSTALL arrow FROM community;
LOAD arrow;
```

Demo

In this demo, we will use the new Arrow DuckDB extension to generate the `lineitem` TPC-H table with scale factor 10 as an Arrow IPC file. While our demo will focus on Arrow IPC data stored in a file, the extension itself also allows you to consume and produce the Arrow IPC format directly as buffers. You can find detailed examples of usage and accepted parameters in the extension's README.

We start off by loading the `arrow` extension and generating our TPC-H tables.

```
LOAD arrow;
```

```
CALL dbgen(sf = 10);
```

To generate the Arrow IPC files we can simply use the `COPY ... TO ...` clause, as follows. We use the recommended file extension `.arrows` since this file is in the Arrow IPC stream format.

```
COPY lineitem TO 'lineitem.arrows';
```

> In this demo, for simplicity, we wrote our table in a single file. However, our Arrow `COPY ... TO ...` clause allows us to set the `chunk_size` and the number of `row_groups` per file. These options allow us to produce data optimized for the best possible performance for your use case. For example, a smaller `chunk_size` may reduce overall performance but benefit streaming scenarios.

We can now run TPC-H query 6 directly on our file using the `read_arrow` function:

```
SELECT
    sum(l_extendedprice * l_discount) AS revenue
FROM
    read_arrow('lineitem.arrows')
WHERE
    l_shipdate >= CAST('1994-01-01' AS date)
    AND l_shipdate < CAST('1995-01-01' AS date)
    AND l_discount BETWEEN 0.05
    AND 0.07
    AND l_quantity < 24;
```

which prints:

```
┌──────────────────┐
│     revenue      │
│  decimal(38,4)   │
├──────────────────┤
│ 1230113636.0101  │
│ (1.23 billion)   │
└──────────────────┘
```

Thanks to replacement scans, you can omit the function `read_arrow` if the filename ends with `.arrow` or `.arrows`. For example:

```
SELECT count(*) FROM 'lineitem.arrows';
```

which prints:

```
┌──────────────────┐
│  count_star()    │
│      int64       │
├──────────────────┤
│     59986052     │
│ (59.99 million)  │
└──────────────────┘
```

> For simplicity, we focus on the scenario of reading a single file, but our reader supports multi-file reading, with functionality on par with the DuckDB Parquet reader.

What if you want to fetch an Arrow IPC stream directly from a server into DuckDB? To demonstrate this, we can start an HTTP file server in the same directory where we saved `lineitem.arrows`. We use the Node.js package serve to do this (instead of Python's built-in `http.server`) because it supports HTTP range requests:

```
npx serve -l 8008
```

Then you can use DuckDB's `httpfs` extension to query the Arrow data over the HTTP(S) protocol:

```
INSTALL httpfs;
LOAD httpfs;
LOAD arrow;

SELECT count(*) FROM read_arrow('http://localhost:8008/lineitem.arrows');
```

which prints the same result:

```
┌──────────────────┐
│  count_star()    │
│      int64       │
├──────────────────┤
│     59986052     │
│ (59.99 million)  │
└──────────────────┘
```

Alternatively, you can use a tool like `curl` to fetch Arrow IPC data from a server and pipe it to DuckDB in the terminal:

```
URL="http://localhost:8008/lineitem.arrows"
SQL="LOAD arrow; FROM read_arrow('/dev/stdin') SELECT count(*);"

curl -s "$URL" | duckdb -c "$SQL"
```

which prints the same result. For other demos of the `arrow` extension, see our arrow-ipc demo repository.

Bonus: Cool Use Cases for Arrow IPC

Running a DuckDB query against data in Arrow IPC format like in the demo above is a pretty neat trick, and it works so well because DuckDB and Arrow are a natural pair due to both using a columnar data layout. However, you may be wondering what else you can do with Arrow IPC data. One of Arrow's main goals is interoperability, and by saving our data in Arrow IPC format, we've opened up many options for connecting with other tools.

For example, we can now work with our data with PyArrow:

```
import pyarrow as pa

with open('lineitem.arrows', 'rb') as source:
   stream = pa.ipc.open_stream(source)
   tbl = stream.read_all()
```

or Polars:

```
import polars as pl

tbl = pl.read_ipc_stream("lineitem.arrows")
```

or ClickHouse:

```
CREATE TABLE
    lineitem
ENGINE MergeTree()
ORDER BY tuple()
AS
    SELECT * FROM file('lineitem.arrows', 'ArrowStream');
```

or any of the numerous other Arrow libraries (available in a dozen different languages) or Arrow-compatible systems.

The benefits of Arrow IPC don't stop there: Arrow IPC is also ideal for larger-than-memory use cases. Using PyArrow, we can memory-map our `lineitem.arrows` file and work with it without reading the entire thing into memory:

```
import pyarrow as pa

with pa.memory_map('lineitem.arrows', 'rb') as source:
    stream = pa.ipc.open_stream(source)
    tbl = stream.read_all()

tbl.num_rows
# => 59986052
```

Then, we can check that PyArrow didn't have to allocate any buffers to hold the data because it all lives on disk:

```
pa.total_allocated_bytes()
# => 0
```

Now we can perform the same query we did in the demo above and show we get the same result:

```
import datetime
import pyarrow.compute as pc

subset = tbl.filter(
    (pc.field("l_shipdate") >= datetime.datetime(1994, 1, 1)) &
    (pc.field("l_shipdate") < datetime.datetime(1995, 1, 1)) &
    (pc.field("l_discount") >= 0.05) &
    (pc.field("l_discount") <= 0.07) &
    (pc.field("l_quantity") < 24.)
)
pc.sum(pc.multiply(subset.column("l_extendedprice"), subset.column("l_discount")))
# => <pyarrow.Decimal128Scalar: Decimal('1230113636.0101')>
```

And, despite `lineitem.arrows` being over 10 GB, PyArrow only had to allocate a fraction of the memory:

```
pa.total_allocated_bytes()
# => 201594240 (192MB)
```

Conclusion & What's Next

In this blog post, we presented the new Arrow community extension, which enables DuckDB users to interact with Arrow IPC streaming buffers and files. Special thanks to Voltron Data for enabling this extension by working with DuckLabs. Below we list our future plans for this extension:

- Support for both ZSTD and LZ4 compression when writing Arrow IPC. DuckDB currently only supports writing uncompressed buffers.
- Support for LZ4 compression when reading Arrow buffers. The reader currently only supports ZSTD or uncompressed buffers.
- Support for writing the Arrow IPC file format containing the file footer, and using the footer to speed up reads.
- Implementation of C API DuckDB functions to produce and consume Arrow IPC data.

If you'd like to work on any of these planned features or suggest other features, or if you find any bugs, feel free to log them in our issue tracker. Happy hacking!

USING KEY in Recursive CTEs

Publication date: 2025-05-23

Authors: Björn Bamberg, Torsten Grust

TL;DR: Recursive CTEs in SQL allow for powerful iterative queries like graph traversals but can be memory-intensive and slow due to repeated row accumulation. DuckDB's new `USING KEY` feature addresses this by treating intermediate results as keyed dictionaries instead of an ever growing set: existing entries can be updated. This can lead to dramatically better performance and memory efficiency, especially in graph algorithms like shortest path and distance-vector routing. It also simplifies query logic by providing direct access to the entire dictionary.

Assembling SQL Queries from Pieces: CTEs

As SQL queries become more complex, managing their readability, modularity and reusability increasingly becomes a challenge. Common Table Expressions (CTEs) were introduced to address these issues by allowing developers to define temporary, named result sets within a query. Similar to functions in programming, CTEs allow a large query to be broken down into logical building blocks, making it easier to understand, maintain and debug.

CTEs are particularly useful for structuring multi-step transformations that might otherwise require deeply nested subqueries or complex joins. By improving both the clarity and structure of SQL code, CTEs have become an essential tool in modern query writing – enabling the clear, declarative expression of even the most sophisticated logic.

Iterate Like It's 1999: Recursive CTEs

To enhance the expressive power of SQL, **recursive CTEs** were introduced in the SQL:1999 standard. These allow a query to reference the results from previous iterations within the same expression, enabling SQL to solve more complex problems such as graph traversal and other iterative computations.

This capability pushes SQL beyond basic data retrieval, allowing for the formulation of complex, iterative logic directly in SQL. In fact, recursive CTEs make SQL Turing-complete, meaning it can theoretically express any computation (given sufficient time and memory).

But how do recursive CTEs work in DuckDB?

Let's look at a simple example to break down the mechanism. Suppose we want to calculate the largest power of 2 that is smaller than 100. We can use a recursive CTE `power` to generate powers of 2 iteratively until we reach that limit. For any row `(a, b, c)` in `power` we will have that `a^b = c`:

```
WITH RECURSIVE power(a, b, c) AS (
    SELECT 2, 0, 1        -- 2^0 = 1
        UNION
    SELECT a, b+1, a * c -- a^(b+1) = a * a^b
    FROM power           -- reads the working table (contains a single row)
    WHERE a * c < 100
)
FROM power;          -- reads the union table (contains all intermediate results)
```

We can divide a recursive CTE into two parts, separated by the `UNION` keyword. The part above the `UNION` is the **non-recursive part** (`SELECT 2, 0, 1` in our example), and the part below is the **recursive part**.

In the recursive part, CTE `power` references itself. This self-reference points to what we call the ***working table***. The working table always holds the rows produced in the **immediately preceding iteration**. And only those.

Here's how it works step by step:

- First, the **non-recursive part** is executed, producing initial rows – in our example, just the row `(2, 0, 1)`. These rows are stored in the ***working table***.
- Then, the **recursive part** is executed using the rows from the working table. Every new row the recursive part produces is stored in the ***intermediate table***, which holds results from the current iteration.
- If the intermediate table is **empty**, the iteration ends.
- Otherwise, we clear the working table and replace it with the contents of the intermediate table – preparing for the next iteration.
- We additionally add the contents of the intermediate table to the ***union table***, which accumulates all intermediate results across iterations.

Here you can see the entries of the three involved tables in each iteration of the CTE:

Iteration	Output of recursive step	Working table	Intermediate table	Union table
0	SELECT 2, 0, 1	∅ (no rows)	(2, 0, 1)	(2, 0, 1)
1	2 * 1 = 2	(2, 0, 1)	(2, 1, 2)	(2, 0, 1) (2, 1, 2)
2	2 * 2 = 4	(2, 1, 2)	(2, 2, 4)	(2, 0, 1) (2, 1, 2) (2, 2, 4)
3	2 * 4 = 8	(2, 2, 4)	(2, 3, 8)	(2, 0, 1) ... (2, 3, 8)
4	2 * 8 = 16	(2, 3, 8)	(2, 4, 16)	(2, 0, 1) ... (2, 4, 16)
5	2 * 16 = 32	(2, 4, 16)	(2, 5, 32)	(2, 0, 1) ... (2, 5, 32)
6	2 * 32 = 64	(2, 5, 32)	(2, 6, 64)	(2, 0, 1) ... (2, 6, 64)
7	2 * 64 = 128	(2, 6, 64)	∅ *stop!*	(2, 0, 1) ... (2, 6, 64)

When the recursive CTE completes, the union table holds the entire result set, including all intermediate rows from every iteration:

```
-- a^b = c
```

a int32	b int32	c int32
2	0	1
2	1	2
2	2	4
2	3	8
2	4	16
2	5	32
2	6	64

The union table provides a complete record of the history of the power computation. This can lead to unnecessary overhead, especially if all we need is the last row – the final result of the recursion. Storing every intermediate value can be inefficient, in particular when the intermediate results aren't required or when working with large datasets (that involve many rows or wide rows, e.g., in the presence of array-typed columns).

Recursive CTEs Suffer from Amnesia

While the union table holds the just mentioned history *once the computation is complete,* recursive CTEs suffer from a case of “amnesia” *while the iteration is going on:* the recursive part only ever sees the intermediate results of the *immediately preceding* iteration. This can be limiting and many of us will have seen how query authors work around this limitation in terms of manually maintained arrays (or similar container structures) that hold information about previous iterations. This may be costly. Then again, enabling the recursive part to access the union table with all its accumulated – potentially sizable – intermediate results may easily incur performance problems while we iterate. A real conundrum.

`USING KEY` to Cure Amnesia

Can we afford to allow the recursive part to access the union table? Yes, but we need a means to control its size. Operating the union table in *append-only mode* thus is a no-go. Instead, let the recursive part optionally *overwrite* existing rows in the union table with fresh information that has been computed in the current iteration. This can reduce the union table's size significantly (see our experiments below).

Starting with version 1.3, DuckDB features a `USING KEY` variant of recursive CTEs that incorporates just this idea.

> If you would like to read more about its origins, check out CIDR 2023. For further information about implementation, see SIGMOD 2025.

The variant introduces two major differences compared to traditional recursive CTEs:

1. It provides access to the working table (as always) as well as **access to the union table** – which we now call the ***recurring table***.
2. Instead of simply appending new rows to the recurring table, the table now functions more like a dictionary (similar to a Python `dict`), allowing **key-based updates**.

To use this new feature, add the `USING KEY (...)` clause to your recursive CTE:

```
WITH RECURSIVE power(a, b, c) USING KEY (a)
    -- key: a, payload: b, c
AS (
    ...
);
```

With `USING KEY`, the schema of the recursive CTE is divided into **key columns** and **payload columns**. The key columns are specified using the `USING KEY (<column names>)`{:.language-sql .highlight} clause, while the remaining columns are treated as payload.

This division affects how the recurring table behaves. Instead of stubbornly appending new rows on each iteration, it acts more like a dictionary: if the recursive part returns a row that hasn't been seen before, it is added to the recurring table as usual. But if a row shares a key with an existing entry in the recurring table, the payload is updated – replacing the previous values for that key in the recurring table.

> If multiple rows with the same key are produced in a single iteration, only the last one is retained. Therefore, you may wish to use the `ORDER BY` clause in the recursive part to control which row is kept.

This approach allows recursive queries to maintain and update state more efficiently, especially for algorithms in which keeping the latest (or "best") value for a given key is crucial.

Overwriting Old Intermediate Results

Let us pick up our recursive example query from above. Now we compute the powers of bases 2 and 3, as long as these are less than 100:

```
WITH RECURSIVE power(a, b, c) USING KEY (a) AS (
    FROM (VALUES (2, 0, 1), (3, 0, 1))   -- 2^0 = 1, 3^0 = 1
        UNION
    SELECT a, b+1, a * c                 -- a^(b+1) = a * a^b
    FROM power                           -- reads the working table (contains two rows)
    WHERE a * c < 100
)
FROM power;                              -- reads the recurring table (contains two rows)
```

We start with two rows in the **non-recursive part**: one with a key (base) of 2 and another with a key of 3. In the **recursive part**, we multiply the intermediate power by the base. This produces two rows, again with the keys 2 and 3.

Unlike traditional recursive CTEs, we **do not append** these two new rows to the recurring table. Instead, we **update the existing rows** with keys 2 and 3 in the recurring table, overwriting their payload values. We thus keep only two rows throughout the entire computation, each holding the current power value for its key (or base) in column a.

Iteration	Output of recursive step	Working table	Intermediate table	Recurring table
0	SELECT 2, 0, 1 SELECT 3, 0, 1	∅ (no rows)	(2, 0, 1) (3, 0, 1)	(2, 0, 1) (3, 0, 1)
1	2 * 1 = 2 3 * 1 = 3	(2, 0, 1) (3, 0, 1)	(2, 1, 2) (3, 1, 3)	(2, 1, 2) (3, 1, 3)
2	2 * 2 = 4 3 * 3 = 9	(2, 1, 2) (3, 1, 3)	(2, 2, 4) (3, 2, 9)	(2, 2, 4) (3, 2, 9)
3	2 * 4 = 8 3 * 9 = 27	(2, 2, 4) (3, 2, 9)	(2, 3, 8) (3, 3, 27)	(2, 3, 8) (3, 3, 27)
4	2 * 8 = 16 3 * 27 = 81	(2, 3, 8) (3, 3, 27)	(2, 4, 16) (3, 4, 81)	(2, 4, 16) (3, 4, 81)
5	2 * 16 = 32 3 * 81 = 243	(2, 4, 16) (3, 4, 81)	(2, 5, 32)	(2, 5, 32) (3, 4, 81)
6	2 * 32 = 64	(2, 5, 32)	(2, 6, 64)	(2, 6, 64) (3, 4, 81)
7	2 * 64 = 128	(2, 6, 64)	∅ *stop!*	(2, 6, 64) (3, 4, 81)

As we can see, the size of the recurring table remains constant. Irrelevant computation history is overwritten which leads to **reduced memory usage**. The final recurring table reads:

```
┌─────────┬─────────┬─────────┐
│    a    │    b    │    c    │  -- a^b = c
│  int32  │  int32  │  int32  │
├─────────┼─────────┼─────────┤
│       2 │       6 │      64 │
│       3 │       4 │      81 │
└─────────┴─────────┴─────────┘
```

This behavior is especially useful with algorithms in which we are interested in the **latest**, **best**, or **smallest** value for a given key. The maximum number of rows in the recurring table is now **bounded by the number of unique keys**. Since the number of distinct keys in use is under control of the recursive part, this can be a powerful advantage when working with sizable datasets.

A Change of Key

Now, *should* you happen to be interested in the history of the computation and are willing to invest the memory space, all that is required is a change of key. With

```
WITH RECURSIVE power(a, b, c)
    USING KEY (a, b)
AS (  -- formerly: USING KEY (a)
    ...
)
FROM power
ORDER BY a, b;
```

the iteration counter (or exponent) in column b is considered to be part of the key, too. The recurring table will thus track unique `(a, b)` combinations (i.e., base, exponent) and we are able to trace what was going on during iteration:

```
┌─────────┬─────────┬─────────┐
│    a    │    b    │    c    │
│  int32  │  int32  │  int32  │
├─────────┼─────────┼─────────┤
│       2 │       0 │       1 │
│       2 │       1 │       2 │
│       2 │       2 │       4 │
│       2 │       3 │       8 │
│       2 │       4 │      16 │
│       2 │       5 │      32 │
│       2 │       6 │      64 │
│       3 │       0 │       1 │
│       3 │       1 │       3 │
│       3 │       2 │       9 │
└─────────┴─────────┴─────────┘
```

```
┌─────────┬─────────┬─────────┐
│    3    │    3    │   27    │
│    3    │    4    │   81    │
└─────────┴─────────┴─────────┘
```

Accessing Relevant History

Another major difference to vanilla recursive CTEs: now that the size of the recurring table is under control, we can afford to reference it directly in the recursive part of the CTE. This allows us to access *any intermediate result* that has not been overwritten yet – regardless of which iteration computed these results. No amnesia anymore! To access the recurring table, simply prefix the CTE name with the pseudo-schema name `recurring`:

```
WITH RECURSIVE t(...) USING KEY (...) AS (
    ...
    FROM recurring.t  -- reads the recurring table while we iterate
)
...;
```

`USING KEY` Can Unlock Performance Advantages

To further highlight the differences between vanilla and key-based CTEs, let's consider a more complex example using a graph dataset – a social network graph.

> The graphs were derived from the LDBC Social Network Benchmark (SNB), which provides a generator for synthetic social network data. To make the graphs more manageable for use with the `WITH RECURSIVE` query, we further filtered them by person name, reducing their density.

In this dataset, nodes represent people while edges correspond to relationships between them. The tables we're working with are `Person(id)`, which contains all existing ids in the network and `knows(person1id, person2id)` in which each row holds a pair of people knowing each other.

> If you would like to try this out for yourself, begin by attaching the database to any DuckDB session.

```
ATTACH 'https://blobs.duckdb.org/data/using-key-graph.duckdb';
USE 'using-key-graph';
```

Our goal is to compute the *shortest path between all pairs of people* in the social network. Since this is a problem with inherent quadratic complexity, we need to closely eye runtime and memory requirements. For each pair, we start by adding a row that has one person as the start node and the other as the target node. We then iteratively explore all individuals known by the start node, continuing the traversal until the target person is reached. Paths through the network are encoded by `via` nodes: to reach the target from the start node, first proceed to the `via` node (an immediate neighbor of the start node) – once there, use that node's `via` entry to continue your traversal.

A recursive CTE that implements this approach reads as follows:

```
WITH RECURSIVE paths(here, current, via, len, there, completed, found) AS (
  SELECT n1.id AS here, n1.id AS current, NULL::BIGINT AS via,
         0 AS len, n2.id AS there, false AS completed, false AS found
    FROM Person AS n1 JOIN Person AS n2 ON (n2.id <> n1.id)
    UNION ALL
  SELECT paths.here,
         person2id AS current,
         coalesce(paths.via, knows.person2id) AS via,
         paths.len+1 AS len,
         paths.there,
         bool_or(knows.person2id = paths.there)
                 OVER (PARTITION BY paths.here, paths.there
                       ROWS BETWEEN UNBOUNDED PRECEDING
                                AND UNBOUNDED FOLLOWING) AS completed,
         knows.person2id = paths.there AS found
      FROM paths
```

```
      JOIN knows ON (paths.current = knows.person1id AND NOT paths.completed)
 )
 SELECT here, there, via, len
 FROM paths WHERE found
 ORDER BY len, here, there;
```

> The working table of recursive CTE `paths` has seven columns:
>
> - `here` refers to the person where the traversal starts,
> - `there` refers to the target person we aim to reach,
> - `via` indicates the immediate neighbor of the start node of the current path,
> - `len` is the length of the current path, which is incremented by one for each step taken in the traversal.
>
> The remaining three columns control the traversal logic and optimize the computation.
>
> - `current` refers to the node that is currently being explored during the traversal,
> - `found` indicates whether the current path has successfully reached the target person,
> - `completed` tracks whether any path with the same `here` and `there` has already reached the target. This prevents further traversal for that pair once a shortest path has been found, thus avoiding the exploration of longer paths.

If we search in a large graph with many edges, the **union table** in a vanilla recursive CTE can grow large, potentially exceeding memory limits. This not only causes significant **performance issues** but can even lead to **query crashes** in extreme cases.

The new key-based CTEs can avoid this problem by changing how the state of the search is maintained. This enables a range of new algorithms to be expressed efficiently in SQL, including algorithms for finding the shortest paths in large graphs.

One such algorithm is **Distance Vector Routing (DVR)**, a method for computing the shortest paths in a network based on node-local routing table that indicate where to "hop next".

- In DVR, each node maintains a **routing table** that records the length of the paths that reach other nodes.
- We use the **recurring table** to store these routing tables for *all* nodes in the network: a row `(here, there, via, len)` indicates that the first hop on the path from node `here` to node `there` is node `via`. The length of the overall path is `len`.
- When a shorter path to a target node is found, the corresponding routing table entry is updated.
- Routing updates found in the last iteration are distributed to neighboring nodes through the **working table**.

To check if a newly incoming routing update improves a currently known path, we perform a lookup in the recurring table. If the new length is smaller than that of the known path, we update the entry and propagate the update to our immediate neighbors. This mechanism allows for efficient path finding even in very large graphs – see below how DVR outperforms the above approach based on a vanilla recursive CTEs.

```
WITH RECURSIVE dvr(here, there, via, len) USING KEY (here, there) AS (
  -- initialize routing tables for all nodes, only the routes to
  -- immediate neighbors are known at this time
  SELECT n.person1id AS here, n.person2id AS there, n.person2id AS via, 1::DOUBLE AS len
  FROM   knows AS n
    UNION
  (SELECT n.person1id AS here, dvr.there, dvr.here AS via, 1 + dvr.len AS len
    FROM dvr -- working table - routing updates shared by neighbors
    JOIN knows AS n ON
        (n.person2id =  dvr.here AND    -- send update only to immediate neighbors
        n.person1id <> dvr.there)       -- no need to store a route to myself
    LEFT JOIN recurring.dvr AS rec ON   -- recurring table (current routing tables)
        (rec.here = n.person1id AND rec.there = dvr.there) -- identify affected routing table entry
    WHERE 1 + dvr.len < coalesce(rec.len, 'Infinity'::DOUBLE) --  does the routing update improve the entry
    in the routing table?
    ORDER BY len -- shortest path first
  )
)
FROM dvr
ORDER BY len, here, there;
```

The larger the social network graphs (A to G), the more pronounced is the performance gap between vanilla recursive CTEs (REC) and the new `USING KEY` variant (KEY). The table below reports **the number of rows processed during each iteration**:

graph	nodes	edges	KEY	REC
A	184	233	744	352,906
B	322	903	8,232	40,732,577
C	424	1,446	19,213	605,859,791
D	484	2,049	30,871	❌
E	1,119	8,809	255,425	❌
F	1,481	14,256	491,880	❌
G	1,618	16,619	607,926	❌

Even for the smallest graph, Graph A, the difference is significant: the REC CTE produces around 350,000 rows, while KEY generates only 744 rows. As the graph size increases, the gap becomes even more striking. In Graph C, with 424 nodes and 1,446 edges, the REC approach processes nearly 1 billion rows, while the KEY method handles less than 20,000 rows. Although this isn't the largest graph in our benchmark, the REC approach is already approaching out-of-memory conditions (OOM, ❌).

This substantial difference in memory usage is only part of the story. The performance of the REC approach also degrades quickly. While both CTEs perform similarly on small graphs, REC becomes significantly slower as the graph grows – eventually crashing – whereas KEY continues to scale smoothly:

And that's the beauty of the new `USING KEY` CTEs. It enables more efficient expression of complex iterative algorithms by providing key-based control over the size of intermediate results that are passed from iteration to iteration – memory pressure lessens, run time performance can go up significantly. If you're working with recursive CTEs in DuckDB, be sure to take advantage of this powerful addition – it may make a significant difference in your queries.

DuckLake: SQL as a Lakehouse Format

Publication date: 2025-05-27

Authors: Mark Raasveldt, Hannes Mühleisen

TL;DR: DuckLake simplifies lakehouses by using a standard SQL database for all metadata, instead of complex file-based systems, while still storing data in open formats like Parquet. This makes it more reliable, faster, and easier to manage.

Faster Dashboards with Multi-Column Approximate Sorting

Publication date: 2025-06-06

Author: Alex Monahan

TL;DR: With any columnar data format, using advanced multi-column sorting when loading data can improve performance on a wide variety of selective read queries. Space filling curve encodings like Morton (Z-Order) and Hilbert approximately sort multiple columns together. Sorting by rounded timestamps adds additional benefits when filtering on recent data.

An animated Hilbert space filling curve by TimSauder – Own work, CC BY-SA 4.0

It is rare to have a dashboard with a single chart. It is even more rare to only query your data one way. It is rarer still to query your data with no filters at all!

When queries read a subset of the total rows, sorting data while loading provides significant benefits. Please have a look at the prior post in this series to understand how this approach works and for practical tips!

In order to see these benefits in a wider variety of real world cases, we have to get a bit more creative. These advanced techniques help when one of these situations occur:

- Queries filter on different columns
- Query patterns are not perfectly predictable
- Queries filter by time and at least one other column

In this post, we describe several advanced sorting strategies, compare them with some experiments (microbenchmarks), and then calculate a metric to measure their effectiveness.

The Strategy

Instead of sorting precisely by one or a small number of columns, we want to sort approximately by a larger number of columns. That will allow queries with different `WHERE` clauses to all benefit from DuckDB's min-max indexes (zone maps). This post introduces two high-level approaches with several examples of each: sorting by space filling curves, and sorting by truncated timestamps.

Space Filling Curves

Both Morton and Hilbert are space filling curve algorithms that are designed to combine multiple columns into an order that preserves some approximate ordering for both columns. One application of space filling curves is in geospatial analytics and it is a helpful illustration.

If a dataset contained the latitude and longitude coordinates of every café on earth (one row per café), but we wanted to sort so that cafés that are physically close to one another are near each other in the list, we could use a space filling curve. Cafés that are somewhat close in both latitude and longitude will receive a similar Morton or Hilbert encoding value. This will allow us to quickly execute queries like "Find all cafés within this rectangular region on a map". (A rectangular region like that is called a bounding box in geospatial-land!) The GIF at the top of this post shows various levels of granularity of a Hilbert encoding – imagine if the x axis were longitude and the y axis were latitude. The "zig-zag" line of the Hilbert algorithm is the list of cafés, sorted approximately.

Both Morton and Hilbert operate on integers or floats, but this post outlines a way to use them to sort VARCHAR columns as well. A SQL macro is used to convert the first few characters of a VARCHAR column into an integer as a pre-processing step.

Truncated Timestamps

More recent data tends to be more useful data, so frequently queries filter on a time column. However, often queries filter on time and on other columns as well.

Don't just sort on the timestamp column! You will miss out on performance benefits. This is because timestamps tend to be so granular, that in practical terms the data is only sorted by timestamp. How many rows of your data were inserted at exactly `2025-01-01 01:02:03.456789`? Probably just one!

To sort on multiple columns as well as a time column, first sort by a truncated timestamp (truncated to the start of the day, week, year, etc.) and then on the other columns.

The "On-Time Flights" Dataset

Ever wondered how likely you will be delayed when you are getting ready to fly? If you are traveling in the US, you can use a government dataset to see how on-time each route is! You can get more details on the dataset at its official site, but I grabbed a Parquet version of the

data from Kaggle. It includes almost 5 years of data on US flights. Combined, the Parquet files are about 1.1 GB. This is fairly small when accessed locally but can be more of a challenge over a slow remote connection. In the rest of the post, we'll run different benchmarks to demonstrate how pruning can help reducing the amount of network traffic required to answer certain queries.

Our first experiment is a hypothetical use case to serve a dashboard where people can explore a single `origin`, a single `dest` (destination), or an `origin-dest` pair. The `origin` and `dest` columns are three-letter airport codes (like `LAX` or `JFK`). We assume once you have picked one of those filters, you want all the rows that match and a few arbitrary columns.

Our second experiment will add in a time component where our hypothetical users could additionally filter to the latest 1, 13, or 52 weeks of data in the dataset.

Experimental Design

We will show a few approaches to sort data approximately to speed up retrieval.

The control groups are:

- `random`: Sort by a hash – the worst case scenario!
- `origin`: Single-column sort on `origin`
- `origin-dest`: Sort on `origin`, then on `destination`

Our alternative sorting approaches are:

- `zipped_varchar`: Sort by one letter at a time, alternating between `origin` and `destination`. – For example, an `origin` of ABC and a `destination` of XYZ would become AXBYCZ, which would then be sorted.
- `morton`: Convert `origin` and `destination` into integers, then order by Morton encoding (Z-order)
- `hilbert`: Convert `origin` and `destination` into integers, then order by Hilbert encoding

> The `zipped_varchar` algorithm is implemented using a SQL macro so no extensions are needed. It also handles arbitrary length strings.

> The Morton and Hilbert encoding functions come from the `lindel` DuckDB community extension, contributed by Rusty Conover. Thank you to Rusty and the folks who have built the `lindel` Rust crate upon which the DuckDB extension is based! The `spatial` extension also contains an `ST_Hilbert` function that works similarly. Thanks to Max Gabrielsson and the GDAL community!

These plots display query runtime when pulling from a DuckDB file hosted on S3. These same techniques can also be successfully applied to the DuckLake integrated data lake and catalog format! DuckLake is the modern evolution of the cloud data lakehouse – take a minute to check out the launch post if you haven't yet! DuckLake has an additional concept of a partition, which enables entire files to be skipped. To take full advantage of DuckLake, first partition your data (by time or otherwise) and then apply the techniques in this post when loading your data.

All experiments were run on an M1 MacBook Pro with DuckDB v1.2.2. The tests were conducted with the DuckDB Python client with results returned as Apache Arrow tables. In between each query, the DuckDB connection is closed and recreated (this time is not measured as a part of the results). This is to better simulate a single user's experience accessing the dashboard in our hypothetical use case.

Experimental Results

We can see that when filtering **by `origin`**, the `origin` single column sort is a full order of magnitude faster than randomly distributed data, requiring only 1.6 seconds instead of 16. Our advanced techniques are nearly as fast as the dedicated sort on `origin`.

However, if you look at the plot **by `destination`**, we see the value of the more balanced techniques. The `origin_dest` approach of just appending `destination` to the list of sorted columns misses out on much of the benefits.

When querying for a **specific `origin` and `destination` pair,** all approaches are significantly faster than random. However, zooming in on the non-random techniques, the more advanced sorting approaches are as fast or slightly faster than sorting by `origin` or by `origin`

and then `destination`. Performance is also faster than the other experiments, as less data needs to be read (since the filter is more selective).

The Hilbert encoding method provides the most consistent performance across the three query patterns. It would be the best choice to support all 3 workloads.

Approximate Time Sorting

Sorting by an "approximate time" involves truncating the time to the nearest value of a certain time granularity. In this experiment, we load the data with 3 different sorting approaches. We first sort on a truncation of the `flightdate` column, truncated to either day, month, or year. We then use our most effective multi-column approach and sort by the Hilbert encoding of `origin` and `dest` next.

When measuring performance, we test each of the query patterns tested previously (filters on `origin`, `destination`, and `origin` / `destination`), but additionally filter on three different time ranges: the latest 1 week, 13 weeks, and 52 weeks.

When querying a week of data for a specific `origin`, sorting at the daily level performs the best. However, sorting by a more approximate time (month or year) performs better when analyzing the latest 13 or 52 weeks of data. This is because the more approximate time buckets allow the Hilbert encoding to separate `origin`s into different row groups more effectively.

Querying by time and destination follows a very similar pattern, with the ideal sort order being highly dependent on how far back in time is analyzed.

Filtering by `origin` and `destination` shows a very different outcome, with a yearly granularity being superior across the board! This is because the `origin` and `destination` filters are much more effective at skipping row groups when the timestamp ordering is very approximate.

As a result, the best compromise across those three workloads is likely to be the very approximate yearly granularity. See! Don't just sort by timestamp!

Table Creation Time

The upfront investment required to see these benefits is the time required to sort the data when inserting. Typically this is still a good trade off – data inserts tend to happen in the background, but nobody wants to be staring at dashboard loading spinners!

Creating a DuckDB table from Parquet files without sorting took slightly over 21 seconds. Each other approach copied from the unsorted DuckDB table and created a new table. The various methods of sorting required similar amounts of time (between 48 and 61 seconds), so we are free to choose the one with the most effective results without considering relative insert performance. However, it is worth noting that overall insert performance slows down by nearly 3× with any sorting.

Table name	Creation time (s)
`from_parquet`	21.4
`random`	60.2
`origin`	51.9
`origin_dest`	48.7
`zipped_varchar`	58.2
`morton`	54.6
`hilbert`	58.5
`hilbert_day`	58.7
`hilbert_month`	53.8
`hilbert_year`	60.2

Measuring Sortedness

The most effective way to choose a sort order is to simulate your production workload, like in the experiments above. However, this is not always feasible or easy to do. Instead, we can measure how well sorted the dataset is on the columns of interest. The metric we will use is "Number of Row Groups per Value". The way to interpret this is that for selective queries to work effectively, each value being filtered on should only be present in a small number of row groups. Smaller is better! However, there are likely diminishing returns when this metric is below the number of threads DuckDB is using.

> Other database systems measure a similar metric called "Clustering Depth".

What can be interpreted from this graph? The random ordering spreads nearly every value across 100 or more row groups (the visualization is truncated at 100 row groups for readability). This highlights how a random ordering will be very slow for selective queries! Sorting by `origin` greatly reduces the number of row groups that each `origin` is spread across, but `destinations` are still spread widely. Sorting by `origin` and `destination` retains the tight distribution of `origins` and slightly improves the metric for `destinations`.

The three advanced approaches (`zipped_varchar`, `morton`, and `hilbert`) are more balanced, with both `origins` and `destinations` only occurring in a moderate number of row groups. While they score worse in the `origin` metric than when sorting directly by `origin`, most `origins` are spread across fewer row groups than a modern laptop processor's core count, so high performance is retained. The Hilbert encoding is the most balanced, so by this metric it would be declared the victor as well!

To calculate this metric, we define several SQL macros using dynamic SQL and the `query` table function.

We can then call the `rowgroup_counts` function on any table and any columns!

```
FROM rowgroup_counts('flights_hilbert', ['origin', 'dest']);
```

table_name	column_name	column_value	total_row_groups	rowgroup_id_count
flights_hilbert	dest	PSG	238	2
flights_hilbert	dest	ESC	238	2
flights_hilbert	origin	YUM	238	2
…	…	…	…	…

The `rowgroup_id_count` column is a measurement of how many distinct row groups that a specific column value is present in, so it is an indicator of how much work DuckDB would need to do to pull all data associated with that value.

> This calculation uses the pseudo-column `rowid`, and it requires data to have been inserted in a single batch to be perfectly accurate. It is only an approximate metric when data is inserted in batches.

Conclusion

When storing data in a columnar file format like a DuckDB database or a Parquet file, approximately sorting by multiple columns can lead to fast read queries across a variety of query patterns. Sorting using the Hilbert encoding provided high performance across multiple workloads, and sorting by year and then by Hilbert performed well when also filtering by time.

Thanks to the "Number of Row Groups per Value" calculation, we can measure the sortedness of any table by any column, and it was predictive of the experimental performance we observed. This way we can experiment with different sorting approaches and quickly forecast their effectiveness, without having to benchmark read workloads each time.

Using these approaches can greatly speed up dashboard interactivity when users are unpredictable (which we always are!). They can also be used in combination with the the additional techniques outlined in the prior post for further benefits. Data is always different, so we get to be creative!

Happy analyzing, folks!

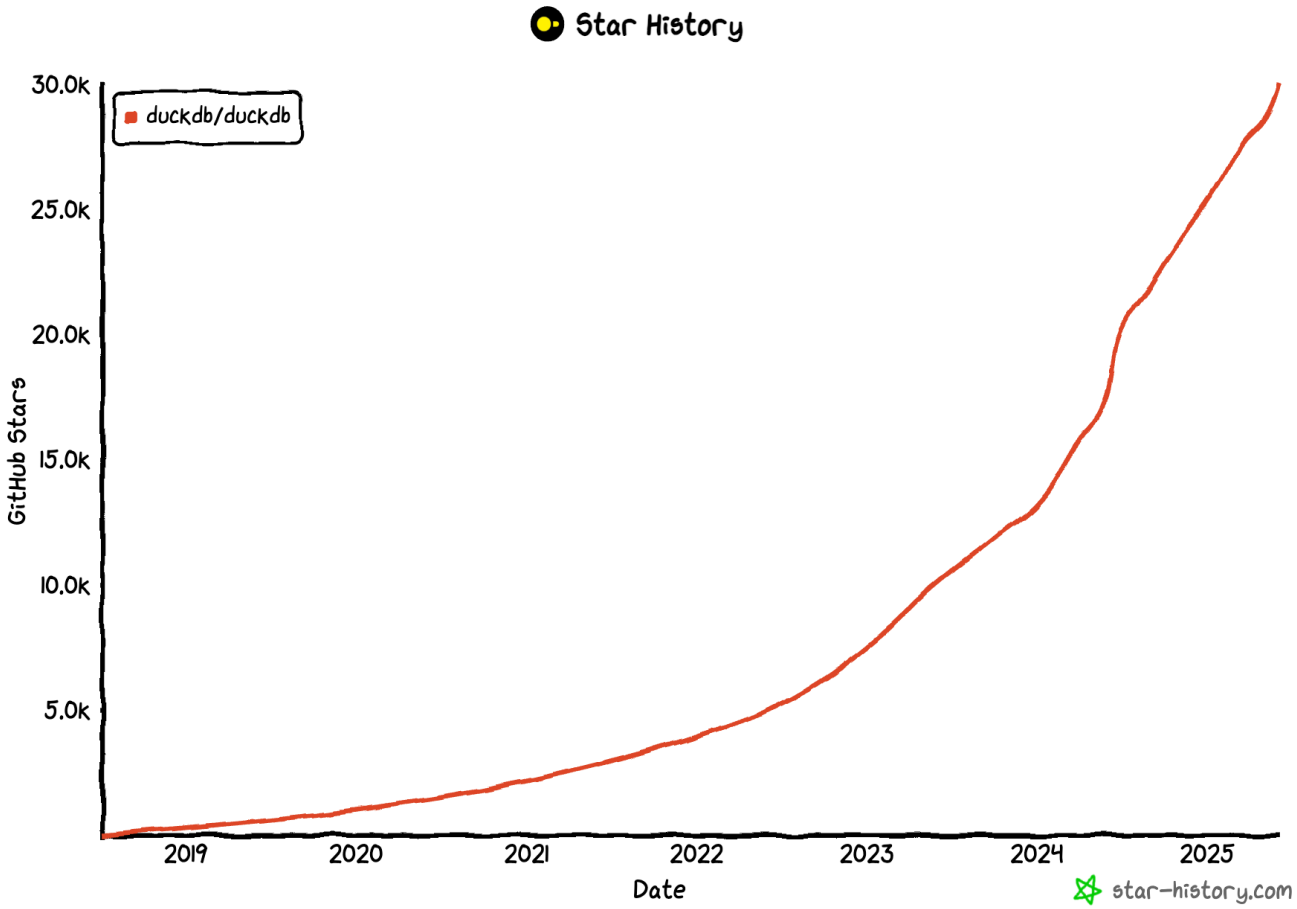
30 000 Stars on GitHub

Publication date: 2025-06-06

Author: The DuckDB team

TL;DR: We have reached 30 000 stars on GitHub! Here's what happened while we accumulated the last 5 000 stars.

The duckdb/duckdb GitHub repository has just passed 30 000 stars!



We would like to use this event to stop for a moment and revisit recent developments in the DuckDB ecosystem since last December, not even six months ago, when we reached 25 000 stars.

Recent Updates

- We released DuckDB 1.2.0 and DuckDB 1.3.0, both packing many new features, performance optimizations and bugfixes.
- We published the DuckLake specification, an open format for SQL-as-a-lakehouse. For details, read the announcement blog post and its coverage in the Register.
- DuckDB now has a built-in UI.
- DuckDB now supports Iceberg REST Catalog, which also allows it to connect to Amazon S3 Tables

- We published the much-requested roadmap.
- Coincidentally, this is the 100th post on our blog.

Metrics

Besides the GitHub stars we have also observed a lot of growth in other metrics:

- Each month, our website now welcomes over 3 million unique visitors – more than double than in December. We also see over 700 TB in traffic from millions of extension downloads. Thanks again to Cloudflare for sponsoring the project.
- Just in this calendar year, we rose 10 positions in the DB Engines ranking from position 55 to 45.
- We now count 20M+ monthly downloads in PyPI.

We would like to reiterate that while we're happy to see these numbers grow, we are not glorifying them and they are not a target per se in accordance with Goodhart's law.

Events

We hosted the first livestreamed DuckCon in Amsterdam and organized another DuckDB Meetup.

Closing Thoughts

That's it for today. We would like to thank our community of users and contributors! We are looking forward to keep growing our community and hope the GitHub stars follow. We'll report back when we reach 40,000 stars.

Lightweight Text Analytics Workflows with DuckDB

Publication date: 2025-06-13

Author: Petrica Leuca

TL;DR: In this post, we demonstrate how to use DuckDB for keyword, full-text, and semantic similarity search with embeddings.

Introduction

Text analytics is a central component of many modern data workflows, covering tasks such as keyword matching, full-text search, and semantic comparison. Conventional tools frequently require complex pipelines and substantial infrastructure, which can pose significant challenges. DuckDB offers a high-performance SQL engine that simplifies and streamlines text analytics. In this post, we will demonstrate how to leverage DuckDB to efficiently perform advanced text analytics in Python.

> The following implementation is executed in a marimo Python notebook, which is available on GitHub, in our examples repository.

Data Preparation

We will be working with a public dataset, available on Hugging Face, which contains English Twitter messages and their classification to one of the following emotions: anger, fear, joy, love, sadness, and surprise.

With DuckDB we are able to access Hugging Face datasets via the `hf://` prefix:

```
from_hf_rel = conn.read_parquet(
        "hf://datasets/dair-ai/emotion/unsplit/train-00000-of-00001.parquet",
        file_row_number=True
    )
from_hf_rel = from_hf_rel.select("""
    text,
    label as emotion_id,
    file_row_number as text_id
""")
from_hf_rel.to_table("text_emotions")
```

> How to access Hugging Face datasets with DuckDB is detailed in the post "Access 150k+ Datasets from Hugging Face with DuckDB".

In the above data we have available only the identifier of an emotion (`emotion_id`), without its descriptive information. Therefore, from the list provided in the dataset description, we create a reference table by unnesting the Python list and retrieving the index for each value with the `generate_subscripts` function:

```
emotion_labels = ["sadness", "joy", "love", "anger", "fear", "surprise"]

from_labels_rel = conn.values([emotion_labels])
from_labels_rel = from_labels_rel.select("""
    unnest(col0) as emotion,
    generate_subscripts(col0, 1) - 1 as emotion_id
""")
from_labels_rel.to_table("emotion_ref")
```

Last we define a relation, by joining the two tables:

```
text_emotions_rel = conn.table("text_emotions").join(
    conn.table("emotion_ref"), condition="emotion_id"
)
```

> By executing `text_emotions_rel.to_view("text_emotions_v", replace=True)` a view with the name `text_emotions_v` will be created, which can be used in SQL cells.

We plot on a bar chart the emotion distribution to have an initial understanding of our data:

Keyword Search

Keyword search is the most basic form of text retrieval, matching exact words or phrases in text fields using SQL conditions such as `CONTAINS`, `ILIKE`, or other DuckDB text functions. It is fast, requires no preprocessing, and works well for structured queries like filtering logs, matching tags, or finding product names.

For example, getting the texts and their emotion label containing the phrase `excited to learn` is a matter of applying `filter` on the relation defined above:

```
text_emotions_rel.filter("text ilike '%excited to learn%'").select("""
    emotion,
    substring(
        text,
        position('excited to learn' in text),
        len('excited to learn')
    ) as substring_text
""")
```

emotion varchar	substring_text varchar
sadness	excited to learn
joy	excited to learn
joy	excited to learn
fear	excited to learn
fear	excited to learn
joy	excited to learn
joy	excited to learn
sadness	excited to learn

One common step in text processing is to split text into tokens (keywords), where raw text is broken down into smaller units (typically words), that can be analyzed or indexed. This process, known as *tokenization*, helps convert unstructured text into a structured form suitable for keyword search. In DuckDB this process can be implemented with the `regexp_split_to_table` function, which will split the text based on the provided regex and return each keyword on a row.

> This step is case sensitive, therefore it is important to convert all text to a consistent case (by applying `lcase` or `ucase`) before processing.

In the below code snippet we select all the keywords by splitting the text on one or more non-word characters (anything except `[a-zA-Z0-9_]`):

```
text_emotions_tokenized_rel = text_emotions_rel.select("""
    text_id,
    emotion,
    regexp_split_to_table(text, '\\W+') as token
""")
```

In the tokenization step, we usually exclude common words (such as `and`, `the`), called *stopwords*. In DuckDB we implement the exclusion by applying an `ANTI JOIN` on a curated CSV file hosted on GitHub:

```
english_stopwords_rel = duckdb_conn.read_csv(
    "https://raw.githubusercontent.com/stopwords-iso/stopwords-en/refs/heads/master/stopwords-en.txt",
    header=False,
).select("column0 as token")
```

```
text_emotions_tokenized_rel.join(
    english_stopwords_rel,
    condition="token",
    how="anti",
).to_table("text_emotion_tokens")
```

Now that we have tokenized and cleaned the text, we can implement keyword search by ranking the match with similarity functions, such as Jaccard:

```
text_token_rel = conn.table(
    "text_emotion_tokens"
).select("token, emotion, jaccard(token, 'learn') as jaccard_score")

text_token_rel = text_token_rel.max(
    "jaccard_score",
    groups="emotion, token",
    projected_columns="emotion, token"
)

text_token_rel.order("3 desc").limit(10)
```

emotion varchar	token varchar	max(jaccard_score) double
fear	learn	1.0
surprise	learn	1.0
love	learn	1.0
joy	lerna	1.0
sadness	learn	1.0
fear	learner	1.0
anger	learn	1.0
joy	leaner	1.0
fear	allaner	1.0
anger	learner	1.0
10 rows		3 columns

We can also visualize the data to gain insights. One simple and effective approach is to plot the most frequently used words. By counting token occurrences across the dataset and displaying them in bubble plots, we can quickly identify dominant themes, repeated keywords, or unusual patterns in the text. For example, we plot the data by using a scatter facet plot per emotion:

From the above plot, we observe repeated keywords, such as `feel - feeling`, `love - loved - loving`. In order to de-duplicate such data, we need to look at the word stem rather than at the word itself. This brings us to full-text search.

Full-Text Search

The Full-Text Search (FTS) DuckDB extension is an experimental extension, which implements two main full-text search functionalities:

- the `stem` function, to retrieve the word stem;
- the `match_bm25` function, to calculate the Best Match score.

By applying `stem` to the token column, we can now visualize the most frequently used word stem in our data:

We observe that `feel` and `love` appear only once and new word stems are plotted, such as `support`, `surpris`.

While the `stem` function can be used standalone, the `match_bm25` one requires the build of a FTS index, a special index that allows fast and efficient searching of text by indexing the words (tokens) in a column:

```
conn.sql("""
    PRAGMA create_fts_index(
```

```
        "text_emotions",
        text_id,
        "text",
        stemmer = 'english',
        stopwords = 'english_stopwords',
        ignore = '(\\.|[^a-z])+',
        strip_accents = 1,
        lower = 1,
        overwrite = 1
    )
""")
```

In the FTS index creation we are using the same list of English stopwords as in the tokenization process, by saving it into a table, named `english_stopwords`. The index is case insensitive due to the `lower` parameter, which will lowercase the text automatically.

> **Warning.** The index can be created only on tables and it requires a unique identifier of the text. It also needs to be rebuilt when the underlying data has been modified.

Once the index has been created, we can rank the match between the `text` column and the phrase `excited to learn`:

```
text_emotions_rel
.select("""
    emotion,
    text,
    emotion_color,
    fts_main_text_emotions.match_bm25(
        text_id,
        'excited to learn'
    )::decimal(3, 2) as bm25_score
""")
.order("bm25_score desc")
.limit(10)
```

Out of the 10 returned texts, displayed above in a table plot, 2 are poor matches to our search input; likely due to BM25 scoring being skewed by common terms or differences in document length.

Semantic Search

Compared to keyword and full-text search, *semantic search* takes into account the meaning and context of the text. Instead of just looking for exact words, it uses techniques like vector embeddings to capture the underlying concepts. Semantic search, which is case insensitive, can be implemented in DuckDB, by making use of the (also experimental) Vector Similarity Search extension.

The vector embeddings of a (list of) text can be calculated with the `sentence-transformers` library and the pre-trained model `all-MiniLM-L6-v2`:

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')

def get_text_embedding_list(list_text: list[str]):
    """
    Return the list of normalized vector embeddings for list_text.
    """
    return model.encode(list_text, normalize_embeddings=True)
```

For example, `get_text_embedding_list(['excited to learn'])` will return:

```
array([[ 3.14795598e-02, -6.66208193e-02,  1.05058309e-02,
         4.12571728e-02, -8.67664907e-03, -1.79746319e-02,
        ...
        -2.50727013e-02, -3.00881546e-03,  1.55055271e-02]], dtype=float32)
```

We register the model inference function as a Python User Defined Function and create a table with a column of type `FLOAT[384]` to load the embeddings into:

```
conn.create_function(
    "get_text_embedding_list",
    get_text_embedding_list,
    return_type='FLOAT[384][]'
)

conn.sql("""
    create table text_emotion_embeddings (
        text_id integer,
        text_embedding FLOAT[384]
    )
""")
```

With the Python UDF, we save, in batches, the model output in `text_emotion_embeddings`:

```
for i in range(num_batches):
    selection_query = (
        duckdb_conn.table("text_emotions")
        .order("text_id")
        .limit(batch_size, offset=batch_size*i)
        .select("*")
    )

    (
        selection_query.aggregate("""
            array_agg(text) as text_list,
            array_agg(text_id) as id_list,
            get_text_embedding_list(text_list) as text_emb_list
        """).select("""
            unnest(id_list) as text_id,
            unnest(text_emb_list) as text_embedding
        """)
    ).insert_into("text_emotion_embeddings")
```

> We wrote about model inference in DuckDB in the post Machine Learning Prototyping with DuckDB and scikit-learn.

We can now perform semantic search by using the cosine distance between the vector embedding of our search text, `excited to learn`, and the embedding of the `text` field:

```
input_text_emb_rel = conn.sql("""
    select get_text_embedding_list(['excited to learn'])[1] as input_text_embedding
""")

text_emotions_rel
.join(conn.table("text_emotion_embeddings"), condition="text_id")
.join(input_text_emb_rel, condition="1=1")
.select("""
        text,
        emotion,
        emotion_color,
        array_cosine_distance(
            text_embedding,
            input_text_embedding
        )::decimal(3, 2) as cosine_distance_score
    """)
.order("cosine_distance_score asc")
.limit(10)
```

Interesting to observe that the phrase `i am excited to learn and feel privileged to be here` didn't make it in the top 10 results in our semantic search!

Similarity Joins

Vector embeddings are most known for their usability in search engines, but they can be used in a variety of text analytics use cases, such as topic grouping, classification or semantic matching between documents. The VSS extension provides vector similarity joins, which can be used to conduct these types of analytics.

For example, we show in the below heatmap chart the number of texts for each combination of emotion labels, where the x-axis corresponds to the semantic matching between the text and the emotion, the y-axis to the classified emotion, and the color indicates the count of texts assigned to each pair:

It is particularly noticeable that from the 6 emotions, only `sadness` has a strong semantic match with the text classified with the same label. As with full-text search, semantic search is affected by differences in document length (in this case the emotion keyword versus a text).

Hybrid Search

While each type of search has its own applicability, we observed that some results are not as expected:

- keyword search and full-text search do not take word meaning into account;
- semantic search scored synonyms higher than the search text.

In practice, the three search methods are combined and used to perform a "hybrid search", in order to improve the search relevance and accuracy. We start by calculating the score for each type of search, by implementing custom logic, such as a check on the emotion:

```
if(
    emotion = 'joy' and contains(text, 'excited to learn'),
    1,
    0
) exact_match_score,

fts_main_text_emotions.match_bm25(
        text_id,
        'excited to learn'
)::decimal(3, 2) as bm25_score,

array_cosine_similarity(
    text_embedding,
    input_text_embedding
)::decimal(3, 2) as cosine_similarity_score
```

The BM25 score is ranked in descending order and the cosine distance in ascending order. In hybrid search, we use the `array_cosine_similarity` score to ensure the same sort order (in this case descending).

> cosine similarity = 1 - cosine distance

Because the BM25 score can be, in theory, unbounded, we need to scale the score to the interval `[0, 1]` by implementing the min-max normalization:

```
max(bm25_score) over () as max_bm25_score,
min(bm25_score) over () as min_bm25_score,
(bm25_score - min_bm25_score) / nullif((max_bm25_score - min_bm25_score), 0) as norm_bm25_score
```

The hybrid search score is calculated by applying a weight to the BM25 and cosine similarity scores:

```
if(
    exact_match_score = 1,
    exact_match_score,
    cast(
        0.3 * coalesce(norm_bm25_score, 0) +
        0.7 * coalesce(cosine_similarity_score, 0)
        as
        decimal(3, 2)
```

```
    )
) as hybrid_score
```

And here are the results! Much better, don't you think?

Conclusion

In this post, we showed how DuckDB can be used for text analytics by combining keyword, full-text, and semantic search techniques. Using the experimental `fts` and `vss` extensions with the `sentence-transformers` library, we demonstrated how DuckDB can support both traditional and modern text analytics workflows.

Discovering DuckDB Use Cases via GitHub

Publication date: 2025-06-27

Author: Petrica Leuca

TL;DR: In this post, we use the GitHub API to find repositories that mention DuckDB, then use DuckDB itself to parse and query the results efficiently with SQL.

Introduction

One of the persistent challenges of maintaining an open-source library is the lack of visibility into how, where, and by whom it's being used. At DuckDB we look at the awesome-duckdb repository, which contains a curated list of DuckDB libraries, tools and resources. To supplement this, we wanted to automate insights from GitHub in order to uncover new and emerging projects using DuckDB. In this post we write about how we use DuckDB to query the GitHub API, export the data into a Markdown file and visualize the historical data, generated from Git commits.

Data Retrieval from GitHub

GitHub provides a series of REST API which can be used to retrieve data from GitHub. Among them, the search repositories API returns a list of public repositories whose name, description or readme matches the search text.

> GitHub also has a search code API in order to match on code dependency, but it does not return all the information the search repository provides. One could use the search code API and for each result to retrieve the repository information from the repository API.

In order to use the GitHub API, we create an access token with read access on public repositories. With the access token, saved as an environment variable, we create a secret in DuckDB to be used by the API request:

```
import os

import duckdb


def get_duckdb_conn():
    conn = duckdb.connect()
    conn.sql(f"""
        CREATE SECRET http_auth (
            TYPE http,
            BEARER_TOKEN '{os.getenv("READ_PUBLIC_REPO_TOKEN")}',
            SCOPE 'https://api.github.com/search'
        );
    """)
    return conn
```

With the DuckDB Python client's `read_json` function we are able to query the API response with SQL. The above secret is passed automatically by DuckDB to the API call:

```
duckdb_conn = get_duckdb_conn()

api_response = duckdb_conn.read_json(
    "https://api.github.com/search/repositories?q=duckdb"
)
```

The API call returns the following response:

total_count int64	incomplete_results boolean	items struct(id bigint, node_id varchar,...
3837	false	[{'id': 138754790, 'node_id': ...

We can analyze the response by:

- getting the list of columns with `api_response.columns`:

```
['total_count', 'incomplete_results', 'items']
```

- getting the column types `api_response.types`:

```
[
    BIGINT,
    BOOLEAN,
    STRUCT(id BIGINT, node_id VARCHAR, "name" VARCHAR, full_name VARCHAR, private BOOLEAN, "owner"
    STRUCT(login VARCHAR, id BIGINT, node_id VARCHAR, avatar_url VARCHAR, gravatar_id VARCHAR, url
    VARCHAR, html_url VARCHAR, followers_url VARCHAR, following_url VARCHAR, gists_url VARCHAR,
    starred_url VARCHAR, subscriptions_url VARCHAR, organizations_url VARCHAR, repos_url VARCHAR,
    events_url VARCHAR, received_events_url VARCHAR, "type" VARCHAR, user_view_type VARCHAR, site_
    admin BOOLEAN), html_url VARCHAR, description VARCHAR, fork BOOLEAN, url VARCHAR, forks_url
    VARCHAR, keys_url VARCHAR, collaborators_url VARCHAR, teams_url VARCHAR, hooks_url VARCHAR, issue_
    events_url VARCHAR, events_url VARCHAR, assignees_url VARCHAR, branches_url VARCHAR, tags_url
    VARCHAR, blobs_url VARCHAR, git_tags_url VARCHAR, git_refs_url VARCHAR, trees_url VARCHAR,
    statuses_url VARCHAR, languages_url VARCHAR, stargazers_url VARCHAR, contributors_url VARCHAR,
    subscribers_url VARCHAR, subscription_url VARCHAR, commits_url VARCHAR, git_commits_url VARCHAR,
    comments_url VARCHAR, issue_comment_url VARCHAR, contents_url VARCHAR, compare_url VARCHAR,
    merges_url VARCHAR, archive_url VARCHAR, downloads_url VARCHAR, issues_url VARCHAR, pulls_url
    VARCHAR, milestones_url VARCHAR, notifications_url VARCHAR, labels_url VARCHAR, releases_url
    VARCHAR, deployments_url VARCHAR, created_at TIMESTAMP, updated_at TIMESTAMP, pushed_at TIMESTAMP,
    git_url VARCHAR, ssh_url VARCHAR, clone_url VARCHAR, svn_url VARCHAR, homepage VARCHAR, size
    BIGINT, stargazers_count BIGINT, watchers_count BIGINT, "language" VARCHAR, has_issues BOOLEAN,
    has_projects BOOLEAN, has_downloads BOOLEAN, has_wiki BOOLEAN, has_pages BOOLEAN, has_discussions
    BOOLEAN, forks_count BIGINT, mirror_url JSON, archived BOOLEAN, disabled BOOLEAN, open_issues_
    count BIGINT, license STRUCT("key" VARCHAR, "name" VARCHAR, spdx_id VARCHAR, url VARCHAR, node_id
    VARCHAR), allow_forking BOOLEAN, is_template BOOLEAN, web_commit_signoff_required BOOLEAN, topics
    VARCHAR[], visibility VARCHAR, forks BIGINT, open_issues BIGINT, watchers BIGINT, default_branch
    VARCHAR, permissions STRUCT("admin" BOOLEAN, maintain BOOLEAN, push BOOLEAN, triage BOOLEAN, pull
    BOOLEAN), score DOUBLE)[]
]
```

- getting the number of items with `api_response.select("len(items)")`:

len(items) int64
30

The total number of repositories mentioning DuckDB is 3837, but the `items` object contains only 30. This is due to pagination that is set to 30 records per page by default. We can retrieve a maximum of 100 records by using the `per_page` query parameter. Additionally, we use the `page` query parameter in order to retrieve the results for each page:

```
(
    duckdb_conn
    .read_json("https://api.github.com/search/repositories?q=duckdb&per_page=100&page=2")
    .select("len(items)")
)
```

```
┌──────────────┐
│ len(items)   │
│    int64     │
├──────────────┤
│          100 │
└──────────────┘
```

Because GitHub has a limit of returning maximum 1000 results in the search API and we retrieve the data daily, we added a filter to retrieve only the repositories which had a push in the last 7 days:

```
api_url = "https://api.github.com/search/repositories?q=duckdb"

last_pushed_date = (datetime.now() - timedelta(days=7)).strftime("%Y-%m-%d")

api_url = f"{api_url}+pushed:>={last_pushed_date}"

duckdb_conn.read_json(f"{api_url}&per_page=100&page=1").to_table("github_raw_data")
```

We store the first page in a table called `github_raw_data` and calculate how many pages we need to retrieve based on the total count returned by the first call. Then, with the `insert_into` method, we append the data from each page into `github_raw_data`:

```
for page in range(2, number_pages + 1):
    logger.info(f"Fetching {page} out of {number_pages}")
    (
        duckdb_conn
        .read_json(f"{api_url}&per_page=100&page={page}")
        .insert_into("github_raw_data")
    )
```

Saving Data to a Markdown File

With the data available in a table, we can continue our data processing, by using the `unnest` function, which will flatten the `items` returned by the API:

```
(
    duckdb_conn.table("github_raw_data")
    .select("unnest(items, recursive := true)")
)
```

By using `recursive` unnesting, the STRUCT objects within `items` will be flattened too; below is a sample of the columns derived from `items` after unnesting:

```
['id',
 'node_id',
 'name',
 'full_name',
 'private',
 'login',
 'id',
 ...
 'pull',
 'score']
```

Our scope is to create a Markdown file containing a table, with the following format:

- **name**, containing the name, link, description, license and owner of the repository;
- **topics**, the list of topics of the repository;
- **stars**, the number of stars;
- **open issues**, the number of open issues;
- **forks**, the number of forks;
- **created at**, when the repository was created;
- **updated at**, when the repository was last updated.

To retrieve the name field we use `concat_ws` and `concat` functions, in order to generate the hyperlink and text with the Markdown newline character(<br>):

```
selection_query = (
    duckdb_conn.table("github_raw_data")
    .select("unnest(items, recursive := true)")
    .select("""
        concat_ws(
            '<br>',
            concat('[', name, '](', concat('https://github.com/', full_name),')'),
            coalesce(description, ' '),
            concat('**License** ', coalesce(name_1, 'unknown')),
            concat('**Owner** ', login)
        ) as repo_details
    """)
)
```

The above returns:

```
repo_details = [duckdb-web](https://github.com/duckdb/duckdb-web)<br>DuckDB website and
documentation<br>**License** MIT License<br>**Owner** duckdb
repo_details = [duckdb](https://github.com/duckdb/duckdb)<br>DuckDB is an analytical in-process SQL
database management system<br>**License** MIT License<br>**Owner** duckdb
...
```

We also calculate metrics such as stars, open issues count and forks, which we sum into a field `activity_count`. A repository will end up in the list only if it has an `activity_count` greater than 3 and if it is not a fork. From above we also see that repositories owned by DuckDB are returned, therefore we filter them out too:

```
selection_query.filter("""
    login != 'duckdb'
    and not fork
    and activity_count >= 3
""")
```

A Markdown table is similar to a CSV file, separated by pipe (|), but it must:

- contain the separator at the beginning and end of each row;
- between the table header and the first table row there needs to be a row with dashes (called delimiter row).

In order to export the data as a Markdown file we are applying a few tricks. The first one is to select the header by adding dummy columns, at the beginning and end, containing `NULL`:

```
duckdb_conn.sql("""
    select
        NULL,
        'Name',
        'Topics',
        'Stars',
        'Open Issues',
        'Forks',
        'Created At',
        'Updated At',
        NULL
""")
```

We then union the above header with the delimiter row:

```
.union(
    duckdb_conn.sql("""
    select
        NULL as '',
        '--' as "Name",
        '--' as "Topics",
        '--' as "Stars",
```

```
        '--' as "Open Issues",
        '--' as "Forks",
        '--' as "Created At",
        '--' as "Updated At",
        NULL as ''
""")
)
```

Returning:

NULL int32	'Name' varchar	'Topics' varchar	'Stars' varchar	'Open Issues' varchar	'Forks' varchar	'Created At' varchar	'Updated At' varchar	NULL int32
NULL	Name	Topics	Stars	Open Issues	Forks	Created At	Updated At	NULL
NULL	--	--	--	--	--	--	--	NULL

And finally we union with the initial GitHub selection query and export the data to CSV, by disabling the header export:

```
(
    ...
    .union(selection_query)
).to_csv("./exported_records.md", sep="|", header=False)
```

Below is a sample of the `exported_records.md` file, rendered as HTML:

Name	Topics	Stars	Open Issues	Forks	Created At	Updated At
tailpipeselect * from logs;**License** GNU Affero General Public License v3.0**Owner** turbot	[aws, azure, detections, devops, duckdb, forensics, gcp, incident-response, log-analysis, mitre-attack, open-source, parquet, siem, tailpipe, threat-detection]	438	41	9	2024-04-18 02:44:35	2025-06-17 11:57:42

Why a Markdown file you may wonder. Because it is rendered automatically by GitHub and there is no need to host our tiny data application somewhere else. In fact, we copy the above table to README, such that it is displayed automatically on the first page of the repository:

```
echo '# Repositories using `duckdb`' > README.md
cat exported_records.md >> README.md
```

Another way to create the README file is by using the `string_agg` function:

```
selected_data = (
    selection_query
    .select("""
        concat(
            '|',
            concat_ws(
                '|',
                repo_details,
                topics, stars,
                open_issues,
                forks,
                created_at,
                updated_at
            )
        ) as line
    """)
    .string_agg(
        'line',
        sep='|\n'
```

```
    )
).fetchone()[0]

with open('README.md', 'w') as readme_file:
    readme_file.write("# Repositories using `duckdb`\n")
    readme_file.write("|Name|Topics|Stars|Open Issues|Forks|Created At|Updated At|\n")
    readme_file.write(f"{duckdb_conn.sql("select concat(repeat('|--', 7),'|')").fetchone()[0]}\n")
    readme_file.write(f"{selected_data}|")
```

In the above code snippet we concatenate the columns with the pipe character and then we aggregate the records into a string, separated by `|\n` in order to add a pipe and newline at the end of each line. We then use Python to write to README the title of the page, the Markdown table header, the delimiter row (by using the `repeat` function) and the data itself.

> The data size is very small, each commit having a README of approx. 35 KB.

Automating with GitHub Workflow

With GitHub workflows we automated the above data processing steps. We first define a Makefile in our project to configure the steps needed to be executed in the workflow:

```
search-repos:
    uv run using_duckdb/search_repositories.py

readme:
    echo '# Repositories using `duckdb`' > README.md && \
    cat exported_records.md >> README.md
```

In the settings of the GitHub repository we create a repository secret which contains the value of the API token generated above. We can then configure an environment variable in our workflow and define the processing steps:

```
name: Search

on:
  workflow_dispatch:
  schedule:
    - cron: '37 5 * * *'

env:
  READ_PUBLIC_REPO_TOKEN : ${{ secrets.READ_PUBLIC_REPO_TOKEN }}

jobs:
  make-readme-md:
    runs-on: ubuntu-latest

    permissions:
      contents: write

    steps:
      - uses: actions/checkout@v4
        with:
          ref: ${{ github.head_ref }}
          fetch-depth: 0

      - name: Install uv
        uses: astral-sh/setup-uv@v6

      - name: Install the project
        run: uv sync

      - name: Get search results
        run: make search-repos

      - name: Export current results
```

```
        run: make readme

      - name: Get datetime
        id: datetime
        run: echo "datetime=$(date -u +'%Y-%m-%dT%H:%M:%SZ')" >> $GITHUB_OUTPUT

      - uses: stefanzweifel/git-auto-commit-action@v5
        with:
          commit_message: 'Update as of ${{ steps.datetime.outputs.datetime }}'
          file_pattern: 'README.md'
```

With git-auto-commit-action we delegate to GitHub the right to create a commit with the updated `README.md` file:

Visualizing Historical Data

Because we store the search results into the README file and we update it daily with Git, we can retrieve the history of the search by getting the Git difference for each commit:

```
git-log:
    echo '|Name|Topics|Stars|Open Issues|Forks|Created At|Updated At|' > git_log.md && \
    echo '|--|--|--|--|--|--|--|' >> git_log.md && \
    git log --follow -p --pretty=format:"" -- README.md | grep '^+|\[' | sed 's/^+//' >> git_log.md && \
    git diff -- README.md | grep '^+|\[' | sed 's/^+//' >> git_log.md
```

In the above Makefile command we:

- create a file, `git_log.md`, which has on the first line the header of the Markdown table;
- add to `git_log.md` the delimiter row;
- for each commit, append to `git_log.md` the new lines of the README differences;
- append to `git_log.md` the new lines from the current change of README.

The `git_log.md` contains now the entire history of appended records to the README file and it can be used to display a bubble chart with the repositories mentioning DuckDB:

The above plot is generated with Plotly by providing as data source a DuckDB Python relation:

```
px.scatter(
    duckdb_conn.read_csv(
        "./git_log.md"
        skiprows=2
    )
        .select("""
        #2 as repo,
        #3 as topics,
        #4 as stars,
        #5 as open_issues,
        #6 as forks,
        strftime(#7, '%B %d, %Y, %H:%m' ) as created_at,
        #8 as updated_at,
        if(stars + forks + open_issues = 0, 0, log(stars + forks + open_issues)) as log_activity_count,
        substr(repo, position('[' in repo) + 1 , position(']' in repo) - 2) as repo_name,
        count(distinct updated_at::date) over (partition by repo_name) as number_of_updates,
        row_number() over (partition by repo_name order by updated_at desc) as rn
    """)
    .filter("rn = 1")
    .order("updated_at"),
    x="updated_at",
    y="log_activity_count",
    labels={
        "updated_at": "Updated Date",
        "log_activity_count": "Activity count, based on stars, open issues and forks"
    },
    ...
```

> Using Markdown as a file format for our historical data might not be the best solution, because – among others – it does not support schema evolution. While we cannot do changes in the middle of the file (except column renames), we can remove the last column or add new ones at the end, due to the way DuckDB reads malformed CSV files.

Conclusion

In this post we showed how DuckDB can be used to process API requests and historical data from git commits. Some of the repositories returned by the GitHub search API made it to awesome-duckdb, which is our go-to list for curated DuckDB related projects; for example tailpipe, an open source Security Information and Event Management for instant log insights, and preswald, a Wasm packager for Python-based interactive data apps.

DuckLake 0.2

Publication date: 2025-07-04

Author: Mark Raasveldt

TL;DR: We released the DuckLake v0.2 standard, which is supported by DuckDB v1.3.1's `ducklake` extension.

Spatial Joins in DuckDB

Publication date: 2025-08-08

Author: Max Gabrielsson

TL;DR: DuckDB v1.3.0 significantly improved the scalability of geospatial joins with a dedicated `SPATIAL_JOIN` operator.

Introduction

Spatial joins are join operations that match rows based on the (geo-)spatial relationship between column(s). In practice they are often used to answer questions like "which of these points are within which of these polygons?". Being able to connect datasets based on the **physical location(s)** they model is fundamental to the whole field of geospatial data science, but at a high level, also an extremely powerful way to correlate and enrich otherwise disparate data sources, anchor insights to the real world, and more often than not tell a great story with your analysis.

Ever since its inception, DuckDB's `spatial` extension has provided a `GEOMETRY` column type to represent locations, regions and shapes, along with plenty of *spatial predicate* functions to use when `JOIN`ing. However, it wasn't until recently in DuckDB v1.3.0 that spatial joins really became *scalable* thanks to the introduction of the dedicated `SPATIAL_JOIN` query **operator**.

We managed to hastily sneak in some information about this new operator towards the end of the v1.3.0 release notes, but we figured we got enough to talk about to justify a separate blog post. So in this post, we will take a closer look at some of the challenges involved in optimizing spatial joins, and how the new `SPATIAL_JOIN` operator addresses them to achieve new levels of efficiency.

What Does a Spatial Join Look Like Again?

Performing a spatial join in SQL is actually very straightforward. There is no special syntax or magic incantation required. Like you probably could've guessed, you just have to `JOIN` two tables with a `GEOMETRY` column using some *spatial predicate*. Below is a simple example that joins two tables `some_table` and `another_table` on the condition that the geometries in `some_table.geom` and `another_table.geom` "intersect", using the `ST_Intersects` spatial predicate function:

```
SELECT *
FROM some_table
JOIN another_table
  ON ST_Intersects(some_table.geom, another_table.geom);
```

Let's try to do something more advanced. We're going to analyze the NYC *Citi Bike Trip* dataset, which contains around 58 million rows representing rental bike trips in New York City, including the start and end locations of each trip. We want to find neighborhoods in NYC that have the most bike trips starting in them. Therefore, we join the bike trip data with a dataset of NYC neighborhood polygons. We then count and group the number of trips starting in each neighborhood, and finally sort and limit the results to return the top-3 neighborhoods in which most trips originate. We've compiled the datasets and extracted the relevant columns for this example into two tables, `rides` and `hoods`, into a 218 MB DuckDB database file.

Let's make sure the `spatial` extension is installed and loaded:

```
INSTALL spatial; -- Install the spatial extension
LOAD spatial;    -- Load the spatial extension
```

Our query looks like this:

```
SELECT neighborhood, count(*) AS num_rides
FROM rides
JOIN hoods ON ST_Intersects(rides.start_geom, hoods.geom)
GROUP BY neighborhood
ORDER BY num_rides DESC
LIMIT 3;
```

neighborhood varchar	num_rides int64
Midtown	6253835
Chelsea	6238693
East Village	4047396

This query joins the 58,033,724 rides with the 310 neighborhood polygons and takes about **30 seconds** to run on my laptop (MacBook with an M3 Pro CPU and 36 GB RAM).

While 30 seconds might not seem that impressive at first glance (a `HASH_JOIN` over similarly sized input would finish an order of magnitude faster), I'm actually pleased DuckDB is able to execute a spatial join at this scale in a time frame that is, if not great, still acceptable for exploratory analysis (on a laptop no less!). To understand the difference in execution time, compared to, e.g., a `HASH_JOIN`, we first need to take a closer look at how DuckDB used to do spatial joins and why spatial joins are so challenging to optimize. Then, we'll dive into how the new `SPATIAL_JOIN` operator changes the game.

How DuckDB (Used to) Do Spatial Joins

Spatial Predicates

The first thing to understand is that a *spatial predicate* is really just a function that evaluates some spatial relation between two geometries and returns `true` or `false`, e.g., "does a *contain* b" or "is a *within distance* x of b". In theory you could write your own spatial predicate function, but in practice, you will probably use one of the many that come with the `spatial` extension. The nuances of the different spatial predicates are beyond the scope of this post, but here is a quick overview of the most commonly used ones, taking two geometries a and b:

Function	Description
`ST_Intersects(a, b)`{:.language-sql .highlight}	Whether a intersects b
`ST_Contains(a, b)`{:.language-sql .highlight}	Whether a contains b
`ST_ContainsProperly(a, b)`{:.language-sql .highlight}	Whether a contains b without b touching a's boundary
`ST_Within(a, b)`{:.language-sql .highlight}	Whether a is within b
`ST_Overlaps(a, b)`{:.language-sql .highlight}	Whether a overlaps b
`ST_Touches(a, b)`{:.language-sql .highlight}	Whether a touches b
`ST_Equals(a, b)`{:.language-sql .highlight}	Whether a is equal to b
`ST_Crosses(a, b)`{:.language-sql .highlight}	Whether a crosses b
`ST_Covers(a, b)`{:.language-sql .highlight}	Whether a covers b
`ST_CoveredBy(a, b)`{:.language-sql .highlight}	Whether a is covered by b
`ST_DWithin(a, b, x)`{:.language-sql .highlight}	Whether a is within distance x of b

> We use `ST_Intersects` in the example query above to keep it simple, even if `ST_Contains` or `ST_ContainsProperly` might be more appropriate for point-in-polygon joins.

Since all the above spatial predicates live in the `spatial` extension, DuckDB's query planner doesn't really know anything about them, except their names and that they are functions that take two `GEOMETRY` arguments and return a boolean value. This means that vanilla DuckDB on its own can't apply any special optimizations to them, and has to treat them like **any other function** in the database.

Optimization Challenges

When DuckDB plans a join where the join condition is an arbitrary function, it normally can't use any of the advanced built-in join strategies like `HASH_JOIN` or `RANGE_JOIN` which are optimized for equality or range comparisons. Instead DuckDB has to fall back to the simplest join strategy, which is to perform a **nested loop join** (NLJ), i.e., to evaluate the join condition for every possible pair of rows in the two tables. In pseudo-code, a nested loop join would look something like this:

```
for row_a in table_a:
    for row_b in table_b:
        if join_condition(row_a, row_b):
            emit(row_a, row_b)
```

This is the most general way to implement a join, but it is also the least efficient. Since the *complexity* of the join is `O(nm)`, where `n` and `m` are the number of rows in the two tables, the time it takes to perform the join grows quadratically with the size of the input.

Complexity Challenges

Obviously, quadratic complexity quickly becomes infeasible for large joins, but DuckDB's raw execution power usually makes it bearable when joining small- to medium-sized tables.

However, what makes the nested loop join strategy impractical for spatial joins in particular, even at small- to medium-scale is that spatial predicates can be *very* computationally expensive to evaluate in the first place. This is mostly due to the sheer complexity of the algorithms involved, but also due to the denormalized nature of geometries, in which a single geometry can contain a very large number of points. Also, besides the inherent theoretical complexity, the reality in `spatial` is that most of the spatial predicates implemented with the help of third-party libraries require a deserialization step to convert the internal binary format of the `GEOMETRY` column to an executable data structure. This usually also requires allocating memory outside of DuckDB's own memory management system, which increases pressure and lock contention on the global memory allocator, limiting the effectiveness of additional parallelism.

To illustrate how this plays out in practice, let's take a look at the query plan for the example query above, but with the `SPATIAL_JOIN` optimization disabled to force DuckDB to use a nested loop join instead:

Experiment

```
LOAD spatial; -- Load the spatial extension

-- Disable the spatial join optimizer
SET disabled_optimizers = 'extension';

-- Print the new plan, using EXPLAIN to verify
-- that we are using a nested loop join
EXPLAIN SELECT neighborhood, count(*) AS num_rides
FROM rides
JOIN hoods ON ST_Intersects(rides.start_geom, hoods.geom)
GROUP BY neighborhood
ORDER BY num_rides DESC
LIMIT 3;
```

Executing this query, with the `BLOCKWISE_NL_JOIN` (blockwise nested loop join), now takes **30 minutes** on my laptop. I've also run the query with a subset of number of rows in the `rides` table to see how the performance scales with the size of the input. The results are as follows:

Number of rows in `rides`	Execution time (s)	Execution time (min)
1,000,000	30.8 s	0.5 min
10,000,000	310.3 s	5.2 min
58,033,724	1799.6 s	30.0 min

This is obviously not great. Running a 1/58th of the full dataset already takes longer than the original query with the spatial join optimization enabled. And processing the full dataset takes almost half an hour to run! This is a clear indication that the nested loop join strategy is not suitable for spatial joins at scale, and we really needed to do something about it.

The IE-Join Optimization

In order to improve the situation, one of the first optimizations implemented in the early versions of the `spatial` extension was a re-write rule for the query planner.

Since evaluating the spatial predicate function itself is so expensive, its a good idea to try to do some *cheap* filtering of potential join matches first in order to reduce the number of pairs that actually need to go through the precise spatial relationship check. Luckily for us, it is relatively easy to do this for most spatial predicates since they all have a common property. Almost all spatial predicates *imply intersection* in some way, and **if two geometries intersect, their bounding boxes must also intersect**.

The *bounding box*, sometimes called *minimum bounding rectangle* (MBR) of a geometry is the smallest rectangle that fully contains the geometry. Because bounding boxes basically represent the minimum and maximum x and y coordinates of all vertices in the geometry, they are relatively efficient to compute by just scanning over the geometry a single pass. Additionally, checking if two bounding boxes intersect is also very cheap, as it can be done in constant time with a few simple less-than and greater-than comparisons.

Since bounding box intersection can be expressed as a series of inequality checks, we can make use of DuckDB's existing **inequality join** functionality to perform a bounding-box join! Explaining the inner workings of inequality joins is beyond the scope of this post because Richard have already done a great job of that in his blog post on range joins, but the gist is that it is a join type that can efficiently join two tables using a series of inequality operators like <, >, <=, >=, etc.

To leverage this, we made the following changes to `spatial`:

- Cache an approximate bounding box for each geometry in the serialized binary representation of the `GEOMETRY` column so that we don't have to recompute it every time we need to evaluate a spatial predicate.
- Introduce a re-write rule to change any inner nested loop join operators with a spatial predicate into two new operators:
  - `PIECEWISE_MERGE_JOIN`, a "IE" (inequality) range-join operator joining on the intersection of the bounding boxes, written as a series of `BETWEEN` clauses on the min/max values of the bounding boxes
  - `FILTER` operator that filters the resulting rows on the actual spatial predicate function.

To show how this affects the query plan, we're going to go back in time and use DuckDB **v1.2.0** (which does contain the `PIECEWISE_MERGE_JOIN` optimization, but **not** the `SPATIAL_JOIN` operator) to re-run the query from above.

```
LOAD spatial; -- Load the spatial extension

-- Check that we are using DuckDB v1.2.0
PRAGMA version;
```

library_version varchar	source_id varchar
v1.2.0	5f5512b827

```
-- Print the new query plan, using EXPLAIN
EXPLAIN SELECT neighborhood, count(*) AS num_rides
FROM rides
```

```
JOIN hoods ON ST_Intersects(rides.start_geom, hoods.geom)
GROUP BY neighborhood
ORDER BY num_rides DESC
LIMIT 3;
```

As you can see, the PIECEWISE_MERGE_JOIN operator has replaced the BLOCKWISE_NL_JOIN operator, and the join condition has been rewritten to use the approximate bounding boxes of the geometries (ST_Extent_Approx) instead of directly joining on the spatial predicate. The FILTER operator is added on top to only apply the spatial predicate functions on the rows that pass the initial bounding box check.

This looks a lot more complicated than the original nested loop join, so how does it perform?

Number of rows in rides	Execution time
1,000,000	2.3 s
10,000,000	19.6 s
58,033,724	107.6 s

Much better, we managed to reduce the execution time from 30 minutes to **just under 2 minutes** for the full dataset! This was a huge improvement. Without requiring any custom operator code, we were able to leverage DuckDB's existing inequality join functionality to significantly speed up spatial joins as well.

However, this approach still has some drawbacks:

- This re-write is only possible to do for INNER joins.
- We now have to store bounding boxes for each geometry, which increases the memory footprint of the GEOMETRY column, and we also have to recompute it whenever create or modify a GEOMETRY.
- The PIECEWISE_MERGE_JOIN operator has to sort the two input tables, which requires a lot of memory and can be really slow for large tables once the data no longer fits in memory.

DuckDB has since optimized sorting considerably, and spatial and its functions have also received various optimizations that improve performance slightly. Nonetheless this was always somewhat of a stop-gap solution to make spatial joins at least usable in the common case. How can we do better?

The Spatial Join Operator

R-Tree Indexing

To quickly recap how an r-tree works, it is a balanced tree data structure where internal nodes contain bounding boxes that cover the bounding boxes of their child nodes, and leaf nodes contain a bounding box for the geometry and a pointer to the row in the table they correspond to. This makes it possible to quickly look up all geometries that intersect a given bounding box by traversing the tree from the top down, only following branches that intersect the search box.

One of the motivations for implementing the RTREE index in the spatial extension last year, was to eventually be able to replace PIECE-WISE_MERGE_JOIN optimization with an index-based join strategy. However, after implementing the RTREE, we realized that creating an R-tree based index on top of existing data is actually surprisingly fast. Instead of requiring the user to always have to load data into a table and then index it just to accelerate their spatial joins, what if we just create a temporary R-tree index on-the-fly when executing the join? Kind of like how we create a temporary hash table when performing a HASH_JOIN.

Joining with R-Trees

This is exactly what the new SPATIAL_JOIN operator does. It takes two input tables, buffers all the input of the “right” table (i.e., table that the join-order-optimizer expects to be smaller), builds an R-tree index on top of the collected input, and then performs a join by looking

up each row of the "left" table in the R-tree index as they arrive. The idea is exactly the same as with a HASH_JOIN, but instead of using a hash table, we use an R-tree as our acceleration data structure to look up the matching rows.

While being a lot more complex to implement compared to the PIECEWISE_MERGE_JOIN-based approach, this has several advantages:

- The query re-write rule is much simpler, we just have to detect a nested loop join with a spatial predicate and replace it with a SPATIAL_JOIN operator. No other modifications to the query plan are needed.
- Only the smaller side input needs to be materialized and (sorted to construct the R-tree). The left side can be streamed in parallel, and matching rows can be emitted immediately as they are found.
- The hierarchical structure of the R-tree allows us to prune more of the right side input more accurately
- Since all logic is in one operator, we don't need to rely on a separate FILTER operator and can therefore support LEFT, RIGHT and FULL OUTER joins as well as INNER joins.

Experiment

Just to show what this plan looks like, this is what you get when you run the same query but on DuckDB v1.3.0 or later with the SPATIAL_JOIN operator enabled:

```
LOAD spatial; -- Load the spatial extension

-- Print the new query plan, using EXPLAIN
EXPLAIN SELECT neighborhood, count(*) AS num_rides
FROM rides
JOIN hoods ON ST_Intersects(rides.start_geom, hoods.geom)
GROUP BY neighborhood
ORDER BY num_rides DESC
LIMIT 3;
```

Nice, we're back to a single join condition again. So how does this perform? We already saw in the first example that DuckDB v1.3.2 was able to execute the original query in about 30 seconds using the SPATIAL_JOIN operator, but here are the precise results for all the different sizes and join strategies for comparison. All queries were run on the same Apple MacBook M3 Pro with 36 GB of memory and timings averaged over 3 runs:

Number of rows in rides	Nested loop join	Piecewise merge join	Spatial join
1,000,000	30.8 s	2.3 s	0.5 s
10,000,000	310.3 s	19.6s s	4.8 s
58,033,724	1799.6 s	107.6 s	28.7 s

The SPATIAL_JOIN operator is able to execute the full 58 million row dataset faster than the original naive nested loop join could execute the 1 million row dataset. That's a 58× improvement!

Limitations and Future work

Even though we've made a lot of improvements, spatial joins in DuckDB are not quite perfect just yet. Like always, there's plenty more to do, and we have a few ideas for how to improve the SPATIAL_JOIN operator even further.

Larger-Than-Memory Build Sides

The current implementation of the SPATIAL_JOIN operator buffers the entire right side input in memory to build the R-tree index. This means that it can only handle right side inputs that fit in memory. We plan to add support for larger-than-memory R-trees by partitioning the right side into smaller separate R-trees, and then merging the results of the join from each partition, similar to how the HASH_JOIN operator handles larger-than-memory hash tables.

Increased Parallelism

DuckDB's execution engine dynamically adjusts the number of threads used in a query based on the number of rows scanned, but since the spatial predicates are so expensive to evaluate, we could potentially benefit from more parallelism as the spatial joins are mostly CPU bound. In particular, we are considering buffering and partitioning the left side of the join as well, which would allow us to spawn our own worker tasks within the operator to evaluate the join condition across a number of threads decoupled from the cardinality of the input. This would increase memory usage and prevent the join from being streamed, but we've been able to employ a similar technique when constructing the HNSW (Hierarchical Navigable Small Worlds) index over in the `vss` extension to great effect.

Faster Predicate Functions

Most predicate functions in the `spatial` extension are implemented using third-party libraries which carry some overhead as we are unable to integrate them as tightly with DuckDB's execution engine and memory management, often requiring some sort of deserialization or conversion step. This can be quite expensive, especially for large geometries. To demonstrate impact of this in practice, we can compare the performance of joining on `ST_Intersects(x, y)` with the somewhat equivalent `ST_DWithin(x, y, 0)`, by comparing the previous results with the timings of the following query:

```
SELECT neighborhood, count(*) AS num_rides
FROM rides
JOIN hoods ON ST_DWithin(rides.start_geom, hoods.geom, 0)
GROUP BY neighborhood
ORDER BY num_rides DESC
LIMIT 3;
```

Number of rows in `rides`	Spatial join (`Intersects`)	Spatial join (`DWithin`)
1,000,000	0.5 s	0.1 s
10,000,000	4.8 s	0.7 s
58,033,724	28.7 s	4.3 s

Now, `ST_DWithin` is not the strictly the same as `ST_Intersects`, so this is not an apples-to-apples comparison. Even if they produce the same output in this example, you should not rush to replace all instances of `ST_Intersects` with `ST_DWithin` in your code just to squeeze out some extra performance. In fact, `ST_DWithin` is technically doing *more* work as the algorithm is more complex. The reason why it is still faster in this case, and the reason why we're bringing it up, is because its implementation in the `spatial` extension was recently changed to our own highly optimized native implementation. This implementation goes the extra mile in avoiding memory allocation and copying, and the results speak for themselves. This goes to show how optimizing the spatial predicate functions themselves can significantly improve the performance of spatial joins. We are actively working on adding optimized versions of the rest of the commonly used spatial predicates (like `ST_Intersects`, `ST_Contains`, `ST_Within`, etc.) to the `spatial` extension and expect to see similar performance improvements across the board. So stick with `ST_Intersects` when appropriate, and look forward to more performance gains in the future!

Advanced Join Conditions

The `SPATIAL_JOIN` operator currently only supports the case where the join condition is a single spatial predicate function. We plan to add support for more complex join conditions that can include comparisons, arithmetic operations, and other functions in combination with the spatial predicates. This will allow users to express more complex spatial relationships in their joins, e.g., `JOIN ... ON ST_Intersects(a.geom, b.geom) AND a.id = b.id`{:.language-sql .highlight}.

`ANTI` / `SEMI` Joins

The `SPATIAL_JOIN` operator currently only supports `INNER`, `LEFT`, `RIGHT` and `FULL OUTER` joins. We plan to add support for `ANTI` and `SEMI` joins in the future.

Conclusion

Spatial joins are a powerful way to connect and enrich datasets based on their geospatial relationship. DuckDB's `spatial` extension provides a wide assortment of spatial predicate functions that can be used to perform spatial joins in SQL. While the performance of spatial joins has been challenging to deal with in the past due to the complexity of the spatial predicates and the limitations of the join strategies, the new `SPATIAL_JOIN` operator introduced in DuckDB v1.3.0 significantly raises the efficiency and scalability of these workloads, making spatial joins a first-class citizen in DuckDB's execution engine.

Basic Feature Engineering with DuckDB

Publication date: 2025-08-15

Author: Petrica Leuca

TL;DR: In this post, we show how to perform essential machine learning data preprocessing tasks, like missing value imputation, categorical encoding, and feature scaling, directly in DuckDB using SQL. This approach not only simplifies workflows, but also takes advantage of DuckDB's high-performance, in-process execution engine for fast, efficient data preparation.

Introduction

Data preprocessing is a necessary step in any machine learning workflow, affecting both the model's effectiveness and the ease of maintenance. While scikit-learn is commonly used for preprocessing due to its integration with the broader Python ecosystem, DuckDB offers a practical alternative by enabling SQL-based data transformations within Python. Its declarative syntax supports modular workflows, making preprocessing steps easier to isolate, inspect, and debug. In addition, DuckDB's support for efficient querying of columnar data formats and its ability to persist preprocessing logic as SQL scripts contribute to more reproducible and transparent pipelines.

Data Preparation

We will be working with a synthetic financial transactions dataset from Kaggle, which contains generic information used for fraud detection in financial transactions.

```
CREATE TABLE financial_trx AS
    FROM read_csv('https://blobs.duckdb.org/data/financial_fraud_detection_dataset.csv');
```

We start by analyzing the data by executing SUMMARIZE:

```
FROM (SUMMARIZE financial_trx)
SELECT
    column_name,
    column_type,
    count,
    null_percentage,
    min;
```

column_name varchar	column_type varchar	count int64	null_percentage decimal(9,2)	min varchar
transaction_id	VARCHAR	5000000	0.00	T100000
timestamp	TIMESTAMP	5000000	0.00	2023-01-01 00:09:26.241974
sender_account	VARCHAR	5000000	0.00	ACC100000
receiver_account	VARCHAR	5000000	0.00	ACC100000
amount	DOUBLE	5000000	0.00	0.01
transaction_type	VARCHAR	5000000	0.00	deposit
merchant_category	VARCHAR	5000000	0.00	entertainment
location	VARCHAR	5000000	0.00	Berlin
device_used	VARCHAR	5000000	0.00	atm
is_fraud	BOOLEAN	5000000	0.00	false
fraud_type	VARCHAR	5000000	96.41	card_not_present
time_since_last_transaction	DOUBLE	5000000	17.93	-8777.814181944444
spending_deviation_score	DOUBLE	5000000	0.00	-5.26

velocity_score	BIGINT	5000000	0.00	1
geo_anomaly_score	DOUBLE	5000000	0.00	0.0
payment_channel	VARCHAR	5000000	0.00	ACH
ip_address	VARCHAR	5000000	0.00	0.0.102.150
device_hash	VARCHAR	5000000	0.00	D1000002
18 rows				5 columns

Feature Encoding

From the above data stats, we see that there are a few category columns, such as `transaction_type`, `merchant_category` and `payment_channel`. Because most machine learning models expect numerical inputs, this type of data is converted to a numerical representation. This process is called *encoding,* and it can be done in multiple ways. In the following, we showcase a few common encoding techniques in SQL.

> In this post, we use several of DuckDB's "friendly SQL" features, including the FROM-first syntax and prefix aliases.

One-Hot Encoding

When applying one-hot encoding over a category column, each distinct value is transposed into its own column and it gets the value 1 for match and 0 for not a match:

```
FROM financial_trx
SELECT DISTINCT
    transaction_type,
    deposit_onehot: (transaction_type = 'deposit')::INT,
    payment_onehot: (transaction_type = 'payment')::INT,
    transfer_onehot: (transaction_type = 'transfer')::INT,
    withdrawal_onehot: (transaction_type = 'withdrawal')::INT
ORDER BY transaction_type;
```

transaction_type varchar	deposit_onehot int32	payment_onehot int32	transfer_onehot int32	withdrawal_onehot int32
deposit	1	0	0	0
payment	0	1	0	0
transfer	0	0	1	0
withdrawal	0	0	0	1

Another way to one-hot encode is by using the PIVOT statement:

```
PIVOT financial_trx
ON transaction_type
USING coalesce(max(transaction_type = transaction_type)::INT, 0) AS onehot
GROUP BY transaction_type;
```

In the above statement we:

- pivot on the category column, `transaction_type`;
- the pivot condition is for `transaction_type` to match each value of itself;
- we convert boolean to integer and apply max over the match;
- we alias the transposed columns as the value of `transaction_type` suffixed by `_onehot`.

If there are more category columns to be one-hot encoded then PIVOT can be used in subqueries or WITH clauses:

```
WITH onehot_trx_type AS (
    PIVOT financial_trx
    ON transaction_type
```

```
    USING coalesce(max(transaction_type = transaction_type)::INT, 0) AS onehot
    GROUP BY transaction_type
), onehot_payment_channel AS (
    PIVOT financial_trx
    ON payment_channel
    USING coalesce(max(payment_channel = payment_channel)::INT, 0) AS onehot
    GROUP BY payment_channel
)
SELECT
    financial_trx.*,
    onehot_trx_type.* LIKE '%\_onehot' ESCAPE '\',
    onehot_payment_channel.* LIKE '%\_onehot' ESCAPE '\'
FROM financial_trx
INNER JOIN onehot_trx_type USING (transaction_type)
INNER JOIN onehot_payment_channel USING (payment_channel);
```

> In the above query we are retrieving all "onehot"-suffixed columns by using the LIKE operator on column names.

Ordinal Encoding

Ordinal encoding assigns a unique identifier to each categorical value and it is usually applied when there is a certain hierarchy in the categorical values. For example, we can assign the identifier with the `row_number` function and ordering by the transaction type:

```
WITH trx_type_ordinal_encoded AS (
    SELECT
        transaction_type,
        trx_type_oe: row_number() OVER (ORDER BY transaction_type) - 1
    FROM (
        SELECT DISTINCT transaction_type
        FROM financial_trx
    )
)
SELECT
    transaction_type,
    trx_type_oe,
    number_trx: count(*)
FROM financial_trx
INNER JOIN trx_type_ordinal_encoded USING (transaction_type)
GROUP BY ALL
ORDER BY trx_type_oe;
```

transaction_type varchar	trx_type_oe int64	number_trx int64
deposit	0	1250593
payment	1	1250438
transfer	2	1250334
withdrawal	3	1248635

Label Encoding

Similarly to ordinal encoding, label encoding assigns a unique identifier, but it does not take order into account and it is, usually, applied to output data:

```
WITH trx_type_label_encoded AS (
    SELECT
        transaction_type,
        trx_type_le: row_number() OVER () - 1
    FROM (
        SELECT DISTINCT transaction_type
```

```
        FROM financial_trx
    )
)
SELECT
    transaction_type,
    trx_type_le,
    number_trx: count(*)
FROM financial_trx
INNER JOIN trx_type_label_encoded USING (transaction_type)
GROUP BY ALL
ORDER BY trx_type_le;
```

transaction_type varchar	trx_type_le int64	number_trx int64
deposit	0	1250593
withdrawal	1	1248635
payment	2	1250438
transfer	3	1250334

Another way to achieve the above is by using list functions, such as `array_agg`, to create an array with the distinct values, and `list_position` to extract the position of each value in the array:

```
WITH trx_ref AS (
    SELECT trx_type_values: array_agg(DISTINCT transaction_type)
    FROM financial_trx
)
SELECT
    transaction_type,
    trx_type_le: list_position(trx_type_values, transaction_type) - 1,
    number_trx: count(*)
FROM
    financial_trx,
    trx_ref
GROUP BY ALL
ORDER BY trx_type_le;
```

transaction_type varchar	trx_type_le int32	number_trx int64
payment	0	1250438
deposit	1	1250593
transfer	2	1250334
withdrawal	3	1248635

> The above queries are non-deterministic, therefore a sort or storing the data in a reference table might be required for incremental processing.

Feature Scaling

One other common data preprocessing step in machine learning is to scale numerical features, such that the values of different features are brought to a similar range or distribution. *Scaling,* also known as *feature normalization* or *standardization,* involves transforming features so they have comparable magnitudes; typically by rescaling them to a fixed range (like 0 to 1) or adjusting them to have zero mean and unit variance. This process is required because many algorithms rely on distance calculations or gradient updates, which can be skewed if features vary widely in scale.

If encoding is performed on the initial raw data (due to the need of knowing the entire categorical values list), scaling requires to split the data into training and testing data sets, in order to avoid data leakage. In DuckDB, we can split the data by sampling it:

```
SET threads = 1;

CREATE TABLE financial_trx_training AS
FROM financial_trx
USING SAMPLE 80 PERCENT (reservoir, 256);

SET threads = 8;

CREATE TABLE financial_trx_testing AS
FROM financial_trx
ANTI JOIN financial_trx_training USING (transaction_id);
```

> We configure DuckDB to use a single-thread during sampling and set a `seed` to make sure that the sampling is reproducible. We also apply the `reservoir` sampling strategy to have exactly 80% of the records in the resulting sample.

Standard Scaling

Standard scaling is a preprocessing technique that transforms numerical features by subtracting the mean and dividing by the standard deviation, so that each feature has a mean of 0 and a standard deviation of 1.

For example, to standard scaling `velocity_score`, we can run:

```
WITH scaling_params AS (
    SELECT
        avg_velocity_score: avg(velocity_score),
        stddev_pop_velocity_score: stddev_pop(velocity_score)
    FROM financial_trx_training
)
SELECT
    ss_velocity_score: (velocity_score - avg_velocity_score) /
        stddev_pop_velocity_score
FROM
    financial_trx_testing,
    scaling_params;
```

The above query can be greatly simplified by using DuckDB macros. With scalar macros, we can create a function for the standard scaler transformation:

```
CREATE OR REPLACE MACRO standard_scaler(val, avg_val, std_val) AS
    (val - avg_val) / std_val;
```

With table macros, we can create a function to return the scaling parameters required by the standard scaler macro:

```
CREATE OR REPLACE MACRO scaling_params(table_name, column_list) AS TABLE
    FROM query_table(table_name)
    SELECT
        "avg_\0": avg(columns(column_list)),
        "std_\0": stddev_pop(columns(column_list));
```

In the above macro definition:

- any table can be provided as input parameter and queried due to `query_table`;
- we apply aggregate functions over the list of columns provided as input parameter, by using column expressions;
- we generate the aggregate alias by using the original column name via the `\0` in the alias definition.

We can now calculate the standard scaling as follows:

```
SELECT
    ss_velocity_score: standard_scaler(
        velocity_score,
        avg_velocity_score,
        std_velocity_score
    ),
    ss_spending_deviation_score: standard_scaler(
```

```
        spending_deviation_score,
        avg_spending_deviation_score,
        std_spending_deviation_score
    )
FROM financial_trx_testing,
    scaling_params(
        'financial_trx_training',
        ['velocity_score', 'spending_deviation_score']
    );
```

Min-Max Scaling

Min-Max scaling is a normalization technique that transforms features to a fixed range, typically 0 to 1, by subtracting the minimum value and dividing by the range (`max - min`). This preserves the shape of the original distribution while ensuring all values are on the same scale.

In order to min-max scale our feature, we expand the `scaling_params` macro with the calculation of `min` and `max` over the input column list:

```
CREATE OR REPLACE MACRO scaling_params(table_name, column_list) AS TABLE
    FROM query_table(table_name)
    SELECT
        "avg_\0": avg(columns(column_list)),
        "std_\0": stddev_pop(columns(column_list)),
        "min_\0": min(columns(column_list)),
        "max_\0": max(columns(column_list));
```

We then define a macro definition for the min-max calculation:

```
CREATE OR REPLACE MACRO min_max_scaler(val, min_val, max_val) AS
(val - min_val) / nullif(max_val - min_val, 0);
```

Finally, we extract the values:

```
SELECT
    min_max_velocity_score: min_max_scaler(
        velocity_score,
        min_velocity_score,
        max_velocity_score
    ),
    min_max_spending_deviation_score: min_max_scaler(
        spending_deviation_score,
        min_spending_deviation_score,
        max_spending_deviation_score
    )
FROM financial_trx_testing,
    scaling_params(
        'financial_trx_training',
        ['velocity_score', 'spending_deviation_score']
    );
```

Robust Scaling

Robust scaling is a data normalization technique that transforms numerical features by subtracting the median and dividing by the interquartile range (IQR). Unlike standard scaling, which uses the mean and standard deviation, robust scaling reduces the influence of outliers by focusing on the middle 50% of the data. This makes it well-suited for datasets with skewed distributions or extreme values.

In DuckDB, we can calculate the quantile ranges with the `quantile_cont` statistical aggregate:

```
CREATE OR REPLACE MACRO scaling_params(table_name, column_list) AS TABLE
    FROM query_table(table_name)
    SELECT
```

```
        "avg_\0": avg(columns(column_list)),
        "std_\0": stddev_pop(columns(column_list)),
        "min_\0": min(columns(column_list)),
        "max_\0": max(columns(column_list)),
        "q25_\0": quantile_cont(columns(column_list), 0.25),
        "q50_\0": quantile_cont(columns(column_list), 0.50),
        "q75_\0": quantile_cont(columns(column_list), 0.75);
```

We define the scalar macro for the robust scaling calculation:

```
CREATE OR REPLACE MACRO robust_scaler(val, q25_val, q50_val, q75_val) AS
    (val - q50_val) / nullif(q75_val - q25_val, 0);
```

And, similarly with the other scaling transformations, we call it in SQL directly:

```
SELECT
    rs_velocity_score: robust_scaler(
        velocity_score,
        q25_velocity_score,
        q50_velocity_score,
        q75_velocity_score
    ),
    rs_spending_deviation_score: robust_scaler(
        spending_deviation_score,
        q25_spending_deviation_score,
        q50_spending_deviation_score,
        q75_spending_deviation_score
    )
FROM financial_trx_testing,
    scaling_params(
        'financial_trx_training',
        ['velocity_score', 'spending_deviation_score']
    );
```

Handling Missing Values

It often happens that our input data is incomplete, i.e., it has missing data. Depending on the use case, such data is excluded, used as it is or filled with a constant value. In DuckDB we can use the `coalesce` function in order to retrieve the value of the column or a default value, if the column is `NULL`.

Some common techniques are to:

- replace missing values with a constant;
- replace missing values with the average;
- replace missing values with the median.

We extend the `scaling_params` macro by adding the median calculation:

```
CREATE OR REPLACE MACRO scaling_params(table_name, column_list) AS TABLE
    FROM query_table(table_name)
    SELECT
        "avg_\0": avg(columns(column_list)),
        "std_\0": stddev_pop(columns(column_list)),
        "min_\0": min(columns(column_list)),
        "max_\0": max(columns(column_list)),
        "q25_\0": quantile_cont(columns(column_list), 0.25),
        "q50_\0": quantile_cont(columns(column_list), 0.50),
        "q75_\0": quantile_cont(columns(column_list), 0.75),
        "median_\0": median(columns(column_list));
```

And we apply coalesce to handle the missing values according to our use-case:

```
SELECT
    time_since_last_transaction_with_0: coalesce(time_since_last_transaction, 0),
```

```
    time_since_last_transaction_with_mean: coalesce(time_since_last_transaction, avg_time_since_last_
transaction),
    time_since_last_transaction_with_libraryn: coalesce(time_since_last_transaction, median_time_since_
last_transaction)
FROM
    financial_trx_testing,
    scaling_params('financial_trx_training', ['time_since_last_transaction'])
WHERE time_since_last_transaction IS NULL;
```

Filling missing data should be done before feature scaling.

Benchmark

Bringing the above data processing steps together, we have decided to benchmark the execution time against the `scikit-learn` data preprocessing pipelines. The code is available in our blog examples repository.

In `scikit-learn`, data preprocessing is done through transformers and pipelines. A transformer is a class that implements the `fit` and `transform` methods, while a pipeline is a sequence of transformers that are applied to the data in a specific order. Unless specified otherwise, each step of the pipeline returns, in numpy arrays, only the result of the transformation step. Because in DuckDB we transform the data through SQL expressions, we can inspect the full data set after each step. Therefore, in our benchmark, the `scikit-learn` data preprocessing steps include the following transformations:

- the output of the transformation step is set to `pandas`;
- all the columns are passed through the transformation step, by setting `remainder='passthrough'`.

```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler


def scikit_feature_scaling_training_data(x_train):
    impute_missing_data = Pipeline(
        [
            ("imputer", SimpleImputer(strategy="mean")),
            ("scaler", MinMaxScaler(copy=False)),
        ]
    )

    scaling_steps = ColumnTransformer(
        [
            (
                "ss",
                StandardScaler(copy=False),
                ["velocity_score"]
            ),
            (
                "minmax_time_since_last_transaction",
                impute_missing_data,
                ["time_since_last_transaction"],
            ),
            (
                "minmax",
                MinMaxScaler(copy=False),
                ["spending_deviation_score"]
            ),
            (
                "rs",
                RobustScaler(copy=False),
                ["amount"]
            ),
        ],
        remainder="passthrough",
```

```
        verbose_feature_names_out=False,
    )

    scaling_steps.set_output(transform="pandas")
    scaling_steps.fit(x_train)

    return scaling_steps, scaling_steps.transform(x_train)
```

The figure below shows the execution times on a MacBook Pro with 16 GB, demonstrating that DuckDB offers a significant performance improvement over `scikit-learn` for the data preprocessing steps.

> In the script `reconcile_results.py`, the results between the DuckDB and `scikit-learn` preprocessing steps are reconciled, demonstrating that both implementations produce the same results.

In the above examples, we demonstrated how to perform data preprocessing with SQL expressions during training. In practice, the same preprocessing steps must be applied at inference time, such that new data is transformed consistently with the training data. With `scikit-learn`, this is achieved by persisting the pipeline alongside the model and applying the pipeline at inference time. With DuckDB, the equivalent consistency is achieved by persisting the original training data (or the metrics returned by the `scaling_params` macro, computed during training). Despite the (transformed) training data being much larger than the model artifacts, versioning the data and features calculated at training time is a common practice, which ensures model traceability and reproducibility.

> For efficient (training) data management, one can leverage solutions that provide time travel, such as DuckLake.

Conclusion

In this article we showed how DuckDB offers a performant and SQL-native approach to data preprocessing for machine learning workflows. By handling tasks such as missing value imputation, categorical encoding, and feature scaling directly within the database engine, one can eliminate unnecessary data movement, during training, and reduce preprocessing latency.

Big Data on the Move: DuckDB on the Framework Laptop 13

Publication date: 2025-09-08

Author: Gábor Szárnyas

TL;DR: We put DuckDB through its paces on a 12-core ultrabook with 128 GB RAM, running TPC-H queries up to SF10,000.

Background

When DuckDB's design was first sketched out in 2018, a key insight was that laptops had gotten powerful enough to handle most analytical workloads that data scientists can throw at them. This made implementing a new analytical database as a single-node system that can run everywhere – including laptops – a feasible proposition: enter DuckDB.

In 2020, the release of Apple Silicon gave DuckDB-on-a-laptop setups a big performance boost. Today, modern MacBooks have CPUs with a dozen plus cores along with lots of memory and fast disks – all of which DuckDB is very eager to make use of. However, this comes at a steep price: here in the Netherlands, speccing up a MacBook Pro with 128 GB of RAM and 8 TB of disk will set you back more than 8,500 euros.

Meanwhile, there have been significant leaps in x86 land too. AMD's Ryzen AI 300 ultrathin mobile processors, released in 2024, pack up to 12 cores and 24 threads. DuckDB is just as happy to run on the x86_64 architecture as it is on arm64 processors, so we were eager to find out how DuckDB performs on the new AMDs.

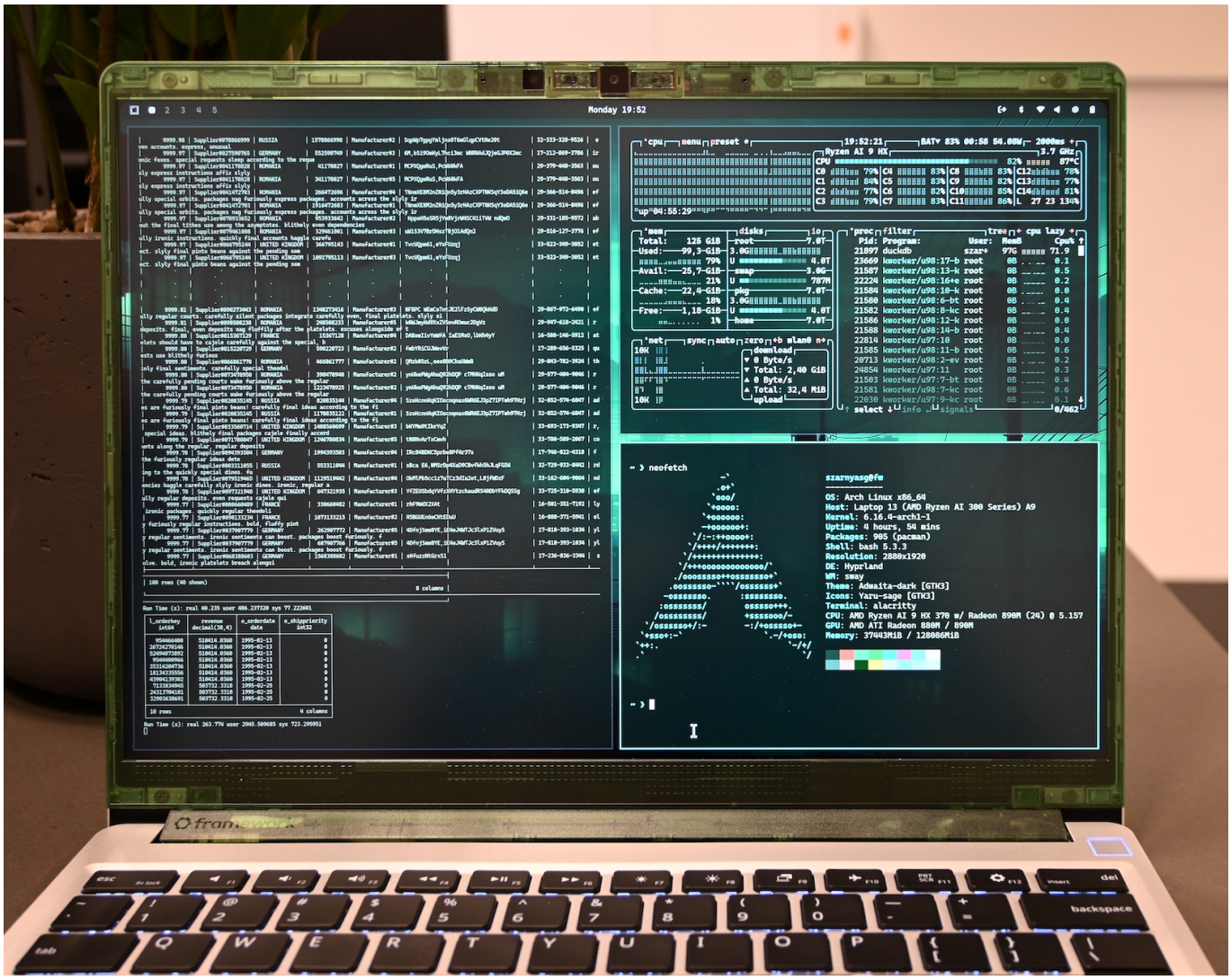
Setup

To run this experiment, we purchased the 13” modular laptop by Framework and kitted it out with 128 GB of RAM and 8 TB of disk – all for just under 3,000 euros.

We installed Omarchy 2.0, an Arch Linux-based distribution with a vibrant community and first-class support for Framework laptops. Assembling the laptop and installing Omarchy took less than an hour in total. Installing the DuckDB command line client only took seconds using the installer script:

```
curl https://install.duckdb.org | bash
```

We changed the theme to *Osaka Jade* for a more Matrix-like look and got to work:



Experiments

To see how this laptop performs, we ran a few benchmarks focusing on loading and query processing times.

CSV Loading

To measure CSV loading performance, we used one of our favorite datasets: the Dutch railway services. We picked the full dataset spanning the last 80 months (between Jan 2019 and Aug 2025). We can fetch and decompress the file as follows:

```
wget https://blobs.duckdb.org/nl-railway/railway-services-80-months.zip
unzip railway-services-80-months.zip
```

The resulting directory is approximately 20 GB. Let's see how quickly DuckDB can load the files:

```
duckdb
```

```
.timer on
CREATE TABLE services AS FROM 'services/services-*.csv';
-- Run Time (s): real 10.219 user 217.596664 sys 7.348692
```

It turns out that DuckDB can load 20 GB of CSV files into memory in just 10.2 seconds, at 1.96 GB/s!

TPC-H Benchmarks

To see query performance, we used TPC-H. We have already run TPC-H queries in all sorts of environments and were curious: how far can DuckDB scale on this laptop?

Data Generation Of course, we first needed some big TPC-H datasets, and we needed to generate them locally – downloading them could take days. Luckily, we could use the `tpchgen-cli` tool, a pure Rust implementation of the TPC-H generator that can produce large-scale datasets on the laptop in just a few hours. We generated the data as Parquet files and loaded them into DuckDB.

SF3,000 We first ran all 22 TPC-H queries on the SF3,000 dataset, which corresponds to 3 TB of CSV files. The total runtime of the queries was 47.5 minutes with a geometric mean query runtime of 86.5 seconds.

During the experiments, we noticed that the laptop bottom cover heated up above 45 degrees Celsius: while the keyboard was still usable, you definitely won't want to keep this machine on your lap while running data crunching workloads. Obviously, the excess heat also results in some thermal throttling that causes the CPU to be downclocked, making the queries slower. We are no stranger to such issues: last year, we dipped an iPhone 16 in dry ice to improve its cooling, but this seemed impractical with the laptop, so we resorted to more practical measures.

For our "cool run", we simply inserted 5-minute sleeping periods between queries to allow the laptop to cool down. This worked very well: the total runtime of the queries (with the cooling periods excluded) went down to 30.8 minutes and the geometric mean runtime reduced to 58.2 seconds – a 32% speedup!

> Most interactive data science workloads have the query execution interspersed with the time spent on coding and analysis – which allows such a cooling period for the laptop.

SF3,000 on Battery So far, we ran our experiments with the laptop plugged into the charger. But laptops are portable devices after all, so we were curious – can we run the queries on the SF3,000 dataset while on the move?

It turns out that we can! Running all TPC-H queries on the SF3,000 dataset took 46.9 minutes (with a geomean runtime of 83.7 seconds) and it drained a fully charged battery to about 30%. That said, if you analyze terabyte-sized datasets on your laptop, you may still want to carry a power bank just to be on the safe side.

TPC-H SF10,000 Finally, it was time for our ultimate challenge – can this laptop handle the SF10,000 dataset, corresponding to 10 TB of CSV files? To find out, we:

- generated the data – *about 4 TB in Parquet files,*
- loaded the files into DuckDB – *about 2.7 TB in DuckDB's file format,*
- cleaned up the Parquet files – *knowing that the extra space will come in handy later,*
- ran the queries – *occasionally witnessing DuckDB spilling 3.6 TB on disk,*

and ultimately saw them finishing without an issue!

The run took a total of 4.2 hours with a geometric mean query runtime of 6.6 minutes.

Unsurprisingly, some thermal throttling also occurred here, so we repeated the experiment with cooling periods. We found that this brings down the total query runtime to 3.8 hours with a geomean runtime of 5.7 minutes (14% speedup). This means that the differences are smaller than for the SF3,000 dataset, which makes perfect sense: given the longer query runtimes, the cooling periods no longer have that much of an effect.

Conclusion

In short, we found that you can build a laptop for less than 3,000 euros that can load CSV files at almost 2 GB / second, run the full range of TPC-H queries on SF3,000 while on battery, and complete all queries on the SF10,000 dataset.

While ours is not an official TPC-H run, it's worth mentioning that among the current ranking of audited SF10,000 TPC-H implementations, the cost of the cheapest setup capable of doing TPC-H SF10,000 is well over 1 million euros or 1.2 million U.S. dollars – our setup costs less than 0.3% of that and will do the job for 99% of data science workloads.

Just don't forget to pause every now and then to allow the laptop to cool down a bit.

Appendix

Cost Breakdown

Here's the cost breakdown of the laptop using retail prices in euros (including the 21% Dutch VAT). The items were purchased in August 2025.

Item	Cost (EUR)
Framework Laptop 13 DIY Edition w/ AMD Ryzen AI 9 HX 370 CPU	1,785.00
Framework Laptop 13 Bezel – Translucent Green	55.00
Keyboard – US English	109.00
Power Adapter – 60W – EU	49.00
HDMI (3rd Gen) Expansion Card	20.00
USB-A Expansion Card	10.00
USB-C Expansion Card – Translucent Green (2 pcs)	20.00
WD_BLACK SN850X NVMe SSD 8TB	582.69
Crucial DDR5 RAM 128GB Kit	327.87
Total (euros, VAT included)	**2,974.87**

Note that Framework currently only sells 96 GB memory kits but both the motherboard and the CPU can handle 128 GB (2 × 64 GB) kits.

Solving Letter Scramble Puzzles with DuckDB

Publication date: 2025-09-11

Author: Gábor Szárnyas

TL;DR: In this lighthearted post, we solve a puzzle type that's on display in Dutch trains.

The Dutch National Railways (NS) is releasing a "letter scramble"-style puzzle every week where they give a term whose letters can be found in a Dutch train station's name. In order to have a match, it doesn't have to be a perfect anagram – for example, `Amsterdam` (9 letters) matches both `mastered` (8 letters) and `Dream Master` (11 letters) because all three terms contain the same letters, just with different amounts of repetitions. Let's call this a "weak anagram".

The puzzle in the first week of September was `Clumsy Rental Red`. Let's try to find the solution using DuckDB!

Letter Scrambling Macro

First, let's create a macro that turns a string into an ordered list of unique characters:

```
CREATE MACRO order_letters(s) AS
    lower(s)                    -- convert all characters to lowercase
    .regexp_replace(            -- remove all non-Unicode letters
        '[^\p{L}]', '', 'g'
    )
    .string_to_array('')        -- turn the string into a list
    .list_distinct()            -- eliminate duplicate elements from the list
    .list_sort();               -- sort the list
```

We can use this to see whether two terms are weak anagrams:

```
SELECT
    order_letters('Amsterdam') AS letters_1,
    order_letters('mastered') AS letters_2,
    order_letters('Dream Master') AS letters_3,
    letters_1 = letters_2 AS matches_1,
    letters_1 = letters_3 AS matches_2;
```

letters_1	letters_2	letters_3	matches_1	matches_2
[a, d, e, m, r, s, t]	[a, d, e, m, r, s, t]	[a, d, e, m, r, s, t]	true	true

Indeed, both expressions are weak anagrams of `Amsterdam`!

Matching against Station Names

To solve the puzzle, we need a list of train stations. Luckily, one of our go-to datasets at DuckDB is the Dutch railway datasets, including its services and train stations. We can create a table with the station names:

```
CREATE TABLE stations AS
    FROM 'https://blobs.duckdb.org/nl-railway/stations-2023-09.csv';
```

Then, we can select the station names that are weak anagrams of the puzzle:

```
SELECT name_long
FROM stations
WHERE order_letters(name_long) = order_letters('Clumsy Rental Red');
```

We are not going to spoil the solution but you can reveal it below.

Table Macro for Finding Weak Anagrams

To find station name, which is a weak anagram to a term, we can use a table macro:

```
CREATE MACRO find_weak_anagram(s) AS TABLE
    SELECT name_long
    FROM stations
    WHERE order_letters(name_long) = order_letters(s);
```

Then, we can find the solution using a simple SQL statement:

```
FROM find_weak_anagram('Clumsy Rental Red');
```

Weak Anagram Station Pairs

We got curious: are there two stations whose names are weak anagrams of each other? We can create a Cartesian product from the station names and compare their ordered letters to find out:

```
SELECT s1.name_long AS station_1, s2.name_long AS station_2
FROM stations s1, stations s2
WHERE s1.name_long.order_letters() = s2.name_long.order_letters()
  -- ensure symmetry-breaking
  AND s1.name_long < s2.name_long
  -- make sure the station names don't contain each other
  AND NOT s1.name_long.contains(s2.name_long)
  AND NOT s2.name_long.contains(s1.name_long);
```

There are in fact three station pairs whose names are weak anagrams of each other:

station_1	station_2
Melsele	Selm
Etten-Leur	Lunteren
Diemen Zuid	Emmen Zuid

Cleaning Up

Most of the time, you don't have to clean up after running a simple DuckDB script: simply closing the in-memory database session takes care of the cleanup. However, it's worth pointing out that macros in DuckDB are persisted and this can get in the way – e.g., when copying the database into a DuckLake:

```
ATTACH 'ducklake:metadata.ducklake' AS my_ducklake;
COPY FROM DATABASE memory TO my_ducklake;
```

DuckLake does not support macros (functions), so it throws the following error:

```
Not implemented Error:
DuckLake does not support functions
```

There are two options to work around this problem.

- If you need to keep the macros and you are using DuckDB as your catalog database for DuckLake, you can use the DuckDB to DuckLake migration script. This will migrate the macros into the catalog of your DuckLake.

- If you do not need the macros or your destination's catalog database does not support them, you can drop them with the following commands:

  ```
  DROP MACRO order_letters;
  DROP MACRO TABLE find_weak_anagram;
  ```

  Without the macros, the copying to DuckLake call succeeds.

Summary

That was our quick guide to solving the NS puzzle. Is this a database problem? Not really, but DuckDB's SQL allows you to succinctly formulate it and solve it in less than 0.1 seconds! And sure, ChatGPT can solve this puzzle – but it spends *almost a minute* (using a ton of computing resources) to reason its way through it.

Happy puzzle solving!

> This week, the puzzle is `Zere Tanda Voozan`. You can find the weekly puzzle's solution on the NS website.

Update on Weak vs. Strong Anagrams

Reader feedback indicated that the NS puzzles are *strong anagrams* (with the same amount of letters as the station name) and the puzzle term for the first week of September was actually `Clumsy Rental Ted` and not `Clumsy Rental Red`! The latter is impossible to fact-check but we looked into historical data, ran it through our DuckDB solver script, and it turned out that 95% of the time there are *strong anagrams.* However, occasionally weak anagrams are also given such as `Alleen Costume Hut` which stands for `Houten Castellum` even though the puzzle term and the station name have different numbers of `e` letters.

Announcing DuckDB 1.4.0 LTS

Publication date: 2025-09-16

Author: The DuckDB team

TL;DR: We're releasing DuckDB version 1.4.0, codenamed “Andium”. This is an LTS release with one year of community support, and it packs several new features including database encryption, the MERGE statement and Iceberg writes.

We are proud to release DuckDB v1.4.0, named “Andium” after the *Andean teal* (Anas andium), which lives in the Andean highlands of Colombia, Venezuela and Ecuador.

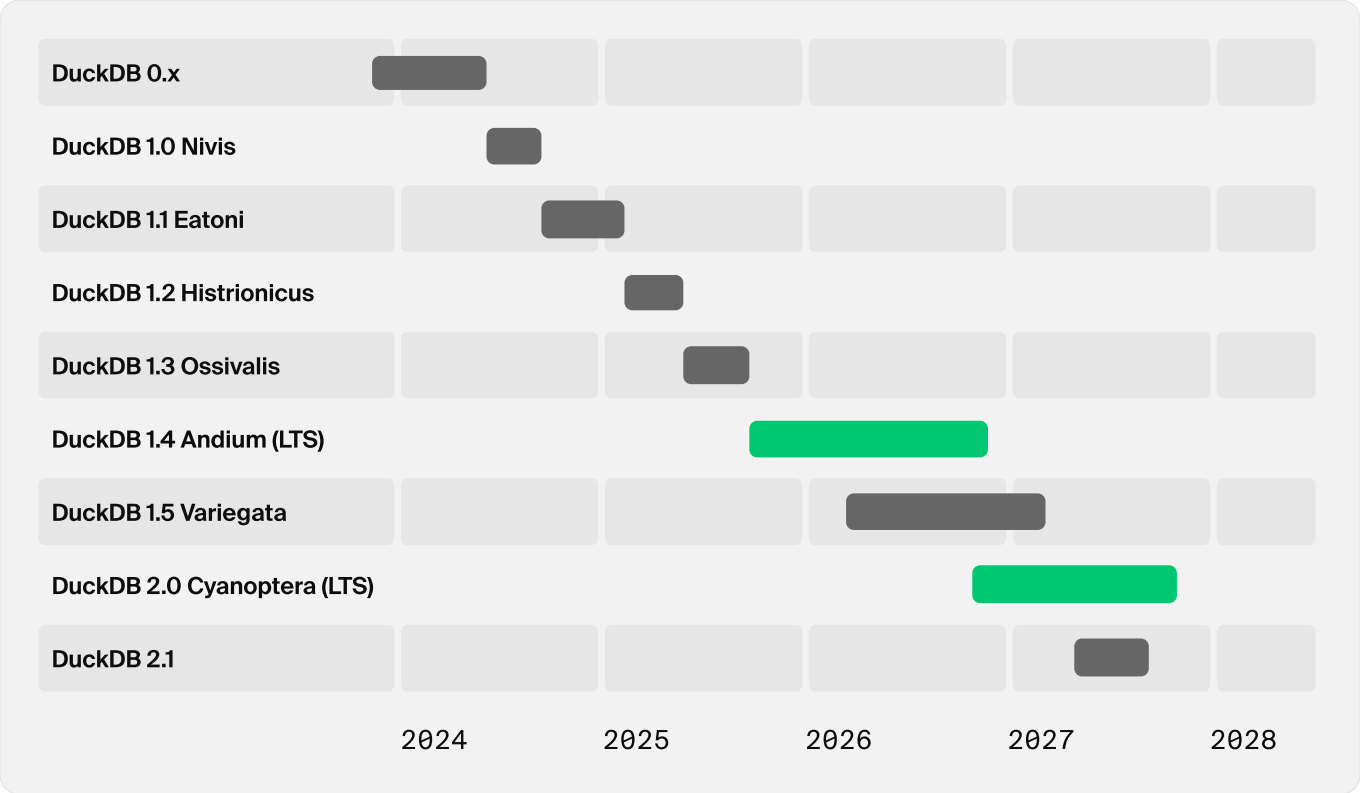
In this blog post, we cover the most important updates for this release around support, features and extensions. DuckDB is moving rather quickly, and we could cover only a small fraction of the changes in this release. For the complete release notes, see the release page on GitHub.

> To install the new version, please visit the installation page due to the extra changes and review rounds required.

Long-Term Support (LTS) Edition

We are delighted to see that DuckDB is used regularly in production environments and realize that such deployments often come with a requirement for long-term maintenance. In the past, we would automatically deprecate old DuckDB versions whenever the newer version was released. But we're changing this today.

Starting with this release, every *other* DuckDB version is going to be a Long-Term Support (LTS) edition. For LTS DuckDB versions, community support will last a year after the release (for now).



{: .lightmode-img }

New Features

Database Encryption

Being able to encrypt DuckDB database files has been a long-standing feature request. Starting with this release, DuckDB supports encryption of its files. Encryption keys are given using the `ENCRYPTION_KEY` parameter to `ATTACH`, like so:

```
ATTACH 'encrypted.db' AS enc_db (ENCRYPTION_KEY 'quack_quack');
```

DuckDB uses the industry-standard AES encryption with a key length of 256 bits using the recommended GCM mode by default.

The encryption covers the main database file, the write-ahead-log (WAL) file, and even temporary files. To encrypt data, DuckDB can use either the built-in `mbedtls` library or the OpenSSL library from the `httpfs` extension. Note that the OpenSSL versions are much faster due to hardware acceleration, so make sure to `LOAD httpfs` for good encryption performance.

Encryption support in DuckDB was implemented by Lotte Felius (@ccfelius).

MERGE Statement

DuckDB now supports `MERGE INTO` as an alternative to `INSERT INTO ... ON CONFLICT`. `MERGE INTO` does not require a primary key since it works on any custom merge condition. This is a very common statement in OLAP systems that do not support primary keys but still want to support upserting (i.e., `UPDATE` plus `INSERT`) functionality.

In this example we use a simple condition matching on a key and we call the `RETURNING` statement to get a summary of the updated and inserted rows.

```
CREATE TABLE Stock(item_id INTEGER, balance INTEGER);
INSERT INTO Stock VALUES (10, 2200), (20, 1900);

WITH new_stocks(item_id, volume) AS (VALUES (20, 2200), (30, 1900))
    MERGE INTO Stock
        USING new_stocks USING (item_id)
    WHEN MATCHED
        THEN UPDATE SET balance = balance + volume
    WHEN NOT MATCHED
        THEN INSERT VALUES (new_stocks.item_id, new_stocks.volume)
    RETURNING merge_action, *;
```

merge_action varchar	item_id int32	balance int32
UPDATE	20	4100
INSERT	30	1900

`MERGE INTO` also supports more complex conditions and `DELETE` statements.

```
WITH deletes(item_id, delete_threshold) AS (VALUES (10, 3000))
    MERGE INTO Stock
        USING deletes USING (item_id)
    WHEN MATCHED AND balance < delete_threshold
        THEN DELETE
    RETURNING merge_action, *;
```

merge_action varchar	item_id int32	balance int32
DELETE	10	2200

Iceberg Writes

The duckdb-iceberg extension now **supports writing to Iceberg**.

This means that copying data from DuckDB or DuckLake to Iceberg is now possible.

- **Copying from Iceberg to DuckDB/DuckLake** is also supported via the COPY statement:

```
COPY FROM DATABASE iceberg_datalake TO duckdb_db;
```

- **Copying from DuckLake/DuckDB to Iceberg** needs manual creation of the schemas on the Iceberg side of things:

```
CREATE SCHEMA iceberg_datalake.main;
COPY FROM DATABASE duckdb_db TO iceberg_datalake;
```

CLI Progress Bar ETA

Community member Rusty Conover (@rustyconover) contributed an ETA (estimated time of arrival) feature to the DuckDB command line client. Estimating the remaining time is a difficult problem as progress measurements can vary a lot due to noise. To alleviate this, the ETA feature first collects some initial performance data, then continues to refine its estimate using a Kalman filter. Here's how it works in practice:

`FILL` Window Function

Richard (@hawkfish) built a new window function, `FILL`, that can be used to *interpolate* missing values in ordered windows. Here is an example, you can see a missing value between 1 and 42, it's interpolated to 21 in the result.

```
FROM (VALUES (1, 1), (2, NULL), (3, 42)) t(c1, c2)
SELECT fill(c2) OVER (ORDER BY c1) f;
```

This will be the result:

f
1
21
42

Teradata Connector

DuckDB now has a Teradata Connector. A separate blog post will be coming.

Performance and Optimizations

DuckDB v1.4.0 received a number of performance optimizations.

Checkpointing In-Memory Tables

In-memory tables now support checkpointing. This has two key benefits:

- In-memory tables now support compression, which can have a significant (5-10×) performance boost for some queries. This is disabled by default – you can turn it on using:

```
ATTACH ':memory:' AS memory_compressed (COMPRESS);
USE memory_compressed;
```

- Checkpointing triggers vacuuming deleted rows, allowing space to be reclaimed after deletes/truncation.

Sorting Rework

Laurens (@lnkuiper) rewrote DuckDB's sorting implementation (again). This new implementation uses a k-way merge sort to reduce data movement. It is also adaptive to pre-sorted data and uses a new API that makes it possible to use this new sorting code elsewhere in DuckDB, for example in window functions. We are seeing much better thread scaling performance with this implementation. We will publish a separate blog post with more detailed performance measurements.

> **Update.** We have now covered this in a separate blog post.

Materializing Common Table Expressions

Common table expressions (CTEs) are now materialized by default (instead of inlining them). This both improves performance and resolves some correctness bugs that happened due to inlining. This feature was implemented by Denis Hirn (kryonix), who contributed support for recursive CTEs back in 2020.

Additional Optimizations

DuckBB v1.4.0 fixes some scaling issues, supports caching for hashes of string dictionaries and has several aggregation optimizations.

Distribution

macOS Notarization

macOS has a fairly advanced model to ensure system integrity built around cryptographic signatures along with so-called "notarization" by Apple. We had been signing our binaries for about two years already. Starting from this release, the DuckDB command line utility (`duckdb`) and the dynamic library `libduckdb...sdylib` are *released with this notarization*. This will reduce the amount of complaints when using web browsers to download our binaries. Unfortunately, macOS does not yet fully support notarization of command line utility, so the "open with double click" use case will still have to wait. The recommended path to install the CLI on macOS is still our install script:

```
curl https://install.duckdb.org | bash
```

Moved Python Integration to its Own Repository

We have been slowly moving language integrations ("clients") into their own repositories from `duckdb/duckdb`. For this release, we moved the Python client to its own repository, `duckdb/duckdb-python`. Please make sure to file issues related to the Python client there.

Final Thoughts

These were a few highlights – but there are many more features and improvements in this release. There have been over 3,500 commits by over 90 contributors since we released v1.3.2. The full release notes can be found on GitHub. We would like to thank our community for providing detailed issue reports and feedback. And our special thanks goes to external contributors, who directly landed features in this release!

DuckLake 0.3 with Iceberg Interoperability and Geometry Support

Publication date: 2025-09-17

Authors: Guillermo Sanchez, Gábor Szárnyas

TL;DR: We are releasing version 0.3 of the DuckLake Specification and the `ducklake` DuckDB extension.

Redesigning DuckDB's Sort, Again

Publication date: 2025-09-24

Author: Laurens Kuiper

TL;DR: After four years, we've decided to redesign DuckDB's sort implementation, again. In this post, we present and evaluate the new design.

DuckDB v1.4.0 was just released, which includes a complete redesign of DuckDB's sort implementation. We redesigned DuckDB's sort just four years ago, which allowed DuckDB to sort more data than fits in main memory, in parallel, with highly efficient comparisons. This implementation served us well, but since then we've implemented larger-than-memory query processing for more operators, such as the hash join and hash aggregation, which both use a new and improved spillable page layout. We presented this layout in an earlier blog post. We decided to integrate this layout in DuckDB's sort, and completely redesigned the implementation.

Not interested in the implementation? Jump straight to the benchmark!

Two-Phase Sorting

DuckDB implements parallel query execution using Morsel-Driven Parallelism. In DuckDB's implementation of this framework, blocking operators, i.e., *operators that must read the entire input before they can output*, such as the hash aggregation and sort operators, have the following phases:

1. **Sink**: Thread-local accumulation of data from a pipeline
2. **Combine**: Signals a thread finishing its *Sink* phase
3. **Finalize**: Called once when all threads have called *Combine*
4. **GetData**: Output data to the next pipeline

For many decades, the preferred option to implement larger-than-memory sorting in database systems has been to generate multiple sorted runs, followed by a merge sort. Specifically, a *k*-way merge sort produces the lowest amount of I/O during larger-than-memory sorting, which is critical to performance. This approach maps well to Morsel-Driven Parallelism: DuckDB performs thread-local sorting in the *Sink* phase, followed by a parallel merge sort in the *Finalize* or *GetData* phase. Both have been redesigned for DuckDB v1.4.0. We first discuss the new thread-local sort implementation before presenting the new merge design.

Thread-Local Sorting

Sorted runs are generated thread-locally in the *Sink* phase. The way DuckDB parallelizes this has not changed in v1.4.0: threads generate sorted runs independently, in parallel. What has changed is the physical sorting implementation.

Key Normalization

Database systems that do not compile the required types into the query plan – e.g., DuckDB – suffer from interpretation overhead, especially when comparing tuples while sorting. One way to get around this is Key Normalization. DuckDB's sort already used an ad-hoc version of this prior to v1.4.0, but the new implementation uses the more generic `create_sort_key` function that is available through SQL.

This function takes any number of inputs and sort conditions, and constructs a BLOB field that produces the specified order. An example from the description of the PR that implemented `create_sort_key`:

```
SELECT
    s,
    create_sort_key(s, 'asc nulls last') AS k1,
    create_sort_key(s, 'asc nulls first') AS k2
FROM
    (VALUES ('hello'), ('world'), (NULL)) t(s);
```

s varchar	k1 blob	k2 blob
hello	\x01ifmmp\x00	\x02ifmmp\x00
world	\x01xpsme\x00	\x02xpsme\x00
NULL	\x02	\x01

Because of the binary-comparable nature of the constructed BLOB, the following queries are equivalent:

```
SELECT * FROM tbl
ORDER BY x DESC NULLS LAST, y ASC NULLS FIRST;
```

```
SELECT * FROM tbl
ORDER BY create_sort_key(x, 'DESC NULLS LAST', y, 'ASC NULLS FIRST');
```

This fixes the problem of interpretation overhead when comparing tuples, as we now only have to consider comparing BLOBs, instead of arbitrary combinations of types in an ORDER BY clause.

Static Integer Comparisons

It's well known that processing strings is a lot slower than processing fixed-size types such as integers. If we would always use the `create_sort_key` function, even for integers, we'd be leaving a lot of performance on the table. However, if we know the size of the resulting BLOB, we can convert it back to one or more unsigned integers, and use integer comparisons instead.

For example, if we have the following query:

```
SELECT *
FROM tbl
ORDER BY
    c0::INTEGER ASC NULLS LAST,
    c1::DOUBLE ASC NULLS LAST;
```

The resulting BLOB from `create_sort_key(c0::INTEGER, 'ASC NULLS LAST', c1::DOUBLE, 'ASC NULLS LAST')` is less than 16 bytes, so the new sorting implementation will swap the bytes (for big-endian integer comparisons) and store them in two 64-bit unsigned integers. A simplified version of the data structure we use in C++:

```
struct FixedSortKeyNoPayload {
    uint64_t part0;
    uint64_t part1;
};
struct FixedSortKeyPayload {
    uint64_t part0;
    uint64_t part1;
    data_ptr_t payload;
};
```

Which can be compared like so:

```
bool LessThan(const FixedSortKeyPayload &lhs, const FixedSortKeyPayload &rhs) {
    return lhs.part0 < rhs.part0 || (lhs.part0 == rhs.part0 && lhs.part1 < rhs.part1);
}
```

The `payload` field is only present if more columns are selected, i.e.:

```
SELECT <many columns>
FROM tbl
ORDER BY <a few columns>;
```

If only columns are selected that also occur in the ORDER BY clause, the `payload` field is not needed, as DuckDB can decode the normalized keys.

Non-Contiguous Iteration

Prior to v1.4.0, DuckDB used fixed-size sort keys, but their size was only known when executing the query. This necessitates comparing and moving sort keys dynamically while sorting, which is much less efficient than statically compiled code. The C++ `struct` that DuckDB uses now, shown above, is known at compile time, which allows it to be sorted with sorting algorithms that implement the C++ `std::iterator` interface. This means that DuckDB no longer needs to implement a sorting algorithm: it can grab an off-the-shelf C++ implementation!

C++ comes with `std::iterator` implementations for various data structures such as `std::array` and `std::vector`. These data structures, however, require storing all data in a contiguous block of memory. DuckDB uses a default page allocation (= contiguous block of memory) size of 256 KiB. The `FixedSortKeyPayload` shown above is 24 bytes, so only ~10k tuples fit in a page. We want sorted runs to be much longer than that (for performance reasons that we will not get into in this blog post). To be able to generate longer sorted runs, we implemented an `std::iterator` that can iterate over non-contiguous blocks of memory:

While this iterator is great at sequential access, some sorting algorithms require random access. With this design, we cannot simply add an offset to a pointer to get the address of a tuple. Instead, we compute the page index and offset within the page using integer division/modulo, as the number of tuples per page is always the same (except for the last page). However, integer division/modulo is not cheap compared to the simple pointer arithmetic that can be used for contiguous blocks of memory, so we use `fastmod` to reduce the cost.

Sorting Algorithm

With the components described so far, we are able to generate large sorted runs, that can be spilled to storage page-by-page, rather than in an all-or-nothing fashion. We use a combination of three sorting algorithms to achieve good sorting performance and high adaptivity to pre-sorted data:

1. Vergesort
2. Ska Sort
3. Pattern-defeating quicksort

Vergesort detects and merges runs of *(almost) sorted* data, which greatly reduces the effort it takes to process, *e.g.*, time-series data, which is often stored in sorted order already. If Vergesort cannot detect any patterns, it falls back to Ska Sort, which performs an adaptive Most Significant Digit (MSD) radix sort on the first 64-bit integer of the sort key. If radix partitions in the recursion become too small, or if the data is not fully sorted after the first 64-bit integer, it falls back to Pattern-defeating quicksort.

Merging

Prior to v1.4.0, DuckDB would materialize the fully-merged data. However, with a *k*-way merge, it is possible to output chunks of sorted data directly from the sorted runs, in somewhat of a streaming fashion. This means that data can be output before the full merge has been computed. We visualize this for four sorted runs:

Chunk 1 can be output to the next pipeline before all sorted runs have been merged. One of the reasons that this is useful is large ORDER BY `... LIMIT ...` queries. If the `LIMIT` is small, DuckDB uses a min-heap, which is much faster than sorting the entire input. However, for large `LIMIT`s, the min-heap approach becomes worse than fully sorting and then applying the `LIMIT`. With a *k*-way merge, the merge can be stopped by a `LIMIT` at any point, meaning that the cost of fully merging the sorted runs is never incurred.

Traditionally, the *k*-way merge is evaluated sequentially using a tournament tree. However, with modern multi-core CPUs, this leaves a lot of performance on the table. The question is, how can we do this in parallel?

K-Way Merge Path

Various algorithms to parallelize merge sort exist, such as Merge Path, which DuckDB's sort used prior to v1.4.0, and Bitonic Merge Sort. However, these algorithms parallelize a *cascading two-way merge sort*, not a *k*-way merge sort. So, while these algorithms are parallel and skew-resistant, they are unattractive for larger-than-memory sorting, as they produce much more I/O.

For *k*-way merging, fewer options for parallelization exist. The work can be divided using value-based splitting. However, it is easy to see that parallelism breaks down when the input distribution is extremely skewed, e.g., when half of the input has the same value, as there is no splitting value that can divide the work into evenly-sized tasks. After searching the web, the only skew-resistant parallel *k*-way merge that we could find is a bachelor thesis from 2014. We wanted a very fine-grained approach, so instead, we *generalized Merge Path to k sorted runs*.

In the previous figure, there is a horizontal line in each sorted run that indicates how much of each sorted run went into the output chunk. The general idea of Merge Path, as explained in our blog post on sorting four years ago, is to compute *where these lines are*, i.e., where the sorted runs intersect. Merge Path does this efficiently for merging two sorted runs using *binary search*.

We generalize this approach to *k* sorted runs, which allows us to choose an arbitrary output chunk size, and compute where the sorted runs intersect such that when they are merged, the resulting chunk will be of the chosen size. This allows for very fine-grained skew-resistant parallelism, which is not possible when choosing specific *splitting values,* as the size of the chunks that this produces depends on the data distribution. This is the pseudo-code for *k*-way merge path:

```
def compute_intersections(sorted_runs, chunk_size):
    intersections = [0 for _ in range(len(sorted_runs))]
    while chunk_size != 0:
        delta = ceil(chunk_size / len(sorted_runs))
        min_idx = 0
        min_val = sorted_runs[0][intersections[0] + delta]
        for run_idx in range(1, len(sorted_runs)):
            val = sorted_runs[run_idx][intersections[run_idx] + delta]
            if val < min_val:
                min_idx = run_idx
                min_val = val
        intersections[min_idx] += delta
        chunk_size -= delta
    return intersections
```

This has been greatly simplified, as this does not take into account any edge cases or going out-of-bounds on the sorted runs. The general idea is that we move up the lower bound for the intersection of one sorted run in each iteration of the `while` loop. This has a worse complexity than the binary search used in the original Merge Path, but it is also has to be called fewer times because a *k*-way merge can merge all sorted runs in a single pass, rather than in many passes. Profiling shows that this computation takes up just 1-2% of the overall execution time.

Threads can compute the intersections independently, and, therefore, in parallel. Once threads have computed the intersections, they are free to merge the data between the intersections, as the data is guaranteed not to overlap with that of other threads. The merged chunks can immediately be output in parallel due to DuckDB's *order-preserving parallelism*.

Benchmark

So, how does the new sorting implementation perform compared to the old one? We run a few experiments on my laptop (M1 Max MacBook Pro with 10 threads and 64 GB of memory).

Raw Performance

We first benchmark raw integer sorting performance. We have three types of inputs (pre-sorted ascending, pre-sorted descending, and randomly ordered), at three different sizes (10, 100, and 1000 million rows). We've generated the data using the following queries:

```
CREATE TABLE ascending10m AS
    SELECT range AS i FROM range(10_000_000);
```

```
CREATE TABLE descending10m AS
    SELECT range AS i FROM range(9_999_999, 0, -1);

CREATE TABLE random10m AS
    SELECT range AS i FROM range(10_000_000) ORDER BY random();

-- and so on for 100m and 1000m
```

We took the median of 5 runs of each of these queries, for each table size:

```
SELECT any_value(i)
FROM (FROM ascending10m ORDER BY i);

SELECT any_value(i)
FROM (FROM descending10m ORDER BY i);

SELECT any_value(i)
FROM (FROM random10m ORDER BY i);

-- and so on for 100m and 1000m
```

This query causes DuckDB to evaluate the entire sort, without materializing the whole table as a query result. This allows us to better isolate the performance of the sorting implementation.

Results

Table	Rows [Millions]	Old [s]	New [s]	Speedup vs. Old [x]
Ascending	10	0.110	**0.033**	3.333
Ascending	100	0.912	**0.181**	5.038
Ascending	1000	15.302	**1.475**	10.374
Descending	10	0.121	**0.034**	3.558
Descending	100	0.908	**0.207**	4.386
Descending	1000	15.789	**1.712**	9.222
Random	10	0.120	**0.094**	1.276
Random	100	1.028	**0.587**	1.751
Random	1000	17.554	**6.493**	2.703

This shows that the new implementation is highly adaptive to pre-sorted data: it is roughly 10x faster at ascending/descending data than the old implementation. It has much better raw sorting performance: it is more than 2× faster at sorting randomly ordered data (at 1000 million).

We also plot the results on a **log-log scale**:

Here, we can see that the new implementation scales much better: the execution time of the new implementation increases less steeply with input size than the old implementation.

Wide Table

The first benchmark evaluated raw sorting performance. In this next benchmark, we sort a *wide table*, i.e., we select many columns to be sorted by the ORDER BY clause. We sort the `lineitem` table from TPC-H which has 15 columns, by the `l_shipdate` column, at scale factors 1 (~6 million rows), 10 (~60 million rows) and 100 (~600 million rows), generated using DuckDB's TPC-H extension.

We took the median execution time of 5 runs of this query for each scale factor:

```
SELECT any_value(COLUMNS(*))
FROM (FROM lineitem ORDER BY l_shipdate);
```

Results

Table	SF	Old [s]	New [s]	Speedup vs. Old [x]
TPC-H SF 1 `lineitem` by `l_shipdate`	1	0.328	**0.189**	1.735
TPC-H SF 10 `lineitem` by `l_shipdate`	10	3.353	**1.520**	2.205
TPC-H SF 100 `lineitem` by `l_shipdate`	100	273.982	**80.919**	3.385

We have set the memory limit to 30 GB, so the data no longer fits in memory at scale factor 100. The new implementation is roughly 2× faster at scale factors 1 and 10, and more than 3× faster at scale factor 100. This shows that the new *k*-way merge sort reduces data movement and I/O, is much more efficient at sorting wide tables than the old cascaded 2-way merge sort.

Again, we plot the results on a **log-log scale**:

And we can see that the new implementation scales much better, especially when the data no longer fits in main memory.

Thread Scaling

Finally, we benchmark how well the sorting implementation scales with threads. We sort the table with 100 million randomly ordered integers from before, with 1, 2, 4, and 8 threads. We use the same data and query as in the first benchmark, and take the median of five runs.

Results

Threads	Old [s]	New [s]	Old Speedup vs. 1 Thread [x]	New Speedup vs. 1 Thread [x]
1	**3.240**	4.234	**1.000**	**1.000**
2	**2.121**	2.193	1.527	**1.930**
4	1.401	**1.216**	2.312	**3.481**
8	0.920	**0.654**	3.521	**6.474**

As we can see, the new, single-threaded sorting performance is around 30% slower than the old one. This is due to the new sorting implementation using an *in-place MSD radix sort*, rather than an *out-of-place Least Significant Digit (LSD) radix sort*. This makes the old implementation perform better specifically on this workload, at the cost of using much more memory.

However, if we increase the number of threads to 2, this advantage is already gone. At 8 threads, the old implementation has a speedup of only ~3.5× over 1 thread, while this speedup is ~6.5× for the new implementation.

Again, we plot the results on a **log-log scale**:

This shows that the new implementation's parallel scaling is much better than the old implementation.

Conclusion

DuckDB's new sorting implementation has greatly improved performance over the old sorting implementation. It is highly adaptive to pre-sorted data, performs less I/O when sorting data that does not fit in main memory, and scales much better with additional threads.

If you've upgraded to v1.4.0, you can enjoy the improved performance when using the `ORDER BY` clause. The new sorting implementation has already been integrated into the window operator, so we expect to see a performance improvements when using the `OVER` clause as well. For v1.5.0, we aim to integrate the new sorting implementation into the joins that use sorting such as the `ASOF` join.

Announcing DuckDB 1.4.1 LTS

Publication date: 2025-10-07

Author: The DuckDB team

TL;DR: Today we are releasing DuckDB 1.4.1, the first patch release of our LTS edition.

In this blog post, we highlight a few important fixes and convenience improvements in DuckDB v1.4.1, the first patch release in DuckDB's 1.4 LTS line. You can find the complete release notes on GitHub.

To install the new version, please visit the installation page.

Iceberg Improvements

The DuckDB `iceberg` extension received a number of patches:

- You can now attach to an Iceberg REST Catalog and specify an access delegation mode. This fixes a bug when using catalogs that did not vend credentials. The `ATTACH` statement will now look like this:

  ```
  ATTACH '<warehouse_name>' AS my_datalake (
      TYPE iceberg,
      ENDPOINT '<endpoint>',
      ACCESS_DELEGATION_MODE '<delegation_mode_option>',
      SECRET '<my_secret>'
  );
  ```

  The current `ACCESS_DELEGATION_MODE` options are `vended_credentials` (default) and `none`.

- When attaching to AWS-managed REST Catalogs, the `http_timeout` setting is now respected.
- Attempting to rename or replace a table within a transaction now throws a clear error message.
- AWS Athena can now read Iceberg tables written by DuckDB.

AWS Improvements

The `aws` extension received a number of changes, which makes it easier to configure and troubleshoot. See the `aws` documentation page for more details.

Secret Validation

Since DuckDB v1.4.0, the AWS `credential_chain` provider looks for any required credentials during `CREATE SECRET` time, failing if absent/unavailable. Since v1.4.1 this behavior can also be configured via the `VALIDATION` option as follows:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    VALIDATION 'exists'
);
```

Two validation modes are supported:

- `exists` (default) requires present credentials.
- `none` allows `CREATE SECRET` to succeed for `credential_chains` with no available credentials.

S3 Default Region

Previously, setting the S3 region incorrectly could result in difficult-to-debug situations (`Unknown error for HTTP HEAD to ...`).

DuckDB v1.4.1 removes `us-east-1` as the default S3 region and returns a 301 error code if an incorrect region is used.

Fixes for Missing Data

Users reported two cases where DuckDB omitted some data:

- The Parquet reader had a regression which caused it to omit some rows when using predicate pushdown on certain string columns.
- In certain edge cases, DuckDB's ART index could omit rows rows non-deterministically when running on multiple threads. Note that this index is only used when you manually specify an index with `CREATE INDEX`.

DuckDB v1.4.1 fixes both of these issues.

Autoloading

In DuckDB v1.4.0, the `httpfs` extension was not always autoloaded. For example, running:

```
COPY (SELECT 42 AS answer) TO 's3://my_bucket/my_file.parquet';
```

without loading `httpfs` manually returned the following error:

```
Cannot open file "s3://my_bucket/my_file.parquet": No such file or directory
```

With v1.4.1, autoloading works and DuckDB can write to the bucket right away.

Docker Image

We now officially distribute a Docker image, making it easy to run DuckDB in a containerized environment:

```
docker run --rm -it -v "$(pwd):/workspace" -w /workspace duckdb/duckdb
```

```
DuckDB v1.4.1 (Andium) b390a7c376
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
D
```

For more details including operational considerations and using the UI, read the Docker image page.

Adoption Metrics and Benchmark Results for DuckDB v1.4 LTS

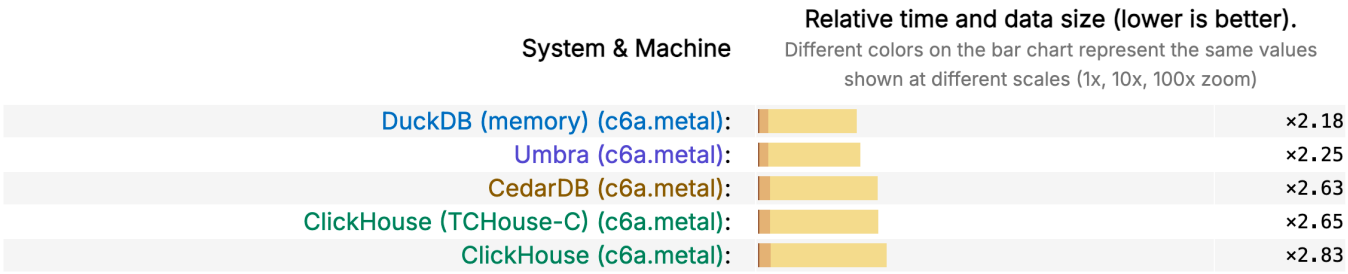
Publication date: 2025-10-09

Author: The DuckDB team

TL;DR: The DuckDB landing page makes some strong claims about DuckDB's popularity. In this blog post, we show evidence for these claims.

#1 on ClickBench

On October 9, 2025, DuckDB's in-memory variant hit #1 on the popular ClickBench database benchmark:



This result was made possible due several performance optimizations in DuckDB v1.4.

> **Update.** As of October 26, the rules of ClickBench changed. The new rules prevent in-memory databases from showing "cold" (and "combined") results. After this change, DuckDB is the #1 open-source system in hot runs, closely trailing the leader in that category, Umbra, a closed-source research prototype.

#3 Most Admired System on Stack Overflow

In Stack Overflow's 2024 Developer Survey, DuckDB was named among the **top-3 most admired database systems**. In the 2025 survey, it achieved position #4 (just 0.2% behind SQLite) but it made up for this by a significant increase in usage, which jumped from 1.4% in 2024 to 3.3% in 2025.

20+ Fortune-100 Companies Use DuckDB

We estimated the number of Fortune-100 companies who use DuckDB by cross-checking self-reported affiliations in the DuckDB issue tracker against a list of Fortune-100 companies.

25M+ Downloads / Month

DuckDB's Python packages has almost 25 million monthly downloads on PyPI alone. This is complemented with the downloads of other popular clients such as the CLI, Go, Node.js, Java, R, Rust and so on.

TPC-H SF 100,000

DuckDB is not only fast but it is also scalable. We have recently run the queries of the TPC-H workload on the scale factor 100,000 dataset, which is equivalent to 100,000 GB of CSV files. Obviously, processing such a dataset size requires the ability to spill to disk.

We ran the experiment on an `i8g.48xlarge` EC2 instance, which has 1.5 TB of RAM and 192 CPU cores (AWS Graviton4, Arm64). This instance has 12 NVMe SSD disks, each 3750 GB in size. We created a RAID-0 array from them to have a single 45 TB partition and formatted it using XFS.

We generated the dataset with the `tpchgen-cli` tool, a pure Rust implementation of the TPC-H generator. We configured the generator to produce chunks of Parquet files and loaded them into DuckDB. The final DuckDB database was about 27 TB in size (as a single file!).

DuckDB completed all 22 queries of the benchmark using its larger-than-memory processing. For some queries, this required spilling *about 7 terabytes of data* to disk. The median query runtime was 1.19 hours and the geometric mean runtime was 1.13 hours.

Streaming Patterns with DuckDB

Publication date: 2025-10-13

Author: Guillermo Sanchez

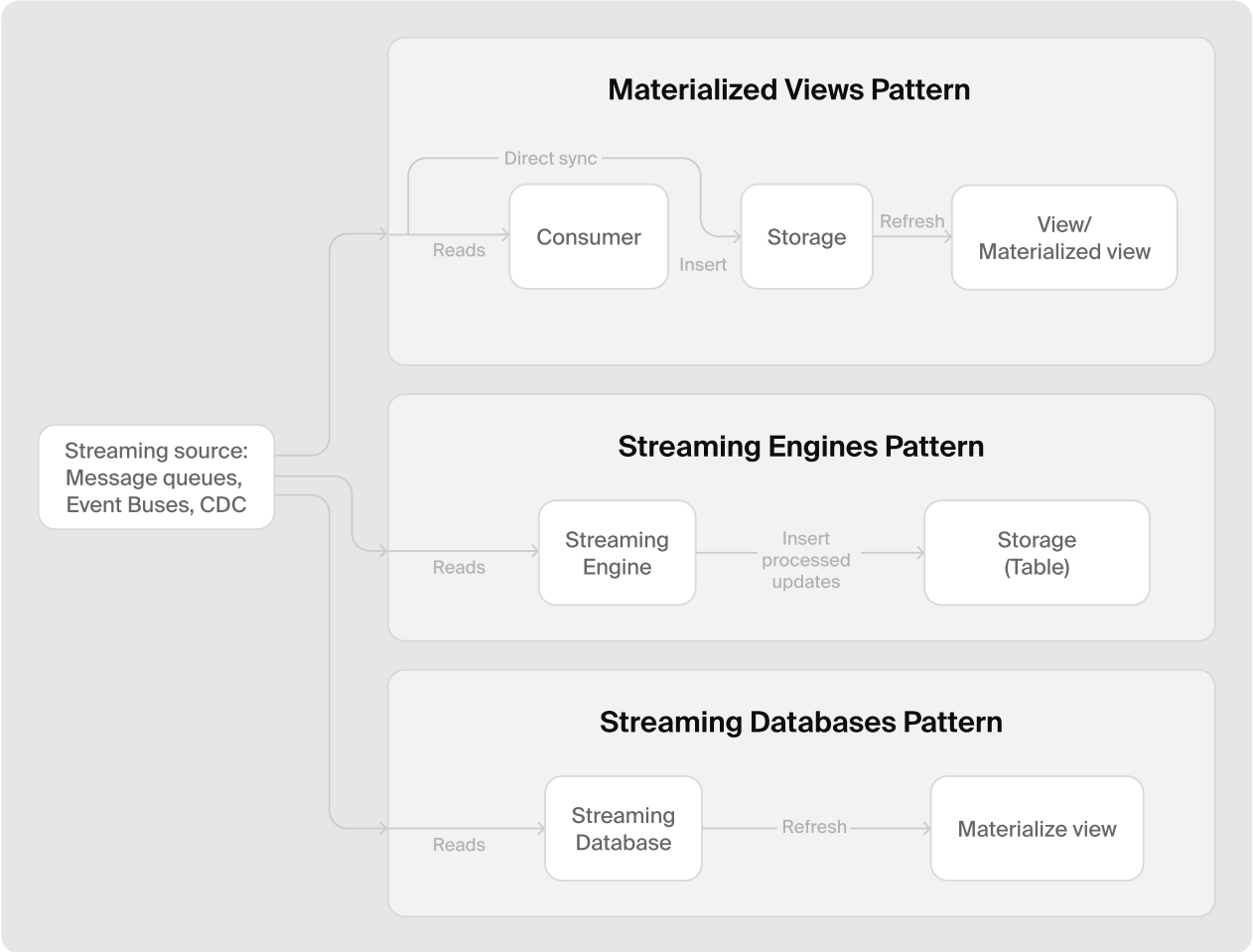
TL;DR: DuckDB used for streaming analytics? This post will show you some patterns in which you can use DuckDB to refresh your data at near real-time speed.

The words “DuckDB” and “streaming” don't usually make it into the same sentence. Maybe this is because DuckDB has been positioned as an all powerful (but very lightweight) OLAP database. Or maybe this is because the ecosystem of streaming analytics has centered around names such as Kafka, Flink and Spark Streaming, and most recently players trying to change the game like Materialize or RisingWave. But can DuckDB be used in the context of streaming analytics? What is streaming analytics in the first place?

Streaming Analytics Patterns

The simplest definition: streaming analytics is the *act of updating an analytical view of your data at near real-time speed as new data comes in.* For example, if three new sessions have just started in your website, the process of gathering those session events and updating the count (+3) is streaming analytics. Streaming analytics is not, in my modest opinion, just about inserting those 3 session events in a table – that would fit well within the realm of a transactional workload. Streaming analytics is also not pushing this events to a Kafka topic and sinking to another system. If you don't update the analytical view of your data, I wouldn't call it streaming analytics.

Now that we have a definition, let's take a look at three common architectural patterns in streaming analytics. The names given to these patterns are of my own making, but I think they help differentiating them from one another.



{:
.lightmode-img }

- In the **Materialized View Pattern**, it is very common to use a cloud data warehouse with support for materialized views (such as BigQuery or Snowflake). The stream of events is usually sunk to a raw table and a materialized view is created on top. This pattern is generally conceived as having a higher latency than the next two. However, there is not that much benchmarking around to conclude anything.
- The **Streaming Engine Pattern** uses the more traditional ETL approach. A separate process using a streaming engine consumes the messages from the source, queries are then done on the fly and results are stored in a persisted table. Common engines are Spark Streaming, Flink, Kafka Streams or most recently Arroyo. This has traditionally come with a set of complications (e.g., dealing with watermarks, state management, increased memory load for infinite queries, etc.).
- The **Streaming Database Pattern** is similar to the previous one in terms of latency but drastically simplifies the experience. Streaming databases like RisingWave or Materialize can directly read from the streaming source and update your materialized view on the fly. They aim at keeping ACID consistency and allowing clients to query data using the PostgreSQL wire protocol.

“Where does DuckDB fit in all this?” – you may ask. Well, DuckDB fits well with patterns one and two. Even if DuckDB does not support materialized views (yet), we can work around this limitation and implement these patterns to still get very good results.

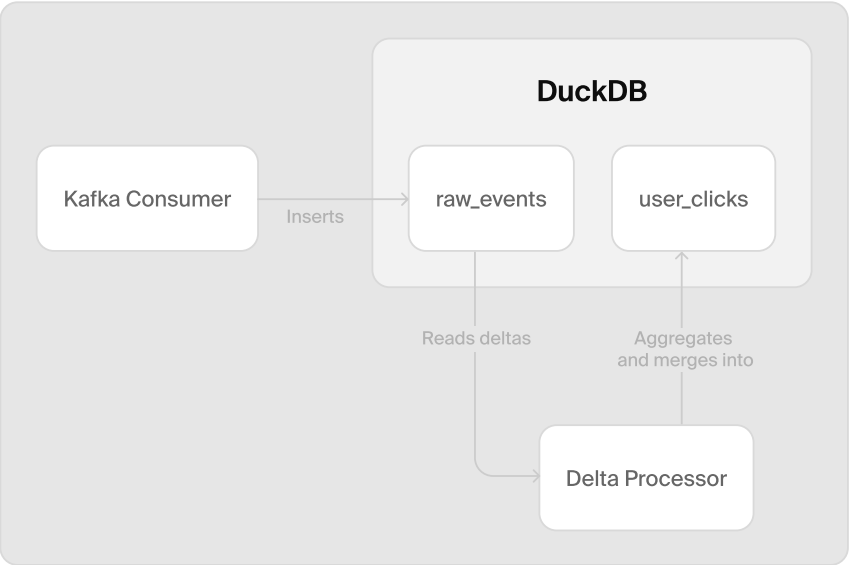
> Interestingly, the streaming engine industry doesn't have many official benchmarks. The Nexmark benchmark seems to be the most common, but there are not many published results comparing engines using this benchmark.

Materialized View Pattern: Cooking Our Own Materialized View with DuckDB

We know that DuckDB is very fast at aggregating data on the fly and also performs very well in transactional workloads (for an OLAP system). So does DuckLake's lakehouse format, thanks to its data inlining feature. In this section we are going to see both DuckDB and DuckLake in action, acting as a sink for Kafka and calculating new metric values based on deltas.

> All the patterns are going to do the same thing in different ways. That is, read events from a Kafka topic and update the analytical view, which can be a persisted table or a view on top of a raw table. What happens in between is what differentiates these patterns.

Querying Deltas with DuckDB



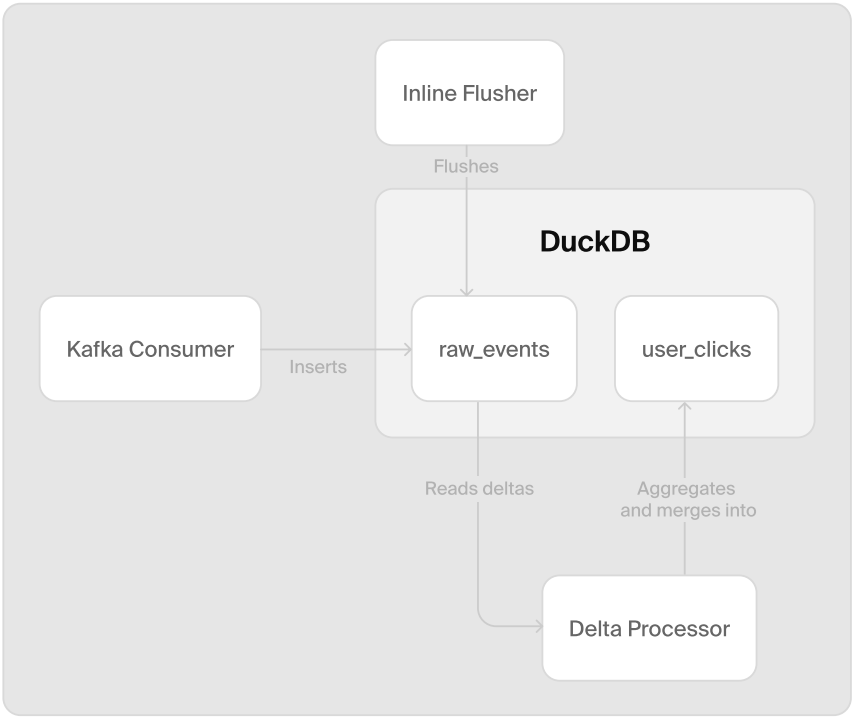
{: .lightmode-img }

The key component in this diagram is what I call “Delta Processor”. This component is basically a function that loops periodically and runs a query to aggregate new data inserted in the `raw_events` table and to update the analytical view, in this case a persisted table called `user_clicks`. This is the query that runs periodically to update `user_clicks` with the new delta:

```
MERGE INTO user_clicks AS dest
USING (
    SELECT
        user_id,
        user_name,
        count(*) AS count_of_clicks,
        max(timestamp) AS updated_at
    FROM raw_events
    WHERE event_type = 'CLICK'
      AND (<LATEST_UPDATED_AT> IS NULL
       OR timestamp > <LATEST_UPDATED_AT>)
    GROUP BY user_id, user_name
) AS src
ON dest.user_id = src.user_id
WHEN MATCHED THEN
    UPDATE SET
        count_of_clicks = dest.count_of_clicks + src.count_of_clicks,
        updated_at = src.updated_at
WHEN NOT MATCHED THEN
    INSERT (user_id, user_name, count_of_clicks, updated_at)
    VALUES (src.user_id, src.user_name, src.count_of_clicks, src.updated_at);
```

You can check the full pipeline in this repository.

Using DuckLake's Change Data Feed



{: .lightmode-img }

This pattern is very similar to 1.1 but with some specifics to DuckLake:

1. We are using DuckLake's Data Inlining to speed up insertion without writing too many small files.
2. The Delta Processor component can take advantage of DuckLake's Data Change Feed to avoid scanning unnecessary data.
3. We have an extra component, called “Inline Flusher”, that periodically flushes inlined data from the metadata catalog to parquet files of the specified file size (512 MB by default). This is a maintenance operation that will keep DuckLake performant.

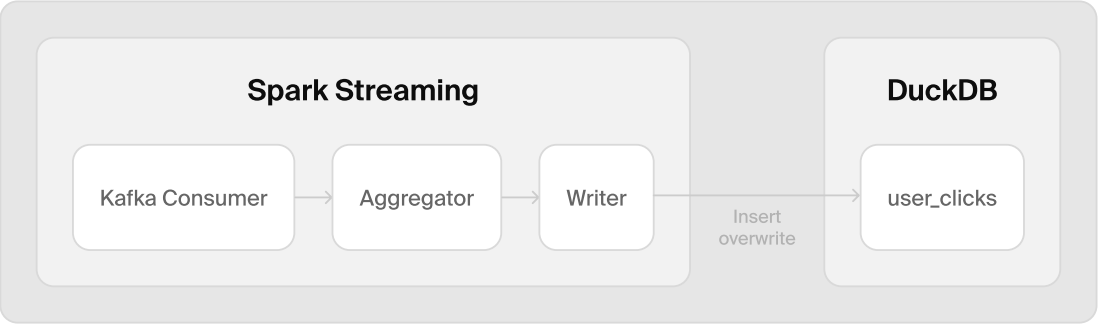
You can check the full pipeline in this repository.

> In order to make better use of filter pushdowns and file pruning, partitioning the data by `timestamp` is recommended

Streaming Engine Pattern: Streaming Engines and DuckDB

Most established streaming engines (Spark Streaming, Flink, Kafka Streams) are JVM based. They can therefore insert data in DuckDB using the JDBC protocol. This pattern is usually a bit difficult to manage. Long running streaming queries tend to consume a lot of memory and restarting interrupted streaming queries always makes me skip a beat. However, it can be a very low latency solution for very large streams of data.

Using Spark Streaming and Sink to DuckDB



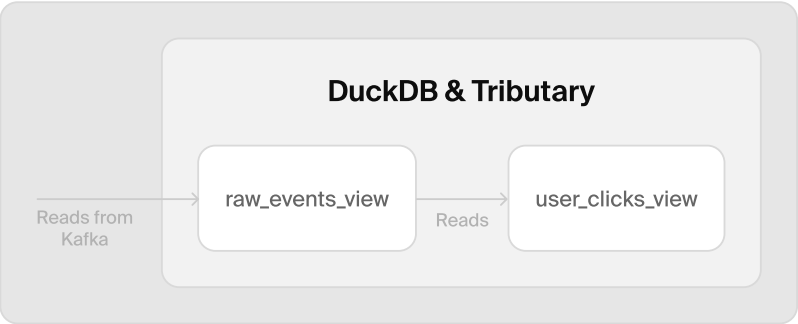
{: .lightmode-img }

In this diagram we can see that most of the components are managed by the Spark Streaming runtime. In Spark Streaming, all of this is contained in a streaming query. When the micro batching mode is being used (like it is the case in this example) you can pass a custom function to the writer that allows you to write each batch in the way you desire. In our case, we just use a JDBC connection and overwrite the destination table (`user_clicks`).

We can also see that there are no intermediate results being saved, meaning in this particular case we do not have a `raw_events` table. This is not a pattern that I love since for auditing purposes I would prefer to store the raw data to ensure that my streaming job isn't doing something funky. In this case, Spark Streaming relies on checkpoints to keep the state and make sure that data is processed just once and queries are able to restart without missing or duplicating data consumed from the Kafka topic.

You can check the full pipeline in this repository.

Bonus: Using DuckDB Tributary Extension to Directory Query Kafka



{: .lightmode-img }

This setup is the most similar thing to the Streaming Databases Pattern that you can do right now with DuckDB. Powered by the `tributary` DuckDB community extension, you can create a view or a table that reads directly from a Kafka topic. To simulate materialized views, we are using views for this specific example. The following query showcases how simple this process is:

```
CREATE VIEW IF NOT EXISTS raw_events_view AS
    SELECT
        * EXCLUDE message,
        decode(message)::JSON AS message
    FROM
        tributary_scan_topic(<TOPIC>, "bootstrap.servers":="localhost:9092");
```

Currently this extension has no state management. Every time this view is queried, we would be reading the whole topic from offset 0. This is not ideal since Kafka has a limited retention policy, which means that at some point it will start flushing messages. A way around this is to materialize this messages to tables and use the offset (or a timestamp) to keep track of what has been ingested.

You can check the full pipeline in this repository.

> This is an experimental extension from Query.Farm.

Some Thoughts

Conclusions always feel very subjective, so I rather write about some of my thoughts regarding streaming patterns in general and particularly around DuckDB.

The Materialized View Pattern is usually good enough. My hot take is that most use cases for analytics are usually covered by the Materialized View Pattern without the need of complexity that comes with other patterns. I believe that DuckDB is very well suited for this pattern because for a small OLAP, it does incredibly well at handling large amounts of streaming inserts. In this article DuckDB was pushed to the limit and was able to handle more than one million rows inserted per second. Also note that **materialized views are on the DuckDB's long-term roadmap**, so this pattern will become even simpler in the near future.

If you are streaming to a lakehouse, you should know that DuckLake's Data Inlining feature was specifically designed to deal with high-throughput inserts while solving the small file problem. This makes DuckLake a great candidate for this pattern if you have a lakehouse-like architecture.

Streaming Engines and Streaming Databases can be hard (or expensive). At scale, Streaming Engines can be hard to manage. It is an evolving field and some work is being done to make forever running streaming queries easier to operate. For example, Apache Fluss is being built with the idea to solve some of the shortcomings described in this post. However, it does add another layer of complexity to an already complex streaming architecture.

Streaming databases are a very elegant solution and have the potential to be very nice to use. However, if you are looking to host the solution, this will require some expertise since these systems are considerably complex (see RisingWave's architecture). This pushes practitioners to buy rather than host and maintain this complex system, which can be costly.

Whatever you choose for your architecture, make sure that the effort you put into it corresponds to your needs. And next time you think of streaming, make sure you also think about DuckDB.

Uncovering Financial Crime with DuckDB and Graph Queries

Publication date: 2025-10-22

Author: Daniël ten Wolde

TL;DR: You can process graphs in DuckDB! In this post, we show how to use DuckDB and the DuckPGQ community extension to analyze financial data for fraudulent patterns with the SQL/PGQ graph syntax that's part of SQL:2023.

Following the money is harder than it looks. Sophisticated criminals hide their tracks using long, complex chains of transactions, hoping to obscure the origin of illicit funds. Unraveling these networks is a classic graph problem: you're looking for suspicious patterns and hidden paths in a vast web of accounts and transactions.

For years, this kind of analysis often meant exporting data to a specialized graph database, adding complexity and overhead. But what if you could perform this powerful graph analysis directly within your daily driver database?

This is where DuckDB's extensibility shines. In this blog post, we'll dive into a financial dataset and use DuckDB with a graph query extension to identify the kinds of patterns that could indicate a money laundering scheme or otherwise high-risk accounts.

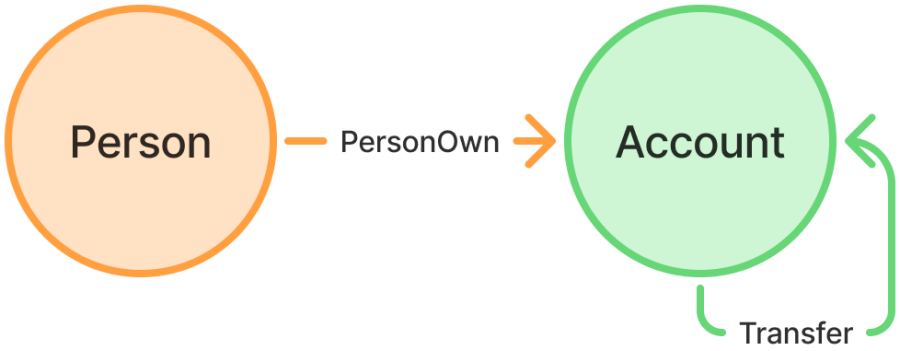
From Relational Tables to a Property Graph

Before we can hunt for suspicious activity, we need to understand our data. We're using the LDBC Financial Benchmark dataset, which simulates a financial network. To attach to the database with the dataset, run:

```
ATTACH 'https://blobs.duckdb.org/data/finbench.duckdb' AS finbench;
USE finbench;
```

> To follow along with the examples in this post, it is recommended to use DuckDB v1.4.1.

In this blog post we will use a subset of the dataset with tables for `Person`, `Account`, and the `AccountTransferAccount` table that links them.



Let's start by getting a feel for the scale of our network:

```
SELECT
    (SELECT count(*) FROM Person) AS num_persons,
    (SELECT count(*) FROM Account) AS num_accounts,
    (SELECT count(*) FROM AccountTransferAccount) AS num_transfers;
```

num_persons int64	num_accounts int64	num_transfers int64
785	2055	8132

This query gives us a quick overview of the number of entities and connections we're dealing with. As the schema diagram above illustrates, these tables of accounts and transfers already form a graph, a structure made of *nodes* or *vertexes* (the entities), connected by *edges* (representing relations – *hint hint* – between the entities).

To make our queries more powerful, we'll use the Property Graph model. This is just a formal way of saying we can add descriptive labels, which are general types like `Account` and `Person`, as well as specific properties, like `accountId` and `nickname`.

If you're thinking this sounds a lot like the relational model, you're exactly right. A `Person` table is just a collection of nodes with the label `Person`, and its columns are the properties. This natural mapping is what makes a high-performance relational database like DuckDB a perfect foundation for graph analytics.

Property Graphs in DuckDB

To write our graph queries, we could use DuckDB and the SQL we are familiar with. But let us make life a little simpler for ourselves and leverage DuckDB's rich extension ecosystem. We will be using DuckPGQ, a community extension that adds support to DuckDB's parser for a new visual graph syntax. This new syntax is SQL / Property Graph Queries (SQL/PGQ), which is part of the official SQL:2023 standard. SQL/PGQ is partially inspired by the popular graph query language Cypher.

> The DuckPGQ extension started out as a research prototype and is now available as a community extension.

Installing and loading the extension is as simple as it gets:

```
INSTALL duckpgq FROM community;
LOAD duckpgq;
```

In DuckPGQ, the first step is to create the property graph, acting as a layer on top of the tables we have created earlier.

```
CREATE PROPERTY GRAPH finbench
VERTEX TABLES (
    Person,
    Account
)
EDGE TABLES (
    AccountTransferAccount
        SOURCE KEY (fromId) REFERENCES Account (accountId)
        DESTINATION KEY (toId) REFERENCES Account (accountId)
        LABEL Transfer,
    PersonOwnAccount
        SOURCE KEY (personId) REFERENCES Person (personId)
        DESTINATION KEY (accountId) REFERENCES Account (accountId)
        LABEL PersonOwn
);
```

During the creation of the property graph, we make a clear distinction between `VERTEX` tables and `EDGE` tables. For `VERTEX` tables, we only have to specify the name of the table. For `EDGE` tables, a little more work is required since for both the `SOURCE` and the `DESTINATION`, we need to specify the column in the edge table that forms the key for the `SOURCE` or `DESTINATION`. This is the same principle as defining a `FOREIGN KEY` constraint, linking our edge table back to the node tables it connects. The `LABEL` clause gives a clean name to the relationship type. While our table is named `AccountTransferAccount`, the edges within it represent a `Transfer` relationship. This is the name we'll use in our graph queries.

Now that we have created our property graph, we are ready to investigate the financial data and uncover its secrets!

Graph Processing

When talking about graph processing in databases, we typically refer to these types of operations:

- Pattern matching, finding a pattern in our data.
- Path-finding, finding a path in our data, potentially of variable length.

Let's see how we can leverage DuckDB and DuckPGQ for these two tasks.

Hunting for Suspicious Activities

As previously mentioned, SQL/PGQ introduces a visual graph syntax to formulate graph patterns more naturally. Now, let's use it to hunt for patterns that might indicate a money laundering scheme.

A common technique used to hide illicit funds is called *smurfing*. The goal of smurfing is to break down a single large transfer, potentially triggering reporting requirements, into smaller transactions over time.

We can search for this behavior by looking for pairs of accounts with a high number of transactions but a relatively low average amount. Let's set the threshold for the average amount at $50,000 and see if we can find any high-frequency relationships:

```
SELECT
    fromName,
    count(amount) AS number_of_transactions,
    round(avg(amount), 2) AS avg_amount,
    toName
FROM GRAPH_TABLE (finbench
    MATCH (a:Account)-[t:Transfer]->(a2:Account)
```

```
    COLUMNS (a.nickname AS fromName,
             t.amount,
             a2.nickname AS toName
            )
)
GROUP BY ALL
HAVING avg_amount < 50_000
ORDER BY number_of_transactions DESC, avg_amount ASC
LIMIT 5;
```

Running the query leads us to the following result:

fromName varchar	number_of_transactions int64	avg_amount double	toName varchar
Noe Trites	1	49365.04	Dale Croucher
Madeleine Bussing	1	46663.56	Delphine Primiano
Bonnie Centeno	1	46663.56	Maile Boon
Darci Sheedy	1	44856.02	Carmella Estelle
Marguerita Gurne	1	44393.68	Delphine Primiano

The query worked, but the result does not show any signs of suspicious activity with the number of transactions always being 1. To understand why, let's break down how the query was constructed.

The magic happens inside the `FROM` clause. The `GRAPH_TABLE (finbench ...)` function allows us to run a graph query over the property graph we have just created and treat its output like a regular table.

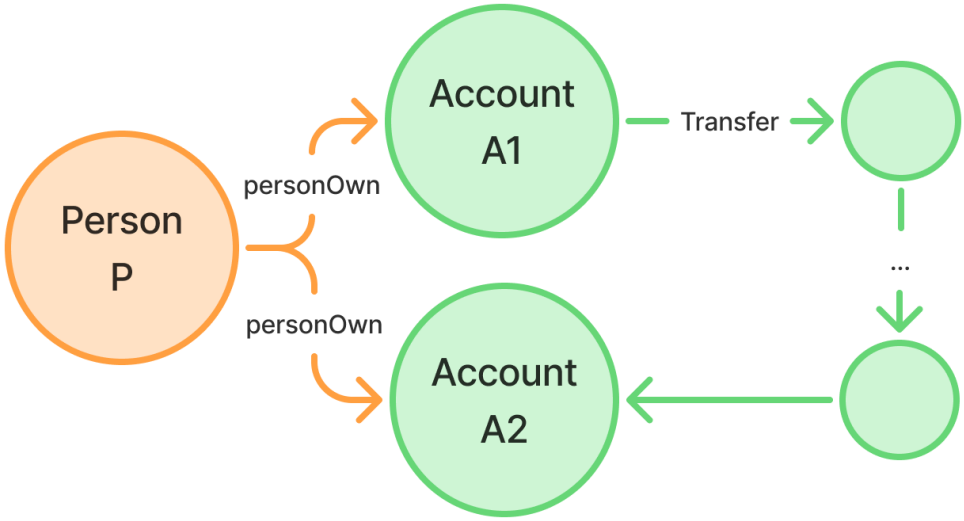
The `MATCH (a:Account)-[t:Transfer]->(a2:Account)` clause is the core of our pattern. It visually describes what we're looking for: a simple transfer from one account `(a:Account)` to another `(a2:Account)`. The `()` denote nodes and the `[]` denotes the connecting edge with the ASCII-style arrow `->` showing the direction of the edge. The `COLUMNS(...)` clause then acts like a `SELECT` list for our pattern, pulling out the nickname from the accounts and the amount from the transfer.

The beauty of SQL/PGQ is that the result of this graph pattern match can be seamlessly returned into the standard SQL we already know. We use `GROUP BY ALL` to aggregate all transfers between the same two people, and our `HAVING avg_amount < 50_000` clause filters for the smurfing pattern we defined.

We know our query is correct but also that this simple *smurfing* pattern is not present in our dataset. This means we must investigate further, using potentially more complex patterns. This leads us to a more powerful feature of graph queries: finding structural patterns that are very difficult to express with traditional SQL `JOIN`s, such as transaction paths.

Finding Paths in the Transactions

Another classical example of possible fraudulent behavior is a cycle of transactions where the money circles back to the person who sent the first transaction in the chain. Writing a SQL query to answer this question is, using the traditional syntax, incredibly difficult. Try it yourself, after first reading this section! We will show the answer in the next section.



With SQL/PGQ writing queries that involve finding paths has become significantly easier. The diagram above illustrates the query pattern we'll use with the goal of finding a path of one or more transfers between two different accounts (A1 and A2) owned by the same person (P). Remember that persons can own multiple account. With the following query we will try to find cycles between all the accounts that the person with id 125 owns:

```
FROM GRAPH_TABLE(finbench
    MATCH p = ANY SHORTEST
                  (p:Person)-[o1:PersonOwn]->(a1:Account)
                  -[t:Transfer]->+
                  (a2:Account)<-[o2:PersonOwn]-(p:Person)
WHERE
    p.personId = 125 AND a1.accountId <> a2.accountId
    COLUMNS (
        path_length(p) AS path_length,
        a1.accountId AS start_account,
        a2.accountId AS end_account
    )
)
ORDER BY path_length;
```

The result shows that there are cycles for this person owning multiple accounts of varying lengths:

path_length int64	start_account int64	end_account int64
8	4753267931712848113	4794926228266025204
8	4769874955338776819	4794926228266025204
8	4796615078126289138	4769874955338776819
9	4753267931712848113	4769874955338776819
9	4769874955338776819	4753267931712848113

9	4794926228266025204	4753267931712848113
9	4796615078126289138	4753267931712848113
9	4796615078126289138	4794926228266025204
12	4794926228266025204	4769874955338776819

Once again, the magic happens inside the FROM clause where we now create a MATCH that finds ANY SHORTEST path along the given pattern. The first part of the pattern finds all the accounts owned by the Person 125: `(p:Person)-[o1:PersonOwn]->(a1:Account)`. In the second part the path-finding occurs. Take special note of the +, which indicates that for the pattern `(a1:Account)-[t:Transfer]->+(a2:Account)` the two nodes `(a1)` and `(a2)` do not necessarily need to be connected. With the +, we indicate there must be one or more `Transfers` between these two accounts, with no upper bound set. Finally, to tie it all up, we take the destination account and check whether it is owned by the same person p.

The results confirm our suspicions. Our query found multiple non-obvious paths between the accounts owned by Person 125, with path lengths ranging from 8 to 12 transfers.

Each row represents a hidden chain of transactions connecting two of the person's accounts. More interestingly, we can see clear cyclical patterns. For instance, the query found a 9-step path from account `4753267931712848113` to `4769874955338776819`, and another 9-step path flowing in the opposite direction. This suggests a sophisticated and intentional effort to move money between accounts, a strong indicator that warrants further investigation.

Doing it the Old-Fashioned Way

Earlier, we challenged you to think about how you would find these ownership cycles using traditional SQL. As promised, here is the answer.

Before we dive into the query, there are two important notes to keep in mind when comparing it to the SQL/PGQ version:

1. **Performance Safeguard:** The query requires a manual upper bound on the path length (`ps.depth < 11`) to prevent infinite recursion and potentially quadratic runtimes on dense graphs. The SQL/PGQ `->+` syntax does not require this.
2. **Path Length Difference:** You'll notice the `path_length` in this query's result is two hops shorter than the result from DuckPGQ. This is because this query only counts the `Transfer` edges, whereas the DuckPGQ query also includes the two `PersonOwn` edges in its path calculation.

With that in mind, here is the traditional recursive CTE to find the shortest path between any two accounts owned by the same person:

```
WITH RECURSIVE
    owned_accounts AS (
        SELECT accountId
        FROM PersonOwnAccount
        WHERE personId = 125
    ),
    path_search(start_node, end_node, path, depth) AS (
        -- Base case: a direct transfer from one of the person's accounts
        SELECT
            fromId,
            toId,
            [fromId, toId],
            1
        FROM
            accounttransferaccount
        WHERE
            fromId IN (SELECT accountId FROM owned_accounts)
        UNION ALL
        -- Recursive step: find the next transfer in the path
        SELECT
            ps.start_node,
            t.toId,
            list_append(ps.path, t.toId),
            ps.depth + 1
```

```
        FROM path_search ps
        JOIN accounttransferaccount t ON ps.end_node = t.fromId
        WHERE
            t.toId NOT IN (SELECT unnest(ps.path)) AND ps.depth < 11
    )
SELECT distinct start_node, end_node, min(depth) AS path_length
FROM path_search
WHERE end_node IN (SELECT accountId FROM owned_accounts)
  AND start_node <> end_node
GROUP BY ALL
ORDER BY path_length;
```

As you can see, the logic requires a `WITH RECURSIVE` clause, manual path tracking in a list, and explicit cycle detection. This is exactly the kind of verbose and complex query that the visual syntax of SQL/PGQ is designed to eliminate.

Wrapping Up

We began this post with a simple goal: to see if we could use DuckDB to hunt for the complex patterns and hidden paths typical of graph analysis. After diving into the Financial Benchmark dataset, the conclusion is clear: you can.

The key takeaway is the drastic improvement in usability. We saw how the visual syntax of SQL/PGQ, enabled by the DuckPGQ extension, transformed a sophisticated “ownership cycle” query from a monstrous recursive CTE into a few readable lines of code. This is exactly the kind of expressive power needed for real-world analytical tasks. For more information and complete documentation on DuckPGQ, be sure to visit its official website: duckpgq.org.

Just as importantly, this entire investigation was performed directly within DuckDB. By leveraging the extension ecosystem, we tapped into the power of graph queries without ever needing to export our data or manage a separate, specialized system. Everything runs on top of DuckDB's high-performance vectorized engine, right where the data lives.

For powerful, analytical graph queries, DuckDB isn't just a viable alternative, it's a powerful, natural solution. The next time you think about analyzing connections in your data, remember that the tools you need are just an `INSTALL` away.

Frozen DuckLakes for Multi-User, Serverless Data Access

Publication date: 2025-10-24

Author: Mark Harrison

TL;DR: We show how you can build high-performance data lakes with no moving parts.

Relational Charades: Turning Movies into Tables

Publication date: 2025-10-27

Author: Hannes Mühleisen

TL;DR: You can store and even process videos in DuckDB. In this post, we show you how.

> Quote Your scientists were so preoccupied with whether they could, they didn't stop to think if they should.
>
> -- Dr. Ian Malcolm, Jurassic Park (1993)

Here at team DuckDB, we *love* tables. Tables are a timeless elegant abstraction that precedes literature by about a thousand years. Relational tables specifically can represent *any* kind of information imaginable. But just because something *can* be done it is not always a great idea to do so. Can we build a rocket propelled by a nuclear chain reaction that irradiates the land it flies over? Yes. Should we? Probably not.

Disclaimer

Array-like data like images and videos are a *textbook example* of something that might not benefit from storing them in a database. While of course any binary data can be added to tables as BLOBs, there is not that much added value from it. Sure, it's harder to lose the image compared to the industry standard solution, storing a file name that points to the image. But there are not that many meaningful operations that can be done on BLOBs other than store and load. Without adding some overhyped AI tech, you can't even ask the database what the picture shows.

Array data also has its own world of highly specialized file formats and compression algorithms. Just think of the ubiquitous MPEG-4 standard to store movies. They are approximate (not exact, lossy) formats that are designed around human perception models, which is why they can avoid storing things people do not notice. They achieve impressive compression rates, with a two hour "Full" HD movie compressing to about 2 GB using MPEG-4.

Ignoring the Disclaimer

But what would it feel like to turn a movie into a table? (Very) deep down, a movie is just a series of fast-moving pictures ("frames"), typically at something around 25 frames per second. At that speed, our monkey-brain cannot distinguish the separate images any more and is fooled into thinking that we are watching smooth movement. Side note for the younger generations, a strip of pictures was the way we shipped around movies for more than 100 years.

So a series of pictures it is. Each picture can be further deconstructed into a two-dimensional array (a "matrix") of points, so-called "pixels". Every pixel in turn consists of three numbers, one each for the intensity of red, green and blue, or RGB for short. Note that we're ignoring the audio tracks in this post, but in principle it would work the exact same way, just with a different kind of intensity.

As an added complexity, the relational model (famously) does not require an absolute order of records. So all the various offsets have to be made explicit to not lose information. This of course greatly increases the size of our data set. We end up with a table that looks like this:

i	y	x	r	g	b
0	0	0	4	5	1
0	0	1	4	5	1
0	0	2	5	6	2

i	y	x	r	g	b
0	0	3	8	9	4
0	0	4	9	10	5
0	0	5	11	12	8
0	0	6	11	12	8
0	0	7	11	12	8
0	0	8	9	10	5
0	0	9	9	10	5

We have the time offset or frame number `i`, we have `x` and `y` for the pixel position in the frame, and `r`, `g` and `b` for the color components red, green and blue. Quite involved.

But now the movie is just a single table. If only just we had a conventional and guaranteed total order of rows, we could in theory skip all columns except for `r`, `g` and `b`, because with a known resolution all other columns can be inferred. This is coincidentally also how actual movie data files are stored, ignoring compression. This is another reason why maybe relational tables are not the best place for a movie to live in, but if all you have is a hammer. We could also have used some more modern features of SQL and use nested fields (a `LIST` in DuckDB), but let's keep it to a table even System R could have dealt with. In addition, having explicit offsets does not require nebulous conventions or additional metadata to know along with axis order the array data was serialized.

Experiments

To investigate this daft idea further (for Science!), we convert the 1963 classic "Charade", a "romantic screwball comedy mystery film" starring Audrey Hepburn and Cary Grant to a DuckDB table. This movie was picked because it is *accidentally in the Public Domain* because a screw-up in the wording of the copyright notice (no, really). Because of this, you can actually freely download this movie from the Internet Archive.

Cary Grant
Audrey Hepburn
You can expect the unexpected when they play...
"Charade"
a STANLEY DONEN production
in TECHNICOLOR®
UNIVERSAL CITY STUDIOS
co-starring
Walter Matthau James Coburn
Screenplay by PETER STONE
Produced and Directed by STANLEY DONEN
Music-HENRY MANCINI
A UNIVERSAL RELEASE

Since we're just creating a table, we will use DuckDB's native storage format. Here is the *complete* code snippet we used to convert the movie. In fact, this code should actually be generic enough to convert anything that `ffmpeg` can read to a table. Just in case you would want to try this at home on your own movies.

```
import imageio
import duckdb

# setup movie reading
vid = imageio.get_reader("Charade-1963.mp4", "ffmpeg")
dim_x = vid.get_meta_data()['size'][0]
dim_y = vid.get_meta_data()['size'][1]
rows_per_frame = dim_y * dim_x

# setup a DuckDB database and table
con = duckdb.connect()
con.execute("ATTACH 'charade.duckdb' AS m (STORAGE_VERSION 'latest'); USE m;")
con.execute("CREATE TABLE movie (i BIGINT, y USMALLINT, x USMALLINT, r UTINYINT, g UTINYINT, b UTINYINT)")

# those offsets don't change between frames, so pre-compute them
con.execute("CREATE TEMPORARY TABLE y AS SELECT unnest(list_sort(repeat(range(?), ?))) y", [dim_y, dim_x])
con.execute("CREATE TEMPORARY TABLE x AS SELECT unnest(repeat(range(?), ?)) x", [dim_x, dim_y])

# loop over each frame in the movie and insert the pixel data
for i_idx, im in enumerate(vid):
    v = im.flatten()
    r = v[0:len(v):3]
    g = v[1:len(v):3]
    b = v[2:len(v):3]

    con.execute('''INSERT INTO movie
        FROM repeat(?, ?) i -- frame offset
        POSITIONAL JOIN   y -- temp table
        POSITIONAL JOIN   x -- temp table
        POSITIONAL JOIN   r -- numpy scan
        POSITIONAL JOIN   g -- numpy scan
        POSITIONAL JOIN   b -- numpy scan
        ''', [i_idx, rows_per_frame])
```

This script makes use of not just one, but (at least) *two* cool DuckDB features. First, we use so-called replacement scans to directly query the NumPy arrays `r`, `g`, and `b`. Note that those have not been created as tables in DuckDB nor registered in any way, but they are referenced by name in the `INSERT`. What happens here is that DuckDB inspects the Python context for the missing "tables" and finds objects with those names that it can read. The other cool feature is the `POSITIONAL JOIN`, which lets us stack multiple tables horizontally by position without running an actual (expensive) `JOIN`. This way, we assemble all the columns we need for a single frame in a bulk `INSERT`, which executes quite efficiently.

The movie file we have has a frame rate of 25 frames per second at a (DVD-ish) resolution of 720x392 pixels. The total runtime is 01:53:02.56 seconds, which comes down to 169 563 individual frames. Because we have a row for each pixel we end up with 169 563 * 720 x 392 rows, or 47 857 461 120. 47 billion rows! Finally Big Data! When stored as a DuckDB database however, the file size is "only" around 200 GB. Totally doable on a laptop!

DuckDB's lightweight compression performs quite well here, given that in a naive binary format we would have to store at least 15 bytes per row. If we multiply that with the row count (47 billion, remember) we would end up at around 700 GB in storage for this hypothetical naive format.

Of course, by turning the data into a relational table we add a bunch of previously implicit information due to the lack of ordering in relations. If we just stored the raw pixel bytes, for example as an implicitly ordered series of BMP (bitmap) files, we would end up with the same amount of bytes as the rows above times three, or 133 GB. Even *including materializing all the offsets*, the DuckDB file still manages to end up at a comparable size (200 GB). And of course, comparing the size of the table with the MPEG-4 version of the movie is not entirely fair because MPEG-4 is a *lossy* compression format. Databases can't just randomly decide to compromise on the numerical accuracy of the tables they store!

To prove that the transformation is accurate, let's try to turn the table data for one random frame back into a human-consumable picture:

we will retrieve the corresponding rows from DuckDB, and use some Python magic to turn them back into a PNG image file:

```
import duckdb
import numpy as np
import PIL.Image

frame = 48000

con = duckdb.connect('charade.duckdb', read_only=True)
dim_y, dim_x = con.execute("SELECT max(y) + 1 dim_y, max(x) + 1 dim_x FROM movie WHERE i=0").fetchone()

res = con.execute("SELECT r, g, b FROM movie WHERE i = ? ORDER BY y, x", [frame]).fetchnumpy()
v = np.zeros(dim_y * dim_x * 3, dtype=np.uint8)
v[0:len(v):3] = res['r']
v[1:len(v):3] = res['g']
v[2:len(v):3] = res['b']

img = PIL.Image.fromarray(v.reshape((dim_y, dim_x, 3)))
img.save(f'frame.png')
```

And voila, we can see a wonderful frame with Audrey and Cary appear. This trick can also be used to create a sequence of pictures and write them to a MPEG-4 file again using – for example – the `moviepy` library.

But now that we have a table, we can have some fun with it. Let's do some basic exploration first: we start with `DESCRIBE`, which basically tells us the schema. We knew this of course.

```
DESCRIBE movie;
```

column_name	column_type	null	key	default	extra
i	BIGINT	YES	NULL	NULL	NULL
y	USMALLINT	YES	NULL	NULL	NULL
x	USMALLINT	YES	NULL	NULL	NULL
r	UTINYINT	YES	NULL	NULL	NULL

column_name	column_type	null	key	default	extra
g	UTINYINT	YES	NULL	NULL	NULL
b	UTINYINT	YES	NULL	NULL	NULL

No surprises there. How many rows are there?

```
FROM movie SELECT count(*);
```

count_star()
47857461120

Ah yes, 47 billion. What are the numeric properties of the columns? DuckDB has this neat `SUMMARIZE` statement that computes single-pass summary statistics on a table (or arbitrary query).

```
SUMMARIZE movie;
```

This one is admittedly a bit of a flex. DuckDB can compute elaborate summary statistics on all the 47 billion rows in ca. 20 minutes on a MacBook. Here are the results:

column_name	column_type	min	max	approx_unique	avg	std	q25	q50	q75	count	null_percentage
i	BIGINT	0	169562	150076	84781.0	48948.62184695794429	42429	84751	127137	47857461120	0.00
y	USMALLINT	0	391	430	195.5	113.16028455346597	98	196	294	47857461120	0.00
x	USMALLINT	0	719	840	359.5	207.84589644146592	180	359	540	47857461120	0.00
r	UTINYINT	0	255	252	65.32575855816734	64.8562760255523127	54	96	47857461120	0.00
g	UTINYINT	0	249	249	56.797138446695737	77.0356245603219328	44	77	47857461120	0.00
b	UTINYINT	0	255	252	43.249715985643935	55.3921896326889916	28	61	47857461120	0.00

Since we're basically storing a lot of colors, just how many different combinations of red, green and blue are there, DuckDB?

```
FROM (FROM movie SELECT DISTINCT r, g, b)
SELECT count(*);
```

Any seasoned data engineer would rightfully caution you to run a `DISTINCT` on this many rows. There have just been too many production outages caused by overflowing aggregations. But thanks to DuckDB's larger-than-memory aggregate hash table, we can confidently issue this query. We even get a nice progress bar and (since 1.4.0) a surprisingly accurate estimate of how long the query will take.

count_star()
826568

So roughly 800 thousand different colors. Computing this took about 2 minutes in the end. But what are the frequencies of those colors? Let's compute a histogram of the 10 most-used colors!

```
FROM movie
SELECT r, g, b, count(*) AS ct
GROUP BY ALL
ORDER BY ct DESC
LIMIT 10;
```

r	g	b	ct
17	20	15	106521429
23	25	15	93004303
23	25	13	85552738
13	22	15	81734796
22	24	13	76560295
24	26	15	75376896
15	19	8	74285763
23	24	19	72904497
22	24	12	69269099
24	26	16	62230136

The most common colors here seems to be dark shades of grey. Makes sense! Keep in mind that the MPEG-4 compression is lossy and will probably produce some odd colors as rounding artifacts.

But we can also have some more fun. We have an analytical database system. How about we compute the average frame for every thousand frames and stitch the results back into a movie? It's just a big aggregation. We first create the actual averages:

```
CREATE TABLE averages AS
    FROM movie
    SELECT
        i // 1000 AS idx,
        y,
        x,
        avg(r)::UTINYINT AS r,
        avg(g)::UTINYINT AS g,
        avg(b)::UTINYINT AS b
GROUP BY ALL
ORDER BY idx, y, x;
```

Then, we use Python again to turn this `averages` table into a movie:

```
# some setup omitted

# fetch a bunch of frames in bulk
res = con.execute("SELECT r, g, b FROM averages ORDER BY i, y, x").fetchnumpy()

# split the rgb arrays by frame again
r_splits = np.split(res['r'], num_frames)
g_splits = np.split(res['g'], num_frames)
b_splits = np.split(res['b'], num_frames)

# generate pictures
image_files = []
for i in range(num_frames):
    v = np.zeros(dim_y * dim_x * 3, dtype=np.uint8)
    v[0:len(v):3] = r_splits[i]
    v[1:len(v):3] = g_splits[i]
    v[2:len(v):3] = b_splits[i]
    image_files.append(v.reshape((dim_y, dim_x, 3), order='C'))

# write movie file
clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(image_files, fps=25)
clip.write_videofile('averages.mp4')
```

There is some wrangling here because we want to retrieve the whole frame dataset in bulk and not run a query for every single one. We then use NumPy to split them into frames and stitch the RGB-channels together into the three-dimensional array that the image libraries

like. This does not achieve any business purpose but the results are kind of funny, here is average frame #68, with apologies to the actors:

We can also stitch all the averages together to make a somewhat twitchy average movie:

For some added fun, we could even write a SQL query that turns a frame into a HTML table with one-pixel fields. Below is the result, let's hope your browser can render this and let's thank Cloudflare again for sponsoring our traffic. Here is the somewhat unholy query to generate this:

```
SELECT '<html><body><table style="padding:0px; margin: 0px; border-collapse: collapse;">';
FROM movie
SELECT
    IF(x = 0, '<tr>', '') ||
    printf('<td style="background-color: #%02x%02x%02x; height: 1px; width: 1px"; ></td>', r, g, b) ||
    IF(x = 719, '</tr>', '')
WHERE i = 48000
ORDER BY y, x;
SELECT '</table></body></html>';
```

You can see the result in `movies-table.html` (keep in mind that it's 20 MB and renders each movie pixels as a table field!).

Conclusion

You can probably tell that this post is not entirely serious. Fun was had. But what did we learn? A few things: first, basically anything can be represented as a table, even an obscure 1963 movie. In the grand scheme of things, it is probably not a great idea, there are amazing open-source libraries like `ffmpeg` and apps like `VLC` to deal with movie files, or similarly with their array cousins that contain music or just images. Despite the massive blow-up and billions of rows of data, DuckDB actually handled this pretty well, both its data format and its execution engine. Here at team DuckDB, our mission is to raise your overall confidence wrangling data of all shapes and sizes, and we hope this post contributes to that. And to finish up, just pay attention to your copyright notices!

Announcing DuckDB 1.4.2 LTS

Publication date: 2025-11-12

Author: The DuckDB team

TL;DR: Today we are releasing DuckDB 1.4.2, the second patch release of our LTS edition. The new release ships several bugfixes and performance optimizations. We also fixed vulnerabilities in DuckDB's database encryption, and introduced some (opt-in) logger/profiler features that help users understand performance, and full write support through the Iceberg extension.

In this blog post, we highlight a few important fixes and convenience improvements in DuckDB v1.4.2, the second patch release in DuckDB's 1.4 LTS line. To see the complete list of updates, please consult the release notes on GitHub.

While this is a patch release, we are shipping some small features. In LTS releases, these can come in two forms:

1. We add small opt-in features such as accessing the profiler's output from the logger in this release. These features have been highly-requested from the community and we are confident that these will not cause any issues for people upgrading to the latest release. In fact, using them carefully can help detect and understand performance regressions.
2. Some of DuckDB's extensions that are marked as "experimental" are shipping full-fledged features. For example, this is how we have rolled out support for Iceberg deletes and updates. Extensions are opt-in by nature, so if you stick to core DuckDB and its stable extensions, changes in the experimental extensions will not affect the stability of your installation.

> To install the new version, please visit the installation page to be released due to the extra changes and review rounds required.

Features and Improvements

Iceberg Improvements

Similarly to the v1.4.1 release blog post, we can start with some good news for our Iceberg users: DuckDB v1.4.2 ships a number of improvements for the `iceberg` extension. Insert, update, and delete statements are all supported now:

We will publish a separate blog post on these improvements shortly. Stay tuned!

Logger and Profiler Improvements

Time HTTP Requests The logger can now log the time of HTTP requests (`#19691`). For example, if we query the Dutch railway tariffs table as a Parquet file (`tariffs.parquet`), we can see multiple HTTP requests: a HEAD request and three GET requests:

```
CALL enable_logging('HTTP');
CREATE TABLE railway_tariffs AS
    FROM 'https://blobs.duckdb.org/tariffs.parquet';
SELECT request.type, request.url, request.duration_ms
FROM duckdb_logs_parsed('HTTP');
```

type varchar	url varchar	duration_ms int64
HEAD	https://blobs.duckdb.org/tariffs.parquet	177
GET	https://blobs.duckdb.org/tariffs.parquet	103
GET	https://blobs.duckdb.org/tariffs.parquet	176
GET	https://blobs.duckdb.org/tariffs.parquet	182

Accessing the Profiler's Output from the Logger The logger can now also access the profiler's output (#19572). This means that if both the profiler and the logger are enabled, you can log information such as the execution time of queries:

```
-- Enable profiling to JSON file
-- This is necessary to make sure that queries are profiled
PRAGMA profiling_output = 'profiling_output.json';
PRAGMA enable_profiling = 'json';

-- Enable logging to an in-memory table
CALL enable_logging();

-- Run some queries
CREATE TABLE small AS FROM range(1_000_000);
CREATE TABLE large AS FROM range(1_000_000_000);

PRAGMA disable_profiling;

SELECT query_id, type, metric, value::DECIMAL(15, 3) AS value
FROM duckdb_logs_parsed('Metrics')
WHERE metric = 'CPU_TIME';
```

You can see in the output that the first CREATE statement took about 3 milliseconds, while the second one took 3.3 seconds.

query_id uint64	type varchar	metric varchar	value decimal(15,3)
8	Metrics	CPU_TIME	0.003
9	Metrics	CPU_TIME	3.267

Profiler Metrics The profiler now supports several new metrics. These allow you the get a deeper understanding on where the execution time is spent in queries.

Performance Improvements

DuckDB v1.4.2 also ships some small performance improvements:

- #19477 DuckDB now buffers WAL index deletes, not only appends
- #19644 Detaching from a database is now faster

Vortex Support

DuckDB now supports the Vortex file format through the `vortex` extension on Linux and macOS.

To use `vortex`, first install and load the extension:

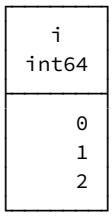
```
INSTALL vortex;
LOAD vortex;
```

Then, you can write Vortex files as follows:

```
COPY (SELECT * FROM generate_series(0, 3) t(i))
TO 'my.vortex' (FORMAT vortex);
```

And read them using the `read_vortex` function:

```
SELECT * FROM read_vortex('my.vortex');
```



i
int64
0
1
2

Fixes

We fixed a vulnerability, several crashes, internal errors, incorrect results and regressions. We also fixed several issues discovered by our fuzzer.

Vulnerabilities

We fixed four vulnerabilities in DuckDB's database encryption:

1. The DuckDB can fall back to an insecure random number generator (`pcg32`) to generate cryptographic keys or IVs.
2. When clearing keys from memory, the compiler may remove the memset() and leave sensitive data on the heap
3. By modifying the database header, an attacker could downgrade the encryption mode from GCM to CTR to bypass integrity checks.
4. Failure to check return value on call to OpenSSL `rand_bytes()`.

See the security advisory for more details.

If you are using database encryption, you are strongly encouraged to update to DuckDB v1.4.2.

> We would like to thank Greg Rubin and Sławek Kosowski for reporting these vulnerabilities!

Crashes and Internal Errors

- #19238 `MERGE INTO` Iceberg table with `TIMESTAMPTZ` columns crashes
- #19355 Unknown expression type invalidates database
- #19357 Expected unified vector format of type `VARCHAR`, but found type `INT32`
- #19455 `MERGE INTO` failed: logical operator type mismatch
- #19498 Window function crash with `pdqsort_loop`
- #19700 RLE select bug

Incorrect Results

- #17757 UUID Comparison in aggregation filter broken on Linux
- #19327 Wrong result for `DISTINCT` and `LEFT JOIN`
- #19377 Array with values shows null depending on query

Regressions

- #19333 DuckDB hangs when using `ATTACH IF NOT EXISTS` on subsequent connections to databases that have previously attached a database file

Storage

- #19424 Fix issue in MetadataManager triggered when doing concurrent reads while checkpointing
- #19527 Ensure that DuckDB outputs the expected `STORAGE_VERSION`
- #19543 Error when setting `force_compression = 'zstd'` in an in-memory environment database

Issues Discovered by the Fuzzer

- duckdb-fuzzer#3389
- duckdb-fuzzer#4208
- duckdb-fuzzer#4296

Data-at-Rest Encryption in DuckDB

Publication date: 2025-11-19

Authors: Lotte Felius, Hannes Mühleisen

TL;DR: DuckDB v1.4 ships database encryption capabilities. In this blog post, we dive into the implementation details of the encryption, show how to use it and demonstrate its performance implications.

> If you would like to use encryption in DuckDB, we recommend using the latest stable version, v1.4.2. For more details, see the latest release blog post.

Many years ago, we read the excellent "Code Book" by Simon Singh. Did you know that Mary, Queen of Scots, used an encryption method harking back to Julius Caesar to encrypt her more saucy letters? But alas: the cipher was broken and the contents of the letters got her executed.

These days, strong encryption software and hardware is a commodity. Modern CPUs come with specialized cryptography instructions, and operating systems small and big contain mostly-robust cryptography software like OpenSSL.

Databases store arbitrary information, it is clear that many if not most datasets of any value should perhaps not be plainly available to everyone. Even if stored on tightly controlled hardware like a cloud virtual machine, there have been many cases of files being lost through various privilege escalations. Unsurprisingly, compliance frameworks like the common SOC 2 "highly recommend" encrypting data when stored on storage mediums like hard drives.

However, database systems and encryption have a somewhat problematic track record. Even PostgreSQL, the self-proclaimed "The World's Most Advanced Open Source Relational Database" has very limited options for data encryption. SQLite, the world's "Most Widely Deployed and Used Database Engine" does not support data encryption out-of-the-box, its encryption extension is a $2000 add-on.

DuckDB has supported Parquet Modular Encryption for a while. This feature allows reading and writing Parquet files with encrypted columns. However, while Parquet files are great and reports of their impending death are greatly exaggerated, they cannot – for example – be updated in place, a pretty basic feature of a database management system.

Starting with DuckDB 1.4.0, DuckDB supports *transparent data encryption* of data-at-rest using industry-standard AES encryption.

> DuckDB's encryption does not yet meet the official NIST requirements. Please follow issue `#20162` "Store and verify tag for canary encryption" to track our progress towards NIST-compliance.

Some Basics of Encryption

There are many different ways to encrypt data, some more secure than others. In database systems and elsewhere, the standard is the Advanced Encryption Standard (AES), which is a block cipher algorithm standardized by US NIST. AES is a symmetric encryption algorithm, meaning that the *same* key is used for both encryption and decryption of data.

For this reason, most systems choose to only support *randomized* encryption, meaning that identical plaintexts will always yield different ciphertexts (if used correctly!). The most commonly used industry standard and recommended encryption algorithm is AES – Galois Counter Mode (AES-GCM). This is because on top of its ability to randomize encryption, it also *authenticates* data by calculating a tag to ensure data has not been tampered with.

DuckDB v1.4 supports encryption at rest using AES-GCM-256 and AES-CTR-256 (counter mode) ciphers. AES-CTR is a simpler and faster version of AES-GCM, but less secure, since it does not provide authentication by calculating a tag. The 256 refers to the size of the key in bits, meaning that DuckDB now only supports GCM with 32-byte keys.

GCM and CTR both require as input a (1) plaintext, (2) an initialization vector (IV) and (3) an encryption key. Plaintext is the text that a user wants to encrypt. An IV is a unique bytestream of usually 16 bytes, that ensures that identical plaintexts get encrypted into different ciphertexts. A *number used once* (nonce) is a bytestream of usually 12 bytes, that together with a 4-byte counter construct the IV. Note that the IV needs to be unique for every encrypted block, but it does not necessarily have to be random. Reuse of the same IV is problematic, since an attacker could XOR the two ciphertexts and extract both messages. The tag in AES-GCM is calculated after all blocks are encrypted, pretty much like a checksum, but it adds an integrity check that securely authenticates the entire ciphertext.

Implementation in DuckDB

Before diving deeper into how we actually implemented encryption in DuckDB, we'll explain some things about the DuckDB file format.

DuckDB has one **main database header** which stores data that enables it to correctly load and verify a DuckDB database. At the start of each DuckDB main database header, the magic bytes ("DUCKDB") are stored and read upon initialization to verify whether the file is a valid DuckDB database file. The magic bytes are followed by four 8-byte of flags that can be set for different purposes.

When a database is encrypted in DuckDB, the main database header remains plaintext at all times, since the main header contains *no sensitive data* about the contents of the database file. Upon initializing an encrypted database, DuckDB sets the first bit in the first flag to indicate that the database is encrypted. After setting this bit, additional metadata is stored that is necessary for encryption. This metadata entails the (1) *database identifier*, (2) 8 bytes of additional metadata for e.g. the encryption cipher used, and (3) the encrypted canary.

The *database identifier* is used as a "salt", and consists of 16 randomly generated bytes created upon initialization of each database. The salt is often used to ensure uniqueness, i.e., it makes sure that identical input keys or passwords are transformed into *different* derived keys. The 8-bytes of metadata comprise the key derivation function (first byte), usage of additional authenticated data (second byte), the encryption cipher (third byte), and the key length (fifth byte). After the metadata, the main header uses the encrypted canary to check if the input key is correct.

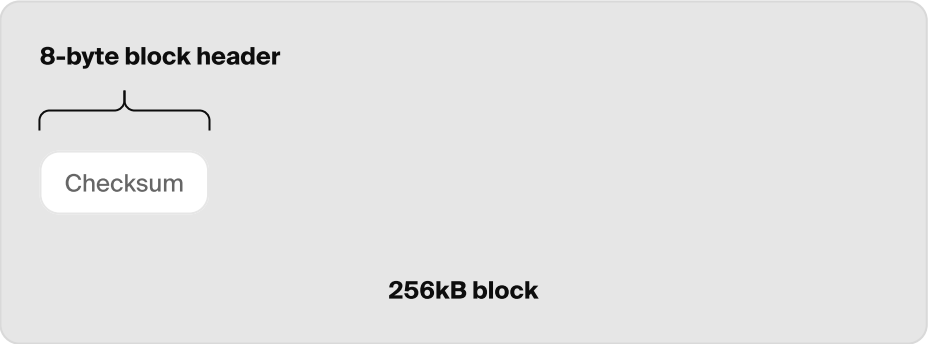
Encryption Key Management

To encrypt data in DuckDB, you can use practically *any* plaintext or base64 encoded string, but we recommend using a secure 32-byte base64 key. The user itself is responsible for the key management and thus for using a secure key. Instead of directly using the plain key provided by the user, DuckDB always derives a more secure key by means of a key derivation function (kdf). The kdf is a function that reduces or extends the input key to a 32-byte secure key. If the correctness of the input key is checked by deriving the secure key and decrypting the canary, the derived key is managed in a *secure* encryption key cache. This cache manages encryption keys for the current DuckDB context and ensures that the derived encryption keys are never swapped to disk by locking its memory. To strengthen security even more, the original input keys are immediately wiped from memory when the input keys are transformed into secure derived keys.

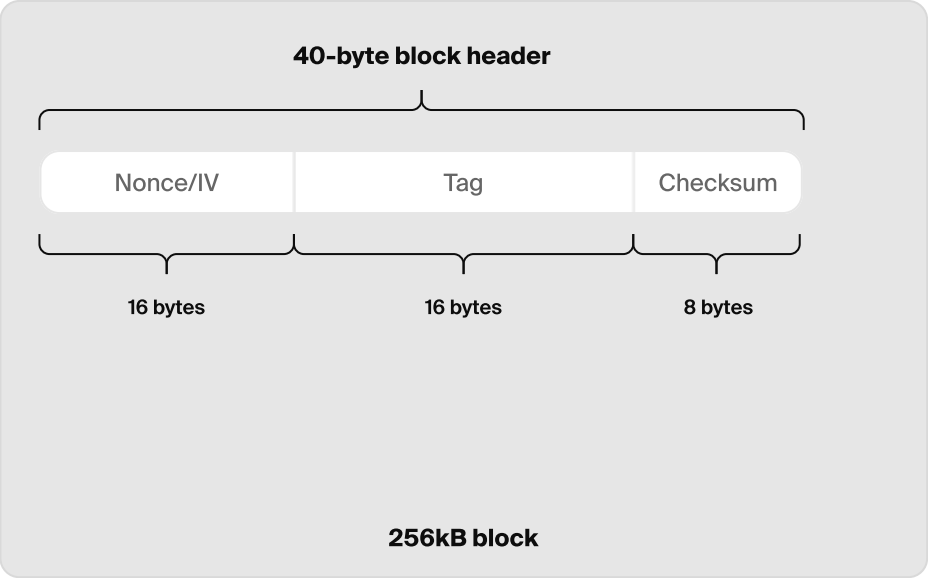
DuckDB Block Structure

After the main database header, DuckDB stores two 4KB database headers that contain more information about e.g. the block (header) size and the storage version used. After keeping the main database header plaintext, *all* remaining headers and blocks are encrypted when encryption is used.

Blocks in DuckDB are by default 256KB, but their size is configurable. At the start of each *plaintext* block there is an 8-byte block header, which stores an 8-byte checksum. The checksum is a simple calculation that is often used in database systems to check for any corrupted data.



For encrypted blocks however, its block header consists of 40 bytes instead of 8 bytes for the checksum. The block header for encrypted blocks contains a 16-byte *nonce/IV* and, optionally, a 16-byte *tag*, depending on which encryption cipher is used. The nonce and tag are stored in plaintext, but the checksum is encrypted for better security. Note that the block header always needs to be 8-bytes aligned to calculate the checksum.



Write-Ahead-Log Encryption

The write ahead log (WAL) in database systems is a crash recovery mechanism to ensure *durability*. It is an append-only file that is used in scenarios where the database crashed or is abruptly closed, and when not all changes are written yet to the main database file. The WAL makes sure these changes can be replayed up to the last checkpoint; which is a consistent snapshot of the database at a certain point in time. This means, when a checkpoint is enforced, which happens in DuckDB by either (1) closing the database or (2) reaching a certain threshold for storage, the WAL gets written into the main database file.

In DuckDB, you can force the creation of a WAL by setting

```
PRAGMA disable_checkpoint_on_shutdown;
PRAGMA wal_autocheckpoint = '1TB';
```

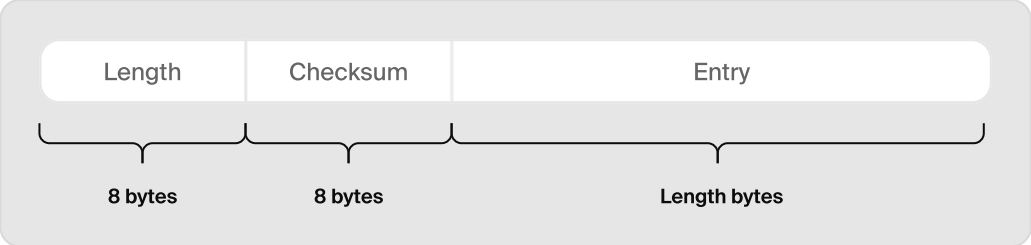
This way you'll disable a checkpointing on closing the database, meaning that the WAL does not get merged into the main database file. In addition, by setting wal_autocheckpoint to a high threshold, this will avoid intermediate checkpoints to happen and the WAL will persist. For example, we can create a persistent WAL file by first setting the above PRAGMAs, then attach an encrypted database, and then create a table where we insert 3 values.

```
ATTACH 'encrypted.db' AS enc (
    ENCRYPTION_KEY 'asdf',
    ENCRYPTION_CIPHER 'GCM'
);
CREATE TABLE enc.test (a INTEGER, b INTEGER);
```

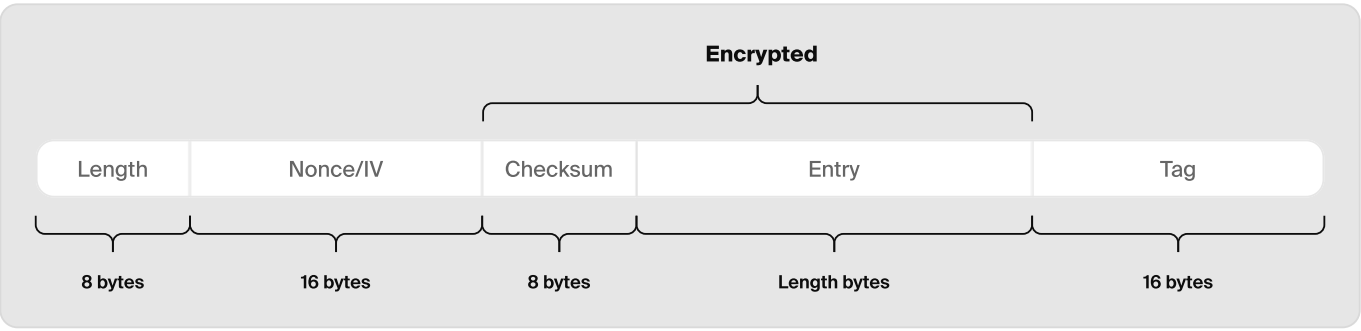
```
INSERT INTO enc.test VALUES (11, 22), (13, 22), (12, 21)
```

If we now close the DuckDB process, we can see that there is a `.wal` file shown: `encrypted.db.wal`. But how is the WAL created internally?

Before writing new entries (inserts, updates, deletes) to the database, these entries are essentially logged and appended to the WAL. Only *after* logged entries are flushed to disk, a transaction is considered as committed. A plaintext WAL entry has the following structure:



Since the WAL is append-only, we encrypt a WAL entry *per value*. For AES-GCM this means that we append a nonce and a tag to each entry. The structure in which we do this is depicted in below. When we serialize an encrypted entry to the encrypted WAL, we first store the length in plaintext, because we need to know how many bytes we should decrypt. The length is followed by a nonce, which on its turn is followed by the encrypted checksum and the encrypted entry itself. After the entry, a 16-byte tag is stored for verification.



Encrypting the WAL is triggered by default when an encryption key is given for any (un)encrypted database.

Temporary File Encryption

Temporary files are used to store intermediate data that is often necessary for large, out-of-core operations such as sorting, large joins and window functions. This data could contain sensitive information and can, in case of a crash, remain on disk. To protect this leftover data, DuckDB automatically encrypts temporary files too.

The Structure of Temporary Files There are three different types of temporary files in DuckDB: (1) temporary files that have the same layout as a regular 256KB block, (2) compressed temporary files and (3) temporary files that exceed the standard 256KB block size. The former two are suffixed with .tmp, while the latter is distinguished by a suffix with .block. To keep track of the size of .block temporary files, they are always prefixed with its length. As opposed to regular database blocks, temporary files do not contain a checksum to check for data corruption, since the calculation of a checksum is somewhat expensive.

Encrypting Temporary Files Temporary files are encrypted (1) **automatically** when you attach an encrypted database or (2) when you use the setting `SET temp_file_encryption = true`{:.language-sql .highlight}. In the latter case, the main database file is plaintext, but the temporary files will be encrypted. For the encryption of temporary files DuckDB internally generates *temporary keys.* This means that when the database crashes, the temporary keys are also lost. Temporary files cannot be decrypted in this case and are then essentially garbage.

To force DuckDB to produce temporary files, you can use a simple trick by just setting the memory limit low. This will create temporary files once the memory limit is exceeded. For example, we can create a new encrypted database, load this database with TPC-H data (SF 1), and then set the memory limit to 1 GB. If we then perform a large join, we force DuckDB to spill intermediate data to disk. For example:

```
SET memory_limit = '1GB';
ATTACH 'tpch_encrypted.db' AS enc (
    ENCRYPTION_KEY 'asdf',
    ENCRYPTION_CIPHER 'cipher'
);
USE enc;
CALL dbgen(sf = 1);

ALTER TABLE lineitem
    RENAME TO lineitem1;
CREATE TABLE lineitem2 AS
    FROM lineitem1;
CREATE OR REPLACE TABLE ans AS
    SELECT l1.* , l2.*
    FROM lineitem1 l1
    JOIN lineitem2 l2 USING (l_orderkey , l_linenumber);
```

This sequence of commands will result in encrypted temporary files being written to disk. Once the query completes or when the DuckDB shell is exited, the temporary files are automatically cleaned up. In case of a crash however, it may happen that temporary files will be left on disk and need to be cleaned up manually.

How to Use Encryption in DuckDB

In DuckDB, you can (1) encrypt an existing database, (2) initialize a new, empty encrypted database or (3) reencrypt a database. For example, let's create a new database, load this database with TPC-H data of scale factor 1 and then encrypt this database.

```
INSTALL tpch;
LOAD tpch;
ATTACH 'encrypted.duckdb' AS encrypted (ENCRYPTION_KEY 'asdf');
ATTACH 'unencrypted.duckdb' AS unencrypted;
USE unencrypted;
CALL dbgen(sf = 1);
COPY FROM DATABASE unencrypted TO encrypted;
```

There is not a trivial way to prove that a database is encrypted, but correctly encrypted data should look like random noise and has a high entropy. So, to check whether a database is actually encrypted, we can use tools to calculate the entropy or visualize the binary, such as ent and binocle.

When we use ent after executing the above chunk of SQL, i.e., `ent encrypted.duckdb`, this will result in an entropy of 7.99999 bits per byte. If we do the same for the plaintext (unencrypted) database, this results in 7.65876 bits per byte. Note that the plaintext database also has a high entropy, but this is due to compression.

Let's now visualize both the plaintext and encrypted data with binocle. For the visualization we created both a plaintext DuckDB database with scale factor of 0.001 of TPC-H data and an encrypted one:

In these figures, we can clearly observe that the encrypted database file seems completely random, while the plaintext database file shows some clear structure in its binary data.

To decrypt an encrypted database, we can use the following SQL:

```
ATTACH 'encrypted.duckdb' AS encrypted (ENCRYPTION_KEY 'asdf');
ATTACH 'new_unencrypted.duckdb' AS unencrypted;
COPY FROM DATABASE encrypted TO unencrypted;
```

And to reencrypt an existing database, we can just simply copy the old encrypted database to a new one, like:

```
ATTACH 'encrypted.duckdb' AS encrypted (ENCRYPTION_KEY 'asdf');
ATTACH 'new_encrypted.duckdb' AS new_encrypted (ENCRYPTION_KEY 'xxxx');
COPY FROM DATABASE encrypted TO new_encrypted;
```

The default encryption algorithm is AES GCM. This is recommended since it also authenticates data by calculating a tag. Depending on the use case, you can also use AES CTR. This is faster than AES GCM since it skips calculating a tag after encrypting all data. You can specify the CTR cipher as follows:

```
ATTACH 'encrypted.duckdb' AS encrypted (
    ENCRYPTION_KEY 'asdf',
    ENCRYPTION_CIPHER 'CTR'
);
```

To keep track of which databases are encrypted, you can query this by running:

```
FROM duckdb_databases();
```

This will show which databases are encrypted, and which cipher is used:

database_name	database_oid	path	…	encrypted	cipher
encrypted	2103	encrypted.duckdb	…	true	GCM
unencrypted	2050	unencrypted.duckdb	…	false	NULL
memory	592	NULL	…	false	NULL
system	0	NULL	…	false	NULL
temp	1995	NULL	…	false	NULL

5 rows — 10 columns (5 shown)

Implementation and Performance

Here at DuckDB, we strive to achieve a good out-of-the-box experience with zero external dependencies and a small footprint. Encryption and decryption, however, are usually performed by pretty heavy external libraries such as OpenSSL. We would much prefer not to rely on external libraries or statically linking huge codebases just so that people can use encryption in DuckDB without additional steps. This is why we actually implemented encryption *twice* in DuckDB, once with the (excellent) Mbed TLS library and once with the ubiquitous OpenSSL library.

DuckDB already shipped parts of Mbed TLS because we use it to verify RSA extension signatures. However, for maximum compatibility we actually disabled the hardware acceleration of MbedTLS, which has a performance impact. Furthermore, Mbed TLS is not particularly hardened against things like nasty timing attacks. OpenSSL on the other hand contains heavily vetted and hardware-accelerated code to perform AES operations, which is why we can also use it for encryption.

In DuckDB Land, OpenSSL is part of the `httpfs` extension. Once you load that extension, encryption will *automatically* switch to using OpenSSL. After we shipped encryption in DuckDB 1.4.0, security experts actually found issues with the random number generator we used in Mbed TLS mode. Even though it would be difficult to actually exploit this, we *disabled writing* to databases in MbedTLS mode from DuckDB 1.4.1. Instead, DuckDB now (version 1.4.2+) tries to auto-install and auto-load the `httpfs` extension whenever a write is attempted. We might be able to revisit this in the future, but for now this seems the safest path forward that still allows high compatibility for reading. In OpenSSL mode, we always used a cryptographically-safe random number generation so that mode is unaffected.

Encrypting and decrypting database files is an additional step in writing tables to disk, so we would naturally assume that there is some performance impact. Let's investigate the performance impact of DuckDB's new encryption feature with a very basic experiment.

We first create two DuckDB database files, one encrypted and one unencrypted. We use the TPC-H benchmark generator again to create the table data, particularly the (somewhat tired) `lineitem` table.

```
INSTALL httpfs;
INSTALL tpch;
LOAD tpch;

ATTACH 'unencrypted.duckdb' AS unencrypted;
CALL dbgen(sf = 10, catalog = 'unencrypted');

ATTACH 'encrypted.duckdb' AS encrypted (ENCRYPTION_KEY 'asdf');
CREATE TABLE encrypted.lineitem AS FROM unencrypted.lineitem;
```

Now we use DuckDB's neat SUMMARIZE command three times: once on the unencrypted database, and once on the encrypted database using MbedTLS and once on the encrypted database using OpenSSL. We set a very low memory limit to force more reading and writing from disk.

```
SET memory_limit = '200MB';
.timer on

SUMMARIZE unencrypted.lineitem;
SUMMARIZE encrypted.lineitem;

LOAD httpfs; -- use OpenSSL
SUMMARIZE encrypted.lineitem;
```

Here are the results on a fairly recent MacBook: SUMMARIZE on the unencrypted table took ca. 5.4 seconds. Using Mbed TLS, this went up to around 6.2 s. However, when enabling OpenSSL the end-to-end time went straight back to 5.4 s. How is this possible? Is decryption not expensive? Well, there is a lot more happening in query processing than reading blocks from storage. So the impact of decryption is not all that huge, even when using a slow implementation. Secondly, when using hardware acceleration in OpenSSL, the overall overhead of encryption and decryption becomes almost negligible.

But just running summarization is overly simplistic. Real™ database workloads include modifications to data, insertion of new rows, updates of rows, deletion of rows etc. Also, multiple clients will be updating and querying at the same time. So we re-surrected the full TPC-H "Power" test from our previous blog post "Changing Data with Confidence and ACID". We slightly tweaked the benchmark script to enable the new database encryption. For this experiment, we used the OpenSSL encryption implementation due to the issues outlined above. We observe "Power@Size" and "Throughput@Size". The former is raw sequential query performance, while the latter measures multiple parallel query streams in the presence of updates.

When running on the same MacBook with DuckDB 1.4.1 and a "scale factor" of 100, we get a Power@Size metric of 624,296 and a Throughput@Size metric of 450,409 *without* encryption.

When we enable encryption, the results are almost unchanged, confirming the observation of the small microbenchmark above. However, the relationship between available memory and the benchmark size means that we're not stressing temporary file encryption. So we reran everything with an 8GB memory limit. We confirmed constant reading and writing to and from disk in this configuration by observing operating system statistics. For the unencrypted case, the Power@Size metric predictably went down to 591,841 and Throughput@Size went down to 153,690. And finally, we could observe a slight performance decrease with Power@Size of 571,985 and Throughput@Size of 145,353. However, that difference is not very great either and likely not relevant in real operational scenarios.

Conclusion

With the new encrypted database feature, we can now safely pass around DuckDB database files with all information inside them completely opaque to prying eyes. This allows for some interesting new deployment models for DuckDB, for example, we could now put an encrypted DuckDB database file on a Content Delivery Network (CDN). A fleet of DuckDB instances could attach to this file read-only using the decryption key. This elegantly allows efficient distribution of private background data in a similar way like encrypted Parquet files, but of course with many more features like multi-table storage. When using DuckDB with encrypted storage, we can also simplify threat modeling when – for example – using DuckDB on cloud providers. While in the past access to DuckDB storage would have been enough to leak data, we can now relax paranoia regarding storage a little, especially since temporary files and WAL are also encrypted. And the best part of all of this, there is almost no performance overhead to using encryption in DuckDB, especially with the OpenSSL implementation.

We are very much looking forward to what you are going to do with this feature, and please let us know if you run into any issues.

Writes in DuckDB-Iceberg

Publication date: 2025-11-28

Author: Tom Ebergen

TL;DR: We shipped a number of features and improvements to the DuckDB-Iceberg extension: insert, update, and delete statements are all supported now.

Over the past several months, the DuckLabs team has been hard at work on the DuckDB-Iceberg extension, with *full read support* and *initial write support* released in v1.4.0. Today, we are happy to announce delete and update support for Iceberg v2 tables is available in v1.4.2!

The Iceberg open table format has become extremely popular in the past two years, with many databases announcing support for the open table format originally developed at Netflix. This past year the DuckDB team has made Iceberg integration a priority and today we are happy to announce another step in that direction. In this blog post we will describe the current feature set of DuckDB-Iceberg in DuckDB v1.4.2.

Getting Started

To experiment with the new DuckDB-Iceberg features, you will need to connect to your favorite Iceberg REST Catalog. There are many ways to connect to an Iceberg REST Catalog: please have a look at the Connecting to REST Catalogs for connecting to catalogs like Apache Polaris or Lakekeeper and the Connecting to S3 Tables page if you would like to connect to Amazon S3 Tables.

```
ATTACH '<warehouse_name>' AS iceberg_catalog (
    TYPE iceberg,
    <other options>
);
```

Inserts, Deletes and Updates

Support for creating tables and inserting to tables was already added in DuckDB v1.4.0: you can use standard DuckDB SQL syntax to insert data into your Iceberg table.

```
CREATE TABLE iceberg_catalog.default.simple_table (
    col1 INTEGER,
    col2 VARCHAR
);
INSERT INTO iceberg_catalog.default.simple_table
    VALUES (1, 'hello'), (2, 'world'), (3, 'duckdb is great');
```

You can also use any DuckDB table scan function to insert data into an Iceberg table:

```
INSERT INTO iceberg_catalog.default.more_data
    SELECT * FROM read_parquet('path/to/parquet');
```

Starting with v1.4.2, the standard SQL syntax also works for deletes and updates:

```
DELETE FROM iceberg_catalog.default.simple_table
WHERE col1 = 2;

UPDATE iceberg_catalog.default.simple_table
SET col1 = col1 + 5
WHERE col1 = 1;

SELECT *
FROM iceberg_catalog.default.simple_table;
```

col1 int32	col2 varchar
3	duckdb is great
6	hello

> The Iceberg write support current has two limitations:
>
> The update support is limited to *tables that are not partitioned and not sorted.* Attempting to perform update, insert or delete operations on partitioned or sorted tables using DuckDB-Iceberg will result in an error.
>
> DuckDB-Iceberg only writes positional deletes for `DELETE` and `UPDATE` statements. Copy-on-write functionality is not yet supported.

Functions for Table Properties

Currently, DuckDB-Iceberg only supports *merge-on-read semantics*. Within Iceberg Table Metadata, table properties can be used to describe what form of deletes or updates are allowed. DuckDB-Iceberg will respect `write.update.mode` and `write.delete.mode` table properties for updates and deletes. If a table has these properties and they are not `merge-on-read`, DuckDB will throw an error and the `UPDATE` or `DELETE` will not be committed. Version v1.4.2 introduces three new functions to add, remove, and view table properties for an Iceberg table:

- `set_iceberg_table_properties`
- `iceberg_table_properties`
- `remove_iceberg_table_properties`

You can use them as follows:

```
-- to set table properties
CALL set_iceberg_table_properties(iceberg_catalog.default.simple_table, {
    'write.update.mode': 'merge-on-read',
    'write.file.size': '100000kb'
});
-- to read table properties
SELECT * FROM iceberg_table_properties(iceberg_catalog.default.simple_table);
```

key varchar	value varchar
write.update.mode	merge-on-read
write.file.size	100000kb

```
-- to remove table properties
CALL remove_iceberg_table_properties(
    iceberg_catalog.default.simple_table,
    ['some.other.property']
);
```

Iceberg Table Metadata

DuckDB-Iceberg also allows you to view the metadata of your Iceberg tables using the `iceberg_metadata()` and `iceberg_snapshots()` functions.

```
SELECT * FROM iceberg_metadata(iceberg_catalog.default.table_1);
```

manifest_path	manifest_sequence_…	manifest_content	status	content	file_path	file_format	record_count
varchar	int64	varchar	varchar	varchar	varchar	varchar	int64
s3://warehouse/def…	1	DATA	ADDED	EXISTING	s3://<storage_location>/simple_table/data/019a6ecc-9e9e-7…	parquet	3
s3://warehouse/def…	2	DELETE	ADDED	POSITION_DELETES	s3://<storage_location>/simple_table/data/d65b1db8-9fa8-4…	parquet	1
s3://warehouse/def…	3	DELETE	ADDED	POSITION_DELETES	s3://<storage_location>/simple_table/data/8d1b92dc-5f6e-4…	parquet	1
s3://warehouse/def…	3	DATA	ADDED	EXISTING	s3://<storage_location>/simple_table/data/019a6ecf-5261-7…	parquet	1

```sql
SELECT * FROM iceberg_snapshots(iceberg_catalog.default.simple_table);
```

sequence_number	snapshot_id	timestamp_ms	manifest_list
uint64	uint64	timestamp	varchar
1	1790528822676766947	2025-11-10 17:24:55.075	s3://<storage_location>/simple_table/data/snap-1790528822676766947-f09658c4-ca52-4305-943f-6a8073529fef.avro
2	6333537230056014119	2025-11-10 17:27:35.602	s3://<storage_location>/simple_table/data/snap-6333537230056014119-316d09bc-549d-46bc-ae13-a9fab5cbf09b.avro
3	7452040077415501383	2025-11-10 17:27:52.169	s3://<storage_location>/simple_table/data/snap-7452040077415501383-93dee94e-9ec1-45fa-aec2-13ef434e50eb.avro

Time Travel

Time travel is also possible via snapshot ids or timestamps using the `AT (VERSION => ...)` or `AT (TIMESTAMP => ...)` syntax.

```sql
-- via snapshot id
SELECT *
FROM iceberg_catalog.default.simple_table AT (
    VERSION => <snapshot_id>
);
```

col1	col2
int32	varchar
1	hello
3	duckdb is great

```sql
-- via timestamp
SELECT *
FROM iceberg_catalog.default.simple_table AT (
    TIMESTAMP => '2025-11-10 17:27:45.602'
);
```

col1	col2
int32	varchar
1	hello
3	duckdb is great

Viewing Requests to the Iceberg REST Catalog

You may also be curious as to what requests DuckDB is making to the Iceberg REST Catalog. To do so, enable HTTP logging, run your workload, then select from the HTTP logs.

```sql
CALL enable_logging('HTTP');
SELECT * FROM iceberg_catalog.default.simple_table;
SELECT request.type, request.url, response.status
FROM duckdb_logs_parsed('HTTP');
```

```
┌─────────┬──────────────────────────────────────────────────────────────────────────────────────────┬──────────────────┐
│  type   │                                           url                                            │      status      │
│ varchar │                                         varchar                                          │     varchar      │
├─────────┼──────────────────────────────────────────────────────────────────────────────────────────┼──────────────────┤
│ GET     │ https://<catalog_endpoint>/iceberg/v1/<warehouse>/iceberg-testing/namespaces/default      │ NULL             │
│ HEAD    │ https://<catalog_endpoint>/iceberg/v1/<warehouse>/iceberg-testing/namespaces/default/tables/simple_table │ NULL │
│ GET     │ https://<catalog_endpoint>/iceberg/v1/<warehouse>/iceberg-testing/namespaces/default/tables/simple_table │ NULL │
│ GET     │ https://<storage_endpoint>/data/snap-5943683398986255948-c2217dde-6036-4e07-88f2-…        │ OK_200           │
│ GET     │ https://<storage_endpoint>/data/f8c95b93-7b6b-4a24-8557-b98b553723d4-m0.avro             │ OK_200           │
│ GET     │ https://<storage_endpoint>/data/214a7988-da39-4dac-aa3a-4a73d3ead405-m0.avro             │ OK_200           │
│ GET     │ https://<storage_endpoint>/data/019a7244-c6e8-7bc9-9dd4-7249fcb04959.parquet             │ PartialContent_206 │
│ GET     │ https://<storage_endpoint>/data/019a7244-fcb5-7308-96ec-1c9e32509eab.parquet             │ PartialContent_206 │
│ GET     │ https://<storage_endpoint>/data/7f14bb06-f57a-42b4-ba7f-053a65152759-m0.avro             │ OK_200           │
│ GET     │ https://<storage_endpoint>/data/71f8b43d-51e7-40e7-be88-e8d869836ecd-deletes.parq…       │ PartialContent_206 │
│ GET     │ https://<storage_endpoint>/data/64f6c6e2-2f54-470e-b990-b201bc615042-m0.avro             │ OK_200           │
│ GET     │ https://<storage_endpoint>/data/4e54afed-6dd8-4ba0-88fb-16f972ac1d91-deletes.parq…       │ PartialContent_206 │
├─────────┴──────────────────────────────────────────────────────────────────────────────────────────┴──────────────────┤
│ 12 rows                                                                                                   3 columns │
└──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

Here we can see calls to the Iceberg REST Catalog, followed by calls to the storage endpoint. The first three calls to the Iceberg REST Catalog are to verify the schema still exists and to get the latest `metadata.json` of the DuckDB-Iceberg table. Next, it queries the manifest list, manifest files, and eventually the files with data and deletes. The data and delete files are stored locally in a cache to speed up subsequent reads.

Transactions

DuckDB is an ACID-compliant database that supports transactions. Work on DuckDB-Iceberg has been made with this in mind. Within a transaction, the following conditions will hold for Iceberg tables.

1. The first time a table is read in a transaction, its snapshot information is stored in the transaction and will remain consistent within that transaction.

2. Updates, inserts and deletes will only be committed to an Iceberg Table when the transaction is committed (i.e., `COMMIT`);

Point #1 is important for read performance. If you wish to do analytics on an Iceberg table and you do not need to get the latest version of the table every time, running your analytics in a transaction will prevent fetching the latest version for every query.

```sql
-- truncate the logs
CALL truncate_duckdb_logs();
CALL enable_logging('HTTP')
BEGIN;
-- first read gets latest snapshot information
SELECT * FROM iceberg_catalog.default.simple_table;
-- subsequent read reads from local cached data
SELECT * FROM iceberg_catalog.default.simple_table;
-- get logs
SELECT request.type, request.url, response.status
FROM duckdb_logs_parsed('HTTP');
```

```
┌─────────┬──────────────────────────────────────────────────────────────────────
│  type   │                                  url
│      status       │
│ varchar │                                varchar
│     varchar       │
├─────────┼──────────────────────────────────────────────────────────────────────
│ GET     │ https://<catalog_endpoint>/iceberg/v1/<warehouse>/iceberg-testing/namespaces/default
│ NULL              │
│ HEAD    │ https://<catalog_
endpoint>/iceberg/v1/<warehouse>/iceberg-testing/namespaces/default/tables/simple_table      │ NULL
│
│ GET     │ https://<catalog_
endpoint>/iceberg/v1/<warehouse>/iceberg-testing/namespaces/default/tables/simple_table      │ NULL
│
│ GET     │ https://<storage_endpoint>/data/snap-5943683398986255948-c2217dde-6036-4e07-88f2-1…
│ OK_200            │
│ GET     │ https://<storage_endpoint>/data/f8c95b93-7b6b-4a24-8557-b98b553723d4-m0.avro
│ OK_200            │
│ GET     │ https://<storage_endpoint>/data/214a7988-da39-4dac-aa3a-4a73d3ead405-m0.avro
│ OK_200            │
│ GET     │ https://<storage_endpoint>/data/019a7244-c6e8-7bc9-9dd4-7249fcb04959.parquet
│ PartialContent_206 │
│ GET     │ https://<storage_endpoint>/data/019a7244-fcb5-7308-96ec-1c9e32509eab.parquet
│ PartialContent_206 │
│ GET     │ https://<storage_endpoint>/data/7f14bb06-f57a-42b4-ba7f-053a65152759-m0.avro
│ OK_200            │
│ GET     │ https://<storage_endpoint>/data/71f8b43d-51e7-40e7-be88-e8d869836ecd-deletes.parquet
│ PartialContent_206 │
│ GET     │ https://<storage_endpoint>/data/64f6c6e2-2f54-470e-b990-b201bc615042-m0.avro
│ OK_200            │
│ GET     │ https://<storage_endpoint>/data/4e54afed-6dd8-4ba0-88fb-16f972ac1d91-deletes.parquet
│ PartialContent_206 │
├─────────┴──────────────────────────────────────────────────────────────────────
│ 12 rows
3 columns │
└───────────────────────────────────────────────────────────────────────────────
```

Here we see all the same requests we saw in the previous section. However, now we are in a transaction, which means the second time we read from `iceberg_catalog.default.simple_table`, we do not need to query the REST Catalog for table updates. This means DuckDB-Iceberg performs no extra requests when reading a table a second time, significantly improving performance.

Conclusion and Future Work

With these features, DuckDB-Iceberg now has a strong base support for the Iceberg tables, which enables users to unlock the analytical powers of DuckDB on their Iceberg tables. There is still more work to come and the Iceberg table specification has many more features the DuckDB team would like to support in DuckDB-Iceberg. If you feel any feature is a priority for your analytical workloads, please reach out to us in the DuckDB-Iceberg GitHub repository or get in touch with our engineers.

Below is a list of improvements planned for the near future (in no particular order):

- Performance improvements
- Updates / deletes / inserts to partitioned tables
- Updates / deletes / inserts to sorted tables
- Schema evolution
- Support for Iceberg v3 tables, focusing on binary deletion vectors and row lineage tracking

Announcing DuckDB 1.4.3 LTS

Publication date: 2025-12-09

Author: The DuckDB team

TL;DR: Today we are releasing DuckDB 1.4.3. Along with bugfixes, we are shipping native extensions and Python support for Windows Arm64.

In this blog post, we highlight a few important fixes in DuckDB v1.4.3, the third patch release in DuckDB's 1.4 LTS line. You can find the complete release notes on GitHub.

To install the new version, please visit the installation page.

Fixes

This version ships a number of performance improvements and bugfixes.

Correctness

- #18782 Incorrect "rows affected" was reported by ART index
- #19313 Wrong result in corner case: a HAVING clause without a GROUP BY returned an incorrect result
- #19517 JOIN with a LIKE pattern resulted in columns being incorrectly included
- #19924 The optimizer incorrectly removed the ORDER BY from aggregates
- #19970 Fixed updates on indexed tables with DICT_FSST compression
- #20009 Fixed updates with DICT_FSST compression

Crashes and Internal Errors

- #19469 Potential error occurred in constraint violation message when checking foreign key constraints
- #19754 Race condition could trigger a segfault in the encryption key cache
- #20044 Fixed edge case in index deletion code path

Performance

- #18997 Macro binding had slow performance for unbalanced trees
- #19901 Memory management has been improved during WAL replay in the presence of indexes
- The vortex extension ships significant performance improvements for writing Vortex files

Miscellaneous

- #19575 Invalid Unicode error with LIKE expressions
- #19916 The default time zone of DuckDB-Wasm had an offset inverted from what it should be
- #19884 Copying to Parquet with a prepared statement did not work

Azure Blob Storage Writes

The `azure` extension can now write to the Azure Blob Storage. This unlocks several other Azure and Fabric features, including using OneLake instances.

Windows Arm64

With this release, we are introducing beta support for Windows Arm64 by distributing native DuckDB extensions and Python wheels.

Extension Distribution

On Windows Arm64, you can now natively install core extensions, including complex ones like `spatial`:

```
duckdb
```

```
PRAGMA platform;
```

```
┌─────────────────┐
│    platform     │
│     varchar     │
├─────────────────┤
│ windows_arm64   │
└─────────────────┘
```

```
INSTALL spatial;
LOAD spatial;
SELECT ST_Area(ST_GeomFromText(
        'POLYGON((0 0, 4 0, 4 3, 0 3, 0 0))'
    )) AS area;
```

```
┌─────────┐
│  area   │
│ double  │
├─────────┤
│  12.0   │
└─────────┘
```

Python Wheel Distribution

We now distribute Python wheels for Windows Arm64 for Python 3.11+. This means that you take e.g. a Copilot+ PC, install the native Python interpreter and run:

```
pip install duckdb
```

This installs the `duckdb` package using the binary distributed through PyPI. Then, you can use it as follows:

```
python
```

```
Python 3.13.9
    (tags/v3.13.9:8183fa5, Oct 14 2025, 14:51:39)
    [MSC v.1944 64 bit (ARM64)] on win32

>>> import duckdb
>>> duckdb.__version__
'1.4.3'
```

> Currently, many Python installations that you'll find on Windows Arm64 computers use the x86_64 (AMD64) Python distribution and run through Microsoft's Prism emulator. For example, if you install Python through the Windows Store, you will get the Python AMD64 installation. To understand which platform your Python installation is using, observe the Python CLI's first line (e.g., `Python`

```
3.13.9 ... (ARM64)).
```

ODBC Driver

We are now shipping a native ODBC driver for Windows Arm64. Head to the ODBC Windows installation page to try it out!

Conclusion

This post was a short summary of the changes in v1.4.3. As usual, you can find the full release notes on GitHub. We would like to thank our contributors for providing detailed issue reports and patches. Stay tuned for DuckDB v1.4.4 and v1.5.0, both released early next year!

Iceberg in the Browser

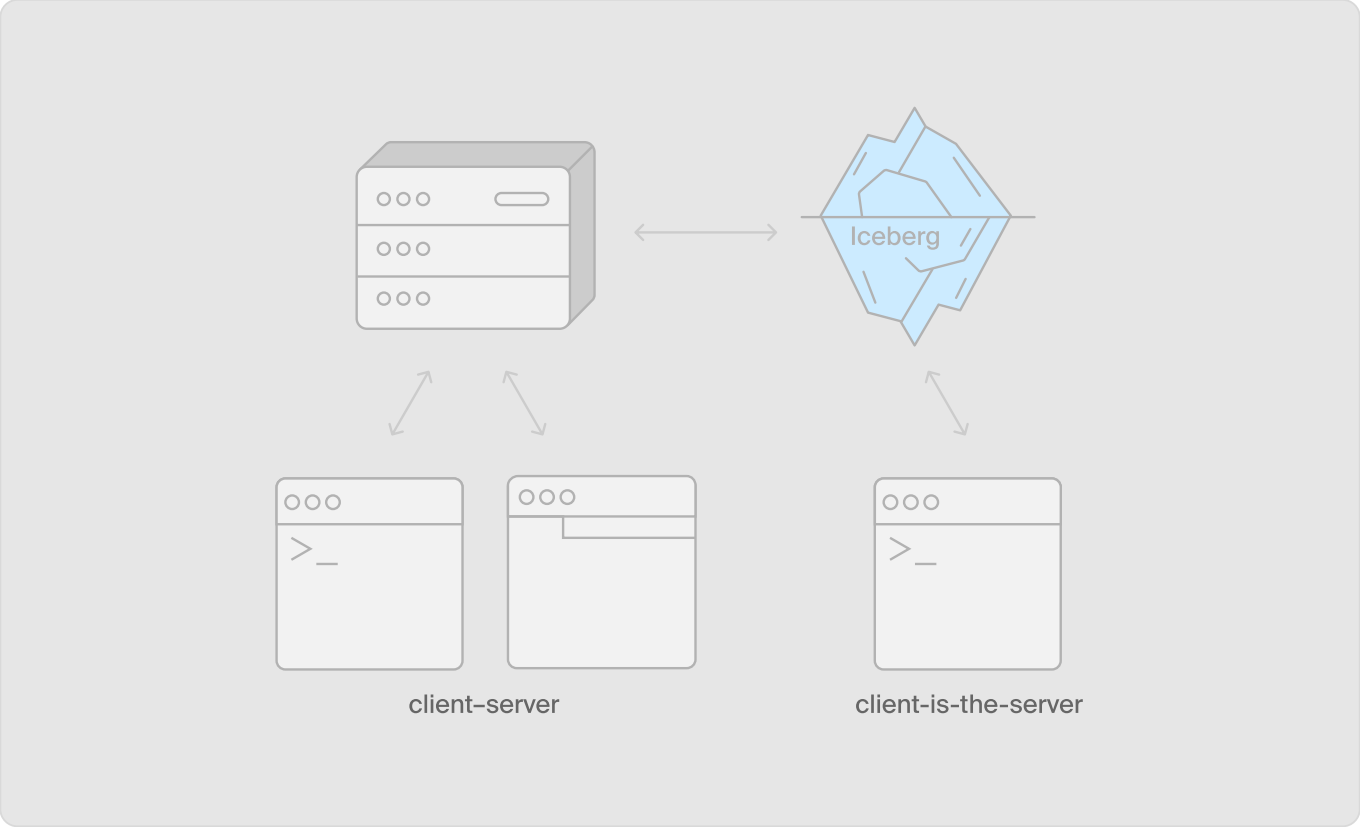
Publication date: 2025-12-16

Authors: Carlo Piovesan, Tom Ebergen, Gábor Szárnyas

TL;DR: DuckDB is the first end-to-end interface to Iceberg REST Catalogs within a browser tab. You can now read and write tables in Iceberg catalogs without needing to manage any infrastructure – directly from your browser!

In this post, we describe the current patterns for interacting with Iceberg Catalogs, and pose the question: could it be done from a browser? After elaborating on the DuckDB ecosystem changes required to unlock this capability, we demonstrate our approach to interacting with an Iceberg REST Catalog. It's browser-only, no extra setup required.

Interaction Models for Iceberg Catalogs



{: .lightmode-img }

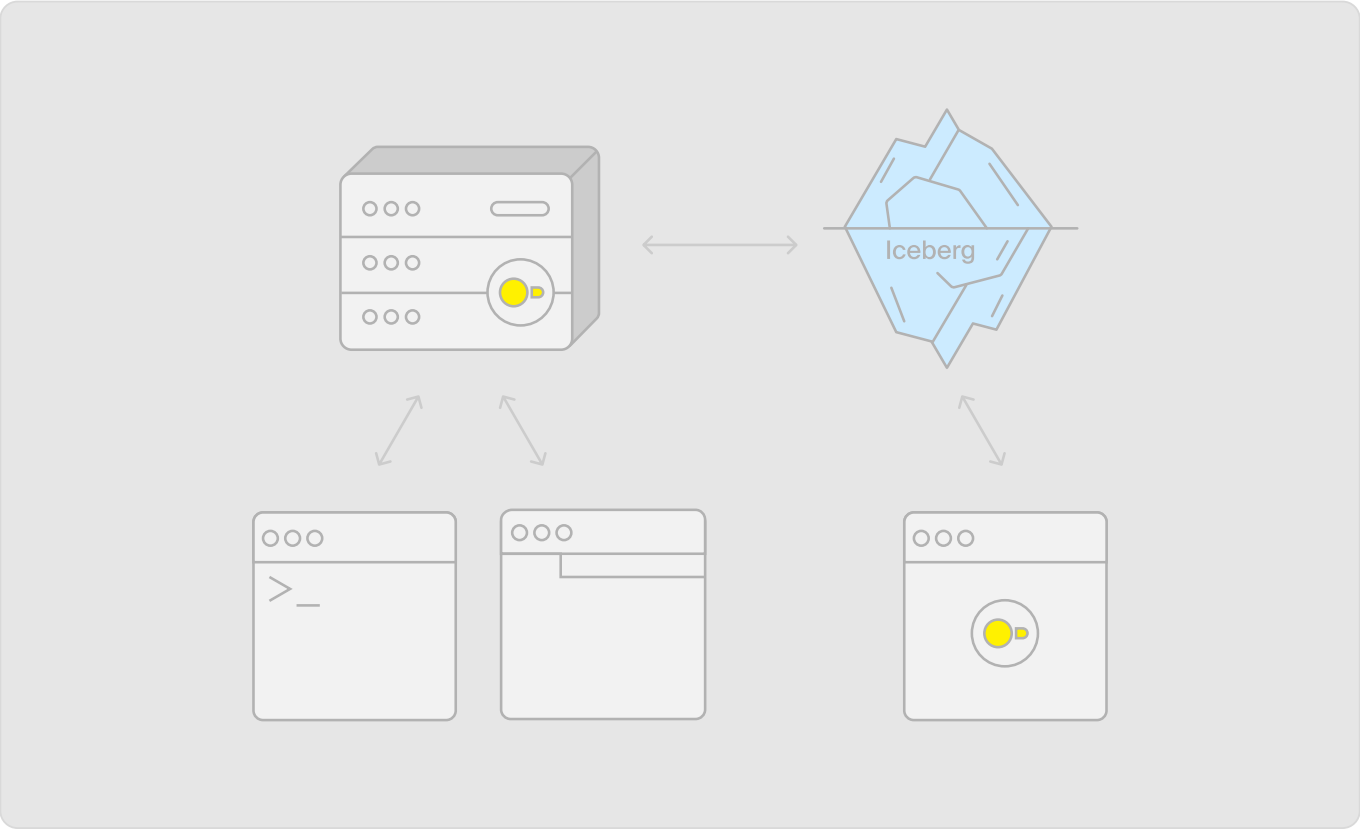
Iceberg is an *open table format,* which allows you to capture a mutable database table as a set of static files on object storage (such as AWS S3). *Iceberg catalogs* allow you to track and organize Iceberg tables. For example, Iceberg REST Catalogs provide these functionalities through a REST API.

There are two common ways to interact with Iceberg catalogs:

- The *client–server model,* where the compute part of the operation is delegated to a managed infrastructure (such as the cloud). Users can interact with the server by installing a local client or using a lightweight client such as a browser.
- The *client-is-the-server model,* where the user first installs the relevant libraries, and then performs queries directly on their machine.

Iceberg engines follow these interaction models: they are either run natively in managed compute infrastructure or they are run locally by the user. Let's see how things look with DuckDB in the mix!

Iceberg with DuckDB



{: .lightmode-img }

DuckDB supports both Iceberg interaction models. In the *client–server model,* DuckDB runs on the server to read the Iceberg datasets. From the user's point of view, the choice of engine is transparent, and DuckDB is just one of many engines that the server could use in the background. The *client-is-the-server* model is more interesting: here, users install a DuckDB client locally and use it through its SQL interface to query Iceberg catalogs. For example:

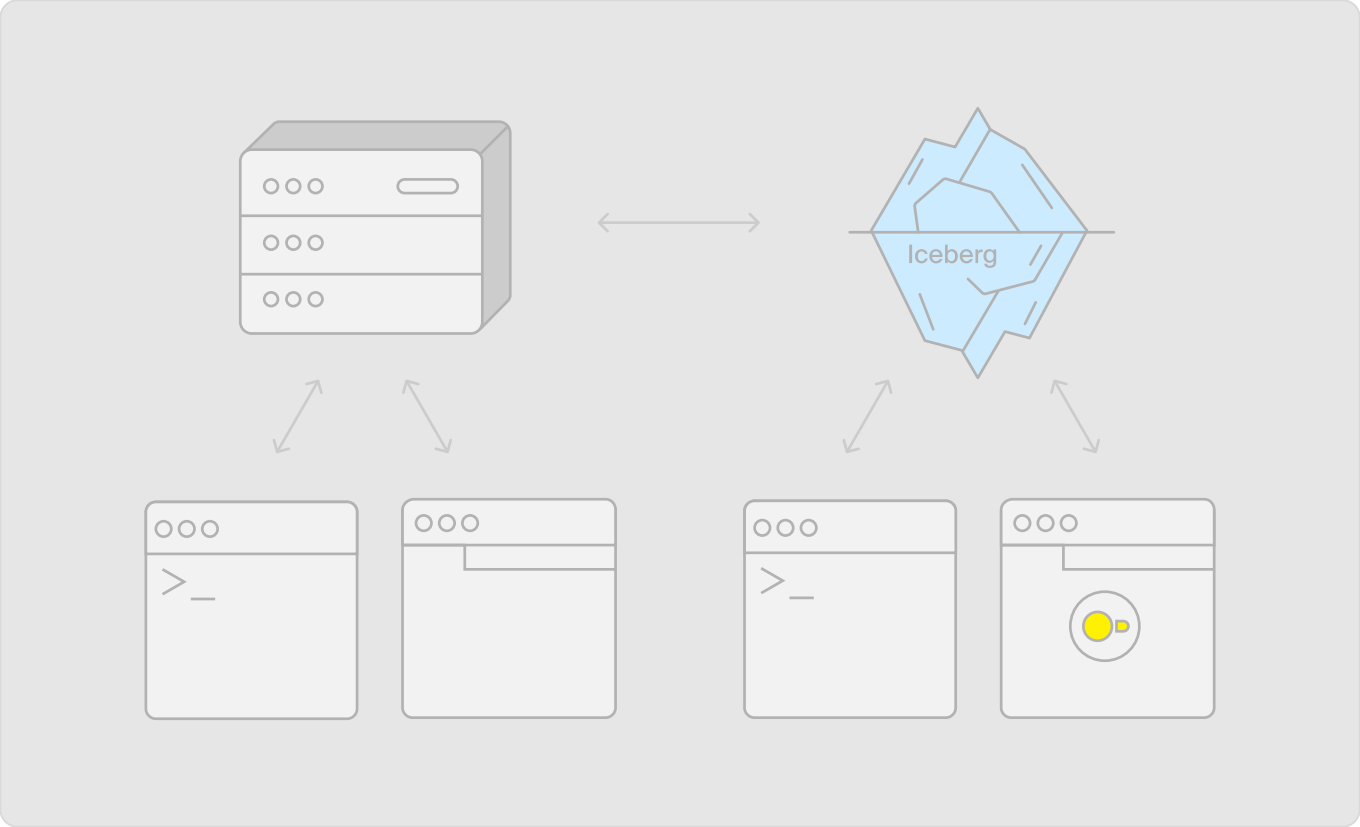
```
CREATE SECRET test_secret (
    TYPE S3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>'
);

ATTACH '<warehouse>' AS db (
    TYPE iceberg,
    ENDPOINT_URL '<https://your-iceberg-endpoint>',
);

SELECT sum(value)
FROM db.table
WHERE other_column = '<some_value>';
```

The client-is-the-server model unlocks empowered clients, which can operate directly on the data.

> You can discover the full DuckDB-Iceberg extension feature set, including insert and update capabilities, in our earlier blog post.

Iceberg with DuckDB in the Browser

While setting up a local DuckDB installation is quite simple, opening a browser tab is even quicker. Therefore, we asked ourselves: could we support the *client-is-the-server* model directly from within a browser tab? This could provide a zero-setup, no-infrastructure, properly serverless option for interacting with Iceberg catalogs.



{:.lightmode-img}

Luckily, DuckDB has a client that can run in any browser! DuckDB-Wasm is a WebAssembly port of DuckDB, which supports loading of extensions.

Interacting with an Iceberg REST Catalog requires a number of functionalities; the ability to talk to a REST API over HTTP(S), the ability to read and write `avro` and `parquet` files on object storage, and finally, the ability to negotiate authentication to access those resources on behalf of the user. All of these must be done from within a browser without calling any native components.

To support these functionalities, we implemented the following high-level changes:

- In the core `duckdb` codebase, we redesigned HTTP interactions, so that extensions and clients have a uniform interface to the networking stack. (PR)
- In `duckdb-wasm`, we implemented such an interface, which in this case is a wrapper around the available JavaScript network stack. (PR)
- In `duckdb-iceberg`, we routed all networking through the common HTTP interface, so that native DuckDB and DuckDB-Wasm execute the same logic. (PR)

The result is that you can now query Iceberg with DuckDB running directly in a browser! Now you can access the same Iceberg catalog using *client–server*, *client-is-the-server*, or properly serverless from the isolation of a browser tab!

Welcome to Serverless Iceberg Analytics

Check out our demo of serverless Iceberg analytics using the DuckDB Table Visualizer{:.button.yellow}

> The current credentials in the demo are provided via a throwaway account with minimal permissions. If you enter your own credentials and share a link, you will be sharing your credentials.

Access Your Own Data

Substituting your own S3 Tables bucket ARN and credentials with policy `AmazonS3TablesReadOnlyAccess`, you can also access your catalog, metadata and data. Computations are fully local, and the credentials and warehouse ID are only sent to the catalog endpoint specified in your ATTACH command. Inputs are translated to SQL, and added to the hash segment of the URL.

This means that:

- no sensitive data is handled or sent to `duckdb.org`
- computations are local, fully in your browser
- you can use the familiar SQL interface with the same code snippets that can run everywhere DuckDB runs
- if you edit the credentials and share the resulting link, you will be sharing the new credentials

As of today, this works with Amazon S3 Tables. This has been implemented through a collaboration with the Amazon S3 Tables team. To learn more about S3 Tables, how to get started and their feature set, you can take a look at their product page or documentation. A demo of DuckDB querying S3 Tables from a browser was presented at AWS re:Invent 2025 – see the presentation.

Conclusion

The DuckDB-Iceberg extension is now supported in DuckDB-Wasm and it can read and edit Iceberg REST Catalogs. Users can now access Iceberg data from within a browser, without having to install or manage any compute nodes!

If you would like to provide feedback or file issues, please reach out to us on either the DuckDB-Wasm or DuckDB-Iceberg repository. If you are interested in using any part of this within your organization, feel free to reach out.

DuckDB on LoongArch

Publication date: 2026-01-06

Author: Hannes Mühleisen

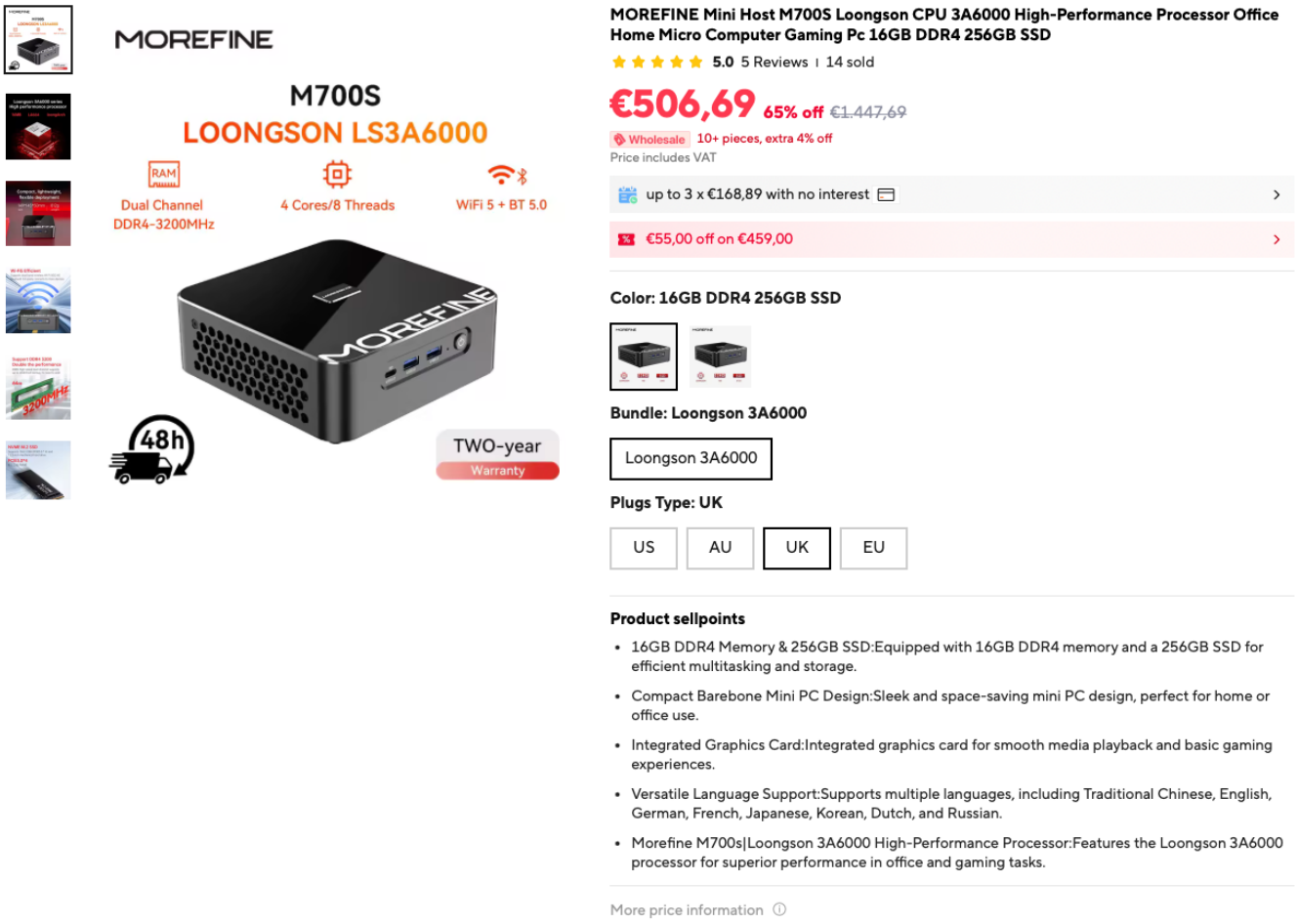
TL;DR: In today's “What's on your desk?” episode, we test a Loongson CPU with the LoongArch architecture.

It’s not every day that a new CPU architecture arrives on your desk. I grew up on the Intel 486 back in the early 90s. I also still remember AMD releasing its 64-bit x86 extension in 2000. Then not a lot happened until Apple released the ARM-based M1 architecture in 2020. But today is the day again (for me), with the long-awaited arrival of the “MOREFINE M700S” in our office.

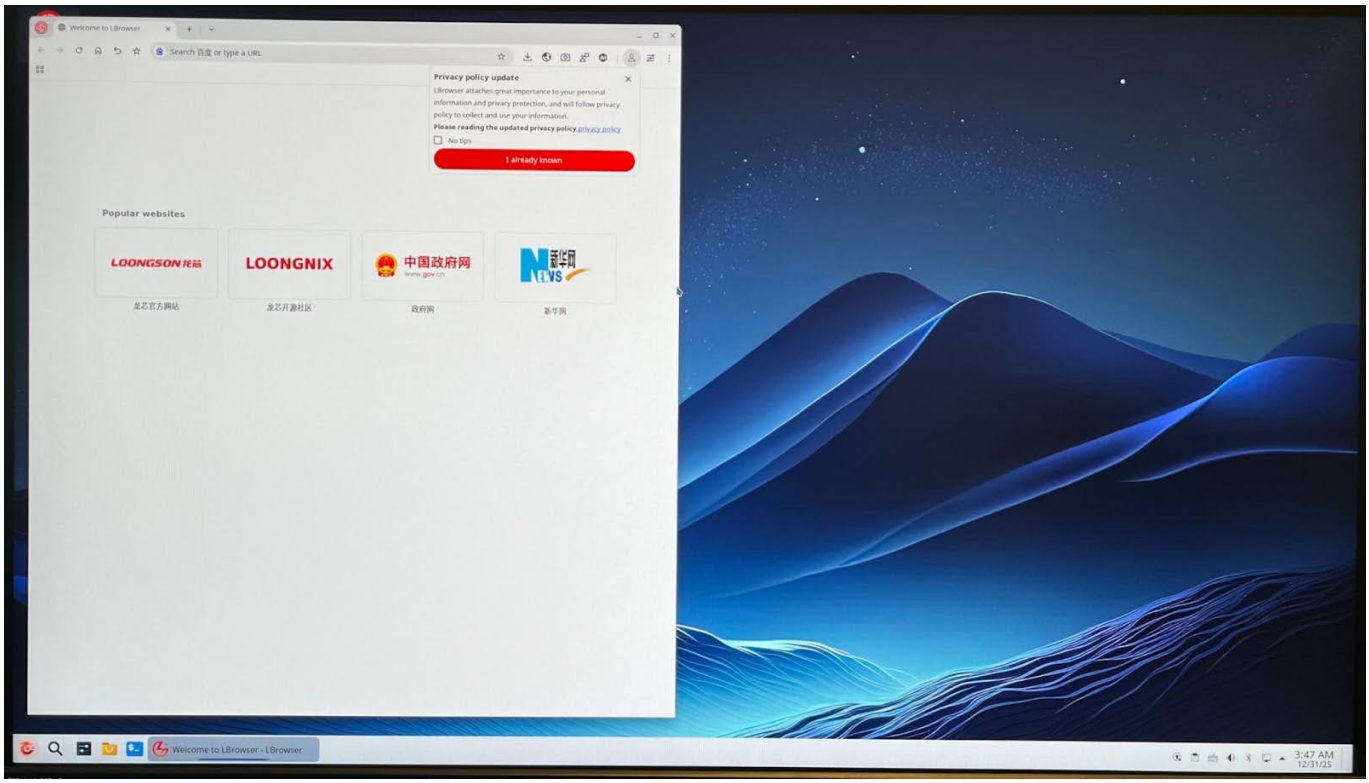
The M700S contains a Loongson CPU. Also called “LoongArch” or “Godson” processors, this CPU was developed in China based on the (somewhat esoteric) MIPS architecture. This is part of a plan to become technologically self-sufficient as part of the government-funded Made in China 2025 plan.

It is probably safe to assume that – given the ongoing trade shenanigans – the Loongson will become much more popular in China as time goes on. DuckDB already sees quite a lot of usage from China, so naturally we want to make sure that DuckDB runs well on the Loongson. Thankfully, one of our community members has already opened a pull request with two minimal changes to allow DuckDB to compile. We became curious.

We purchased the M700S on (where else?) AliExpress for around 500 EUR. Besides the Loongson 8-core 3A6000 CPU it contains 16 GB of main memory and a 256 GB solid-state disk.



Once plugged in and booted up, things feel pretty normal besides the loud fan that seems to be always on. On the screen, a variant of Debian called Loongnix boots up. The GUI seems to be KDE-based and comes with a custom browser "LBrowser" which is a fork of Chromium. Just because it was not obvious we document it here: the default `root` password is `M700S`. There is also a user account `m700s` with the same password.



Overall, the software seems a little dated, even after running `apt upgrade`: the Linux kernel seems to be version 4.19, which was released back in 2018, and which has been EOL for a year now. The GCC version is 8.3, which similarly came out in 2019.

With the aforementioned patch, we managed to compile DuckDB 1.4.3 on Loongnix. There was one small issue where the CMake file `append_metadata.cmake` was not compatible with the older CMake version (3.13.4) available on Loongnix. But simply replacing that file with an empty one allowed us to complete the build. Of course we could also have updated CMake, but life is short. Once completed, we ran DuckDB's extensive unit test suite (`make allunit`) to confirm that our build runs correctly on the Loongson CPU. Results looked good.

For performance comparison, we re-used the methodology from our previous blog post that ran DuckDB on a Raspberry Pi. In short, we run the 22 TPC-H benchmark queries on "Scale Factor" 100 and 300, which in DuckDB format is a 25 GB and 78 GB database file, respectively. We compare those numbers with the nearest computer, which is my day-to-day MacBook Pro with an M3 Max CPU. For fairness, we limit DuckDB to 14 GB of RAM on both platforms. The reported timings are "hot" runs, meaning we re-ran the query set and took the timings from the second run.

Here are the results, and they are not great. We start with aggregated timings:

SF	System	Geometric mean	Sum
SF100	MacBook	0.6	16.9
SF100	MOREFINE	6.1	192.8
SF300	MacBook	2.8	78.8
SF300	MOREFINE	27.3	791.6

We can see that the MacBook is around *ten times faster* than the MOREFINE on this benchmark, both in the geometric mean of runtimes as well as in the sum. If you are interested in the individual query runtimes, you can find them below. Click here to see the individual query runtimes. Q SF100/MacBook SF100/MOREFINE SF300/MacBook SF300/MOREFINE 1 1.247 7.363 4.528 26.475 2 0.117 1.058 0.474 4.101 3 0.697 8.563 2.759 32.432 4 0.570 7.348 2.331 27.185 5 0.631 8.498 3.217 34.462 6 0.180 1.236 1.395 13.225 7 0.620 7.702 3.119 37.411 8 0.640 5.593 3.611 29.914 9 1.906 30.560 6.670 99.884 10 0.923 11.755 4.036 40.412 11 0.102 1.037 0.709 4.444 12 0.535 6.422 2.918 31.501 13 1.847

21.185 6.394 74.081 14 0.408 5.616 3.240 26.613 15 0.252 2.652 1.906 17.454 16 0.273 3.108 0.879 11.480 17 0.805 5.184 4.655 28.469 18 1.538 15.492 7.619 71.845 19 0.779 9.143 4.379 39.111 20 0.441 4.993 3.234 25.967 21 1.996 23.231 9.503 96.452 22 0.441 5.036 1.237 18.709

It is always exciting to get DuckDB running on a new platform. Of course, we have built DuckDB to be ulta-portable and agnostic to hardware environments while still delivering excellent performance. So it was not that surprising that it was not that difficult to get DuckDB running on the MOREFINE with its new-ish CPU. However, performance on the standard TPC-H benchmark was not that impressive, with the MacBook being around ten times faster than the MOREFINE.

Of course, there are many opportunities for improvement. For starters, the `gcc` toolchain on LoongArch is likely not as advanced by far compared with its x86/ARM counterpart, so advances there could make a big difference. The same applies of course to IO performance, which we have not measured separately. But hey, the "glass half full" department could also rightfully claim that the Loongson CPU can complete TPC-H SF300!

One could also argue that a MacBook Pro is much more expensive than 500 EUR MOREFINE. However, a recent M4 Mac Mini with the same memory and storage specs will cost around 700 EUR, not that much more all things considered. It will run circles around the MOREFINE. And it will not constantly annoy you with its fan.

Announcing Vortex Support in DuckDB

Publication date: 2026-01-23

Authors: Guillermo Sanchez, SpiralDB Team

TL;DR: Vortex is a new columnar file format with a very promising design. SpiralDB and DuckLabs have partnered to give you a very fast experience while reading and writing Vortex files!

I think it is worth starting this intro by talking a little bit about the established format for columnar data. Parquet has done some amazing things for analytics. If you go back to the times where CSV was the better alternative, then you know how important Parquet is. However, even if the specification has evolved over time, Parquet has some design constraints. A particular limitation is that it is block-compressed and engines need to decompress pages in order to do further operations like filtering, decoding values, etc. For a while, researchers and private companies have been working on alternatives to Parquet that could improve on some of Parquet's shortcomings. Vortex, from the SpiralDB team, is one of them.

What is Vortex?

Vortex is an extensible, open source format for columnar data. It was created to handle heterogeneous compute patterns and different data modalities. But, what does this mean?

> The project was donated to the Linux Foundation by the SpiralDB team in August 2025.

Vortex provides different layouts and encodings for different data types. Some of the most notable are ALP for floating point encoding or FSST for string encoding. This lightweight compression strategy keeps data sizes down while allowing one of Vortex's most important features: compute functions. By knowing the encoded layout of the data, Vortex is able to run arbitrary expressions on compressed data. This allows a Vortex reader to execute, for example, filter expressions within storage segments without decompressing data.

We mentioned heterogeneous compute to emphasize that Vortex was designed with the idea of having optimized layouts for different data types, including vectors, large text or even image or audio, but also to maximize CPU or GPU saturation. The idea is that decompression is deferred all the way to the GPU or CPU, enabling what Vortex calls "late materialization". The FastLanes encoding, a project originating at CWI (like DuckDB), is one of the main drivers behind this feature.

Vortex also supports dynamically loaded libraries (similar to DuckDB extensions) to provide new encodings for specific types as well as specific compute functions, e.g. for geospatial data. Another very interesting feature is encoding WebAssembly into the file, which can allow the reader to benefit from specific compute kernels applied to the file.

Besides DuckDB, other engines such as DataFusion, Spark and Arrow already offer integration with Vortex.

> For more information, check out the Vortex documentation.

The DuckDB Vortex Extension

DuckDB is a database as the name says, yes, but it is also widely used as an engine to query many different data sources. Through core or community extensions, DuckDB can integrate with:

- Databases like Snowflake, BigQuery or PostgreSQL.
- Lakehouse formats like Delta, Iceberg or DuckLake.
- File formats, most notably JSON, CSV, Parquet and most recently Vortex.

> The community has gotten very creative, though, so these days you can even read YAML and Markdown with DuckDB using community extensions.

All this is possible due to the DuckDB extension system, which makes it relatively easy to implement logic to interact with different file formats or external systems.

The SpiralDB team built a DuckDB extension. Together with the DuckLabs team, we have made the extension available as a core DuckDB extension, so that the community can enjoy Vortex as a first-class citizen in DuckDB.

Example Usage

Installing and using the Vortex extension is very simple:

```
INSTALL vortex;
LOAD vortex;
```

Then, you can easily use it to read and write, similar to other extensions such as Parquet.

```
SELECT * FROM read_vortex('my.vortex');

COPY (SELECT * FROM generate_series(0, 3) t(i))
TO 'my.vortex' (FORMAT vortex);
```

Why Vortex and DuckDB?

Vortex claims to do well primarily at three use cases:

- Traditional SQL analytics: Through late decompression and compute expressions on compressed data, Vortex can filter down data within the storage segment, reducing IO and memory consumption.
- Machine learning pre-processing pipelines: By supporting a wide variety of encodings for different data types, Vortex claims to be effective at reading and writing data, whether it is audio, text, images or vectors.
- AI model training: Encodings such as FastLanes allow for a very efficient copy of data to the GPU. Vortex is aiming at being able to copy data directly from S3 object storage to the GPU.

The promise of more efficient IO and memory use through late decompression is a good reason to try DuckDB and Vortex for SQL analytics. On another note, if you are looking at running analytics on unified datasets that are used for multiple use cases, including pre-processing pipelines and AI training, then Vortex may be a good candidate since it is designed to fit all of these use cases well.

Performance Experiment

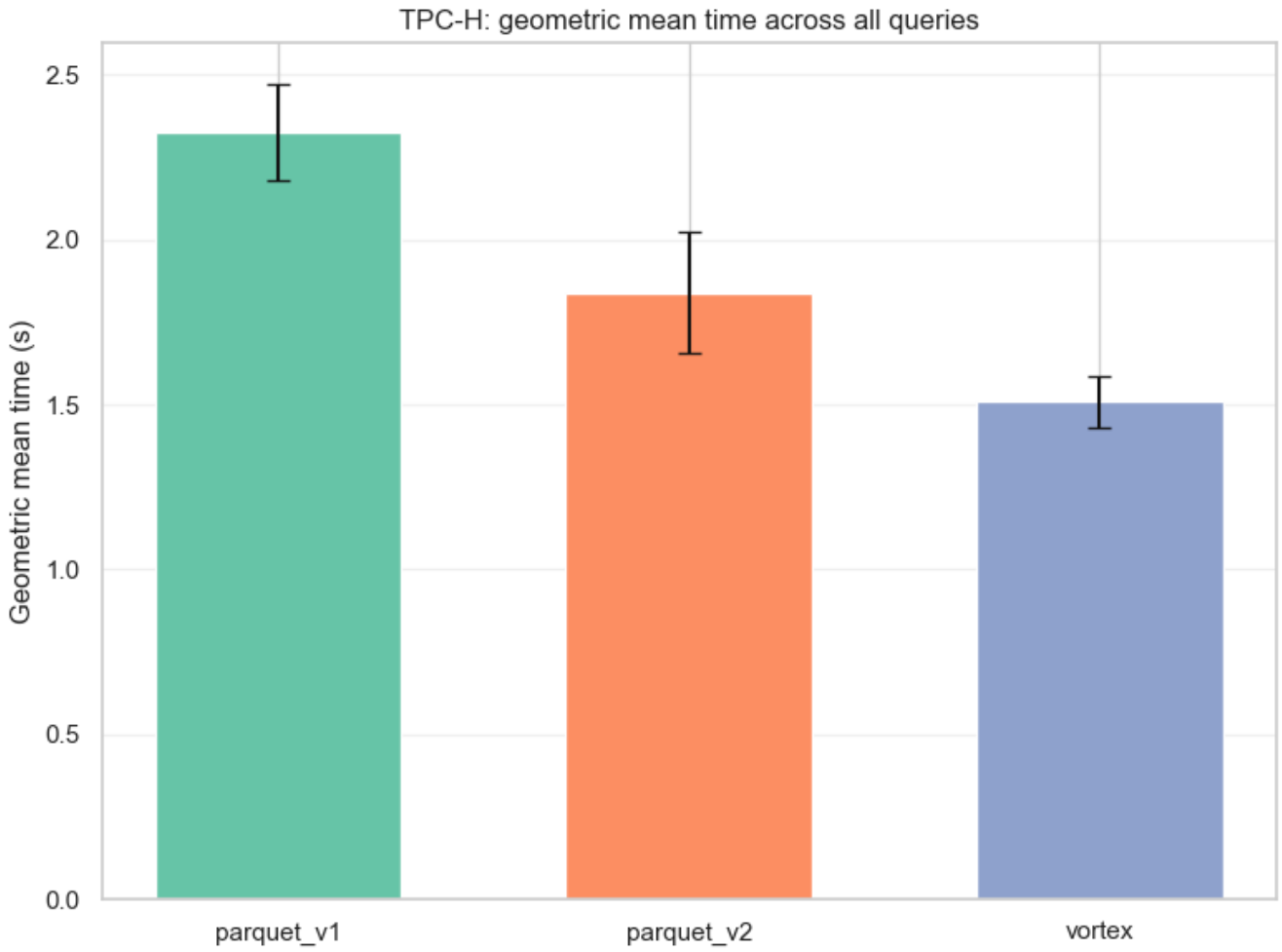
For those who are number hungry, we decided to run a TPC-H benchmark scale factor 100 with DuckDB to understand how Vortex can perform as a storage format compared to Parquet. We tried to make the benchmark as fair as possible. These are the parameters:

- Run on Mac M1 with 10 cores & 32 GB of memory.
- The benchmark runs each query 5 times and the average is used for the final report.
- The DuckDB connection is closed after each query to try to make runs “colder” and avoid DuckDB's caching (particularly with Parquet) from influencing the results. OS page caching does have an influence in subsequent runs but we decided to acknowledge this factor and still keep the first run.
- Each TPC-H table is a single file, which means that lineitem files for Parquet and Vortex are quite large (both around 20 GB). This allows us to ignore the effect of globbing and having many small files.

- Data files used for the benchmark are generated with tpchgen-rs and are copied out using DuckDB's Parquet and Vortex extensions.
- We compared Vortex against Parquet v1 and v2. The v2 specification allows for considerably faster reading than the v1 specification but many writers do not support this, so we thought it was worth including both.

The results are very good. The TPC-H benchmark runs 18% faster with respect to Parquet V2 and 35% faster than Parquet V1 (using the geometric means, which is the recommended approach).

Another interesting result is the standard deviation across runs. There was a considerable difference between the first (and coldest) run of each query and subsequent runs in Parquet, while Vortex performed very well across all runs with a much smaller standard deviation.



Format	Geometric Mean (s)	Arithmetic Mean (s)	Avg Std Dev (s)	Total Time (s)
parquet_v1	2.324712	2.875722	0.145914	63.265881
parquet_v2	1.839171	2.288013	0.182962	50.336281
vortex	1.507675	1.991289	0.078893	43.808349

> The times did vary across different runs of the same benchmark, and subsequent runs have yielded similar results but with slight variations. The differences between Parquet v2 and Vortex have always been around 12-18% in geometric means and around 8-14% in total times. Benchmarking is very hard!

Conclusion

Vortex is a very interesting alternative to established columnar formats like Parquet. Its focus on lightweight compression encodings, late decompression and being able to run compute expressions on compressed data makes it very interesting for a wide range of use cases. With regard to DuckDB, we see that Vortex is already very performant for analytical queries, where it is on par or better than Parquet v2 on the TPC-H benchmark queries.

> Vortex has been backwards compatible since version 0.36.0, which was released more than 6 months ago. Vortex is now at version 0.56.0.

Announcing DuckDB 1.4.4 LTS

Publication date: 2026-01-26

Author: The DuckDB team

TL;DR: Today we are releasing DuckDB 1.4.4 with bugfixes and performance improvements.

In this blog post, we highlight a few important fixes in DuckDB v1.4.4, the fourth patch release in DuckDB's 1.4 LTS line. The release ships bugfixes, performance improvements and security patches. You can find the complete release notes on GitHub.

To install the new version, please visit the installation page.

Fixes

This version ships a number of performance improvements and bugfixes.

Correctness

- `#20008` Unexpected Result when Using Utility Function ALIAS #20008
- `#20410` ANTI JOIN produces wrong results with materialized CTEs
- `#20156` Streaming window unions produce incorrect results
- `#20413` ASOF joins with `predicate` fail with different errors for `FULL`, `RIGHT`, `SEMI`, and `ANTI` join types
- `#20090` mode() produces corrupted UTF-8 strings in parallel execution

Crashes and Internal Errors

- `#20468` Segfault in Hive partitioning with NULL values
- `#20086` Incorrect results when using positional joins and indexes
- `#20415` C API data creation causes segfault

Performance

- `#20252` Optimize prepared statement parameter lookups
- `#20284` dbgen: use TaskExecutor framework to respect the `threads` setting

Miscellaneous

- `#20233` Function chaining not allowed in `QUALIFY`
- `#20339` Use UTF-16 console output in Windows shell

Conclusion

This post was a short summary of the changes in v1.4.4. As usual, you can find the full release notes on GitHub. We would like to thank our contributors for providing detailed issue reports and patches. In the coming month, we'll release DuckDB v1.5.0. We'll also keep v1.4 LTS updated until mid-September. We'll announce the release date of v1.4.5 in the release calendar in the coming months.

> Earlier today, we pushed an incorrect tag that was visible for a few minutes. No binaries or extensions were available under this tag and we replaced it as soon as we noticed the issue. Our apologies for the erroneous release.

Announcing DuckDB 1.5.0

Publication date: 2026-03-09

Author: The DuckDB team

TL;DR: We are releasing DuckDB version 1.5.0, codenamed "Variegata". This release comes with a friendly CLI (a new, more ergonomic command line client), support for the `VARIANT` type, a built-in `GEOMETRY` type, along with many other features and optimizations. The v1.4.0 LTS line ("Andium") will keep receiving updates until its end-of-life in September 2026.

We are proud to release DuckDB v1.5.0, codenamed "Variegata" after the *Paradise shelduck* (Tadorna variegata) endemic to New Zealand.

In this blog post, we cover the most important updates for this release around support, features and extensions. As always, there is more: for the complete release notes, see the release page on GitHub.

> To install the new version, please visit the installation page due to the extra changes and review rounds required.

With this release, we will have two DuckDB releases available: v1.4 (LTS) and v1.5 (current). The next release – planned for September – will ship a major version, DuckDB v2.0.

New Features

Command Line Client

For users who use DuckDB through the terminal, the highlight of the new release is a rework of the CLI client with a new color scheme, dynamic prompts, a pager and many other convenience features.

Color Scheme We shipped a new color palette and harmonized it with the documentation. The color palette is available in both dark mode and light mode. Both use two shades of gray, and five colors for keywords, strings, errors, functions and numbers. You can find the color palette in the Design Manual.

You can customize the color scheme using the `.highlight_colors` dot command:

```
.highlight_colors column_name darkgreen bold_underline
.highlight_colors numeric_value red bold
.highlight_colors string_value purple2
FROM ducks;
```

```
DuckDB v1.5.0 (Variegata)
Enter ".help" for usage hints.
ducks D .highlight_colors column_name darkgreen bold_underline
ducks D .highlight_colors numeric_value red bold
ducks D .highlight_colors string_value purple2
ducks D FROM ducks;
```

id int32	name varchar	extinct_year int32
1	Labrador Duck	1878
2	Mallard	NULL
3	Crested Shelduck	1964
4	Wood Duck	NULL
5	Pink-headed Duck	1949

{: .lightmode-img }

Dynamic Prompts in the CLI DuckDB v1.5.0 introduces dynamic prompts for the CLI (PR #19579). By default, these show the database and schema that you are currently connected to:

duckdb

```
«memory» D ATTACH 'my_database.duckdb';
«memory» D USE my_database;
«my_database» D CREATE SCHEMA my_schema;
«my_database» D USE my_schema;
«my_database.my_schema» D ..
```

These prompts can be configured using bracket codes to have a maximum length, run a custom query, use different colors, etc. (#19579).

`.tables` and `DESCRIBE` To show the columns of an individual table, use the `DESCRIBE` statement:

```
«memory» D ATTACH 'https://blobs.duckdb.org/data/animals.db' AS animals_db;
«memory» D USE animals_db;
«animals_db» D DESCRIBE ducks;
```

```
┌─────────────┬─────────┐
│    ducks    │         │
├─────────────┼─────────┤
│ id          │ integer │
│ name        │ varchar │
│ extinct_year│ integer │
└─────────────┴─────────┘
```

The `.tables` dot command lists the attached catalogs, the schemas and tables in them, and the columns in each table.

```
«memory» D ATTACH 'https://blobs.duckdb.org/data/animals.db' AS animals_db;
«memory» D ATTACH 'https://blobs.duckdb.org/data/numbers1.db';
«memory» D .tables

──────────── animals_db ────────────
──────────────── main ────────────────
```

```
┌──────────────────┐
│      swans       │
│                  │
│ id      integer  │
│ name    varchar  │
│ species varchar  │
│ color   varchar  │
│ habitat varchar  │
│                  │
│     3 rows       │
└──────────────────┘
```

```
┌──────────────────────┐
│        ducks         │
│                      │
│ id           integer │
│ name         varchar │
│ extinct_year integer │
│                      │
│       5 rows         │
└──────────────────────┘
```

```
  numbers1
── main ──
```

```
┌──────────┐
│   tbl    │
│          │
│ i bigint │
│          │
│  2 rows  │
└──────────┘
```

Accessing the Last Result Using _ You can access the last result of a query inline using the underscore character _. This is not only convenient but also makes it unnecessary to re-run potentially long-running queries:

```
«memory» D ATTACH 'https://blobs.duckdb.org/data/animals.db' AS animals_db;
«memory» D USE animals_db;
«animals_db» D FROM ducks WHERE extinct_year IS NOT NULL;
```

id int32	name varchar	extinct_year int32
1	Labrador Duck	1878
3	Crested Shelduck	1964
5	Pink-headed Duck	1949

```
«animals_db» D FROM _;
```

id int32	name varchar	extinct_year int32
1	Labrador Duck	1878
3	Crested Shelduck	1964
5	Pink-headed Duck	1949

Pager Last but not least, the CLI now has a pager! It is triggered when there are more than 50 rows in the results.

```
«memory» D .maxrows 100
«memory» D FROM range(0, 100);
```

You can navigate on Linux and Windows using Page Up / Page Down. On macOS, use Fn + Up / Down. To exit the pager, press Q.

The initial implementation of the pager was provided by tobwen in #19004.

PEG Parser

DuckDB v1.5 ships an experimental parser based on PEG (Parser Expression Grammars). The new parser enables better suggestions, improved error messages, and allows extensions to extend the grammar. The PEG parser is currently disabled by default but you can opt-in using:

```
CALL enable_peg_parser();
```

The PEG parser is already used for generating suggestions. You can cycle through the options using TAB.

```
«animals_db» D FROM ducks WHERE habitat IS
IS           ISNULL       ILIKE        ⌧IN⌧          INTERSECT    LIKE
```

We are planning to make the switch to the new parser in the upcoming DuckDB release.

> As a tradeoff, the parser has a slight performance overhead, however, this is in the range of milliseconds and is thus negligible for analytical queries. For more details on the rationale for using a PEG parser and benchmark results, please refer to the CIDR 2026 paper by Hannes and Mark, or their blog post summarizing the paper.

`VARIANT` Type

DuckDB now natively supports the `VARIANT` type, inspired by Snowflake's semi-structured `VARIANT` data type and available in Parquet since 2025. Unlike the JSON type, which is physically stored as text, VARIANT stores typed, binary data. Each row in a VARIANT column is self-contained with its own type information. This leads to better compression and query performance. Here are a few examples of using `VARIANT`.

Store different types in the same column:

```
CREATE TABLE events (id INTEGER, data VARIANT);
INSERT INTO events VALUES
    (1, 42::VARIANT),
    (2, 'hello world'::VARIANT),
    (3, [1, 2, 3]::VARIANT),
    (4, {'name': 'Alice', 'age': 30}::VARIANT);

SELECT * FROM events;
```

id int32	data variant
1	42
2	hello world
3	[1, 2, 3]
4	{'name': Alice, 'age': 30}

Check the underlying type of each row:

```
SELECT id, data, variant_typeof(data) AS vtype
FROM events;
```

id int32	data variant	vtype varchar
1	42	INT32
2	hello world	VARCHAR
3	[1, 2, 3]	ARRAY(3)
4	{'name': Alice, 'age': 30}	OBJECT(name, age)

You can extract fields from nested variants using the dot notation or the `variant_extract` function:

```
SELECT data.name FROM events WHERE id = 4;
-- or
SELECT variant_extract(data, 'name') AS name FROM events WHERE id = 4;
```

name variant

```
┌─────────┐
│  Alice  │
└─────────┘
```

DuckDB also supports reading `VARIANT` types from Parquet files, including *shredding* (storing nested data as flat values).

`read_duckdb` Function

The `read_duckdb` table function can read DuckDB databases without first attaching them. This can make reading from DuckDB databases more ergonomic – for example, you can use globbing. You can read the example `numbers` databases as follows:

```
SELECT min(i), max(i)
FROM read_duckdb('numbers*.db');
```

```
┌─────────┬─────────┐
│ min(i)  │ max(i)  │
│  int64  │  int64  │
├─────────┼─────────┤
│       1 │       5 │
└─────────┴─────────┘
```

Azure Writes

You can now write to the Azure Blob or ADLSv2 storage using the COPY statement:

```
-- Write query results to a Parquet file on Blob Storage
COPY (SELECT * FROM my_table)
TO 'az://my_container/path/output.parquet';

-- Write a table to a CSV file on ADLSv2 Storage
COPY my_table
TO 'abfss://my_container/path/output.csv';
```

ODBC Scanner

We are now shipping an ODBC scanner extension. This allows you to query a remote endpoint as follows:

```
LOAD odbc_scanner;
SET VARIABLE conn = odbc_connect('Driver={Oracle Driver};DBQ=//127.0.0.1:1521/XE;UID=scott;PWD=tiger;');
SELECT * FROM odbc_query(getvariable('conn'), 'SELECT SYSTIMESTAMP FROM dual;');
```

In the coming weeks, we'll publish the documentation page and release a followup post on the ODBC scanner. In the meantime, please refer to the project's README.

Major Changes

Breaking Change for Datetime Function

The `date_trunc` function, when applied to a `DATE`, now returns a `TIMESTAMP` instead of a date.

```
-- v1.4.4:
SELECT typeof(date_trunc('month', DATE('2026-03-27')));
-- returns DATE

-- v1.5.x:
SELECT typeof(date_trunc('month', DATE('2026-03-27')));
-- returns TIMESTAMP
```

Lakehouse Updates

All of DuckDB's supported Lakehouse formats have received some updates in DuckDB v1.5.

DuckLake The main DuckLake change for DuckDB v1.5 is updating the DuckLake specification to v0.4. We are aiming for this to be the same specification that ships with DuckLake v1.0, which will be released in April. Its main highlights include:

- Macro support.
- Sorted tables.
- Deletion inlining and addition of partial delete files.
- Internal rework of DuckLake options.

We'll announce more details about these features in the blog post for DuckLake v1.0.

Delta Lake For the Delta Lake extension, the team has focused on improving support for writes via Unity Catalog, Delta idempotent writes and table `CHECKPOINT`s.

Iceberg For the Iceberg extension, the team is working on a larger release for v1.5.1. For v1.5.0, the main feature is the addition of table properties in the `CREATE TABLE` statement:

```
CREATE TABLE test_create_table (a INTEGER)
WITH (
    'format-version' = '2', -- format version will be elevated to format-version when creating a table
    'location' = 's3://path/to/data', -- location will be elevated to location when creating a table
    'property1' = 'value1',
    'property2' = 'value2'
);
```

Other minor additions have been made to enable passing `EXTRA_HTTP_HEADERS` when attaching to an Iceberg catalog, which has unlocked Google's BigLake.

> Both Delta and DuckLake have implemented the `VARIANT` type. Iceberg's `VARIANT` type will ship in the v1.5.1 release with some other features that are specific to the Iceberg v3 specification.

Network Stack

The default backend for the httpfs extension has changed from `httplib` to `curl`. As one of the most popular and well-tested open-source projects, we expect `curl` to provide long-standing stability and security for DuckDB. Regardless of the `http` library used, `openssl` is still the backing SSL library and options such as `http_timeout`, `http_retries`, etc. are still the same.

Our community has been testing the new network stack for the last few weeks. Still, if you encounter any issues, please submit them to the `duckdb-httpfs` repository.

Lambda Syntax

Up to DuckDB v1.2, the syntax for defining lambda expressions used the arrow notation `x -> x + 1`. While this was a nice syntax, it clashed with the JSON extract operator (`->`) due to operator precedence and led to error messages that some users found difficult to troubleshoot. To work around this, we introduced a new, Python-style lambda syntax in v1.3, `lambda x: x + 1`.

While DuckDB v1.5 supports both styles of writing lambda expressions, using the deprecated arrow syntax will now throw a warning:

```
SELECT list_transform([1, 2, 3], x -> x + 1);
```

```
WARNING:
Deprecated lambda arrow (->) detected. Please transition to the new lambda syntax, i.e., lambda x, i: x +
i, before DuckDB's next release.
```

You can use the `lambda_syntax` configuration option to change this behavior to suppress the warning or to behave more strictly:

```
-- Suppress the warning
SET lambda_syntax = 'enable_single_arrow';
-- Turn the deprecation warning into an error
SET lambda_syntax = 'disable_single_arrow';
```

DuckDB 2.0 will disable the single arrow syntax by default and it will only be available if you opt-in explicitly.

Spatial Extension

The spatial extension ships several important changes.

Breaking Change: Flipping of Axis Order Most functions in `spatial` operate in Cartesian space and are unaffected by axis order, e.g., whether the X and Y axes represent "longitude" and "latitude" or the other way around. But there are some functions where this matters, and where the assumption, counterintuitively, is that all input geometries use (x = latitude, y = longitude). These are:

- `ST_Distance_Spheroid`
- `ST_Perimeter_Spheroid`
- `ST_Area_Spheroid`
- `ST_Distance_Sphere`
- `ST_DWithin_Spheroid`

Additionally, `ST_Transform` also expects that the input geometries are in the same axis order as defined by the source coordinate reference system, which in the case of e.g., `EPSG:4326` is also (x = latitude, y = longitude).

This has been a long-standing source of confusion and numerous issues, as other databases, formats and GIS systems tend to always treat X as "easting", "left-right" or "longitude", and Y as "northing", "up-down" or "latitude".

We are changing how this currently works in DuckDB to be consistent with how other systems operate, and hopefully cause less confusion for new users in the future. However, to avoid silently breaking existing workflows that have adapted to this quirk (e.g., by using `ST_FlipCoordinates`), we are rolling out this change gradually via a new `geometry_always_xy` setting:

- In DuckDB v1.5, setting `geometry_always_xy = true` enables the new behavior (x = longitude, y = latitude). Without it, affected functions emit a warning.
- In DuckDB v2.0, the warning will become an error. Set `geometry_always_xy = false` to preserve the old behavior.
- In DuckDB v2.1, `geometry_always_xy = true` will become the default.

So to summarize, nothing is changing by default in this release, but to avoid being affected by this change in the future, set `geometry_always_xy` explicitly now. Set it to `true` to opt into the new behavior, or `false` to keep the existing one.

Geometry Rework

GEOMETRY Becomes a Built-In Type The `GEOMETRY` type has been moved from the `spatial` extension into core DuckDB!

Geospatial data is no longer niche. The Parquet standard now treats `GEOMETRY` as a first-class column type, and open table formats like Apache Iceberg and DuckLake are moving in the same direction. Many widely used data formats and systems also have geospatial counterparts—GeoJSON, PostGIS, GeoPandas, GeoPackage/Spatialite, and more.

DuckDB already offers extensions that integrate with many of these formats and systems. But there's a structural problem: as long as `GEOMETRY` lives inside the `spatial` extension, other extensions that want to read or write geospatial data must either depend on `spatial`, implement their own incompatible geometry representation, or force users to handle the conversions themselves.

By moving `GEOMETRY` into DuckDB's core, extensions can now produce and consume geometry values natively, without depending on `spatial`. While the `spatial` extension still provides most of the functions for working with geometries, the type itself becomes a shared foundation that the entire ecosystem can build on. We've already added `GEOMETRY` support to the Postgres scanner and GeoArrow conversion for Arrow import and export. Geometry support in additional extensions is coming soon.

This change also enables deeper integration with DuckDB's storage engine and query optimizer, unlocking new compression techniques, query optimizations, and CRS awareness capabilities that were not possible when `GEOMETRY` only existed as an extension type. This is all documented in the new geometry page in the documentation, but we will highlight some below.

Improved Storage: WKB and Shredding Geometry values are now stored using the industry-standard little-endian Well-Known Binary (WKB) encoding, replacing the custom format used by the `spatial` extension. However, we are still experimenting with the in-memory representation we want to use in the execution engine so you should still use the conversion functions (e.g., `ST_AsWKT`, `ST_AsWKB`, `ST_GeomFromText`, `ST_GeomFromWKB`) when moving data in and out of DuckDB.

We've also implemented a new storage technique specialized for `GEOMETRY`. When a geometry column contains values that all share the same type and vertex dimensions, DuckDB can additionally apply "shredding": rather than storing opaque blobs, the column is decomposed into primitive `STRUCT`, `LIST`, and `DOUBLE` segments that compress far more efficiently. This can reduce on-disk size by roughly 3x for uniform geometry columns such as point clouds. Shredding is applied automatically for uniform row groups of a certain size, but can be configured via the `geometry_minimum_shredding_size` configuration option.

Geometry Statistics and Query Optimization Geometry columns now track per-row-group statistics - including the bounding box and the set of geometry types and vertex dimensions present. The query optimizer can use these to skip row groups that cannot match a query's spatial predicates, similar to min/max pruning for numeric columns. The && (bounding box intersection) operator is the first to benefit; broader support across `spatial` functions is in progress.

Coordinate Reference System Support The `GEOMETRY` type now accepts an optional CRS parameter (e.g., `GEOMETRY('OGC:CRS84')`), making CRS part of the type system rather than implicit metadata. Spatial functions enforce CRS consistency across their inputs, catching a common class of silent errors that arises when mixing geometries from different coordinate systems. Only a couple of CRSs are built in by default, but loading the `spatial` extension registers over 7,000 CRSs from the EPSG dataset. While CRS support is still a bit experimental, we are planning to develop it further to support e.g., custom CRS definitions.

Optimizations

Non-Blocking Checkpointing During checkpointing, it's now possible to run concurrent reads (#19867), writes (#20052), insertions with indexes (#20160) and deletes (#20286). The rework of checkpointing benefits concurrent RW workloads and increases the TPC-H throughput score on SF100 from 246,115.60 to 287,122.97, a **17% improvement**.

Aggregates Aggregate functions received several optimizations. For example, the `last` aggregate function was optimized by community member `xe-nvdk` to iterate from the end of each vector batch instead of the beginning. In synthetic benchmarks, this results in a 40% speedup.

Distribution

Python Pip You can install the DuckDB CLI on any platform where pip is available:

```
pip install duckdb-cli
```

You can then launch DuckDB in your virtual environment using:

```
duckdb
```

Both DuckDB v1.4 and v1.5 are supported. We are working on shipping extensions as extras using the `duckdb[extension_name]` syntax – stay tuned!

Windows Install Script (Beta) On Windows, you can now use an install script:

```
powershell -NoExit iex (iwr "https://install.duckdb.org/install.ps1").Content
```

Please note that this is currently in the beta stage. If you have any feedback, please let us know.

CLI for Linux with musl libc We are distributing CLI clients that work with musl libc (e.g., for Alpine Linux, commonly used in Docker images). The archives are available on GitHub.

Note that the musl libc CLI client requires the `libstdc++`. To install this package, run:

```
apk add libstdc++
```

Extension Sizes We reworked our build system to make the extension binaries smaller! The DuckLake extension's size was reduced by ~30%, from 17 MB to 12 MB. For smaller extensions such as Excel, the reduction is more than 60%, from 9 MB to 3 MB.

Summary

These were a few highlights – but there are many more features and improvements in this release. There have been over 6500 commits by close to 100 contributors since v1.4. The full release notes can be found on GitHub. We would like to thank our community for providing detailed issue reports and feedback. And again, our special thanks go to external contributors!

PS: If you visited this blog post through a direct link – we also rolled out a new landing page!

Appendix: Example Dataset

Big Data on the Cheapest MacBook

Publication date: 2026-03-11

Author: Gábor Szárnyas

TL;DR: How does the latest entry-level MacBook perform on database workloads? We benchmarked it using ClickBench and TPC-DS SF300. We found that it could complete both workloads, sometimes with surprisingly good results.

Apple released the MacBook Neo today and there is no shortage of tech reviews explaining whether it's the right device for you if you are a student, a photographer or a writer. What they *don't* tell you is whether it fits into our Big Data on Your Laptop ethos. We wanted to answer this *using a data-driven approach,* so we went to the nearest Apple Store, picked one up and took it for a spin.

What's in the Box?

Well, not much! If you buy this machine in the EU, there isn't even a charging brick included. All you get is the laptop and a braided USB-C cable. But you likely already have a few USB-C bricks lying around – let's move on to the laptop itself!

The only part of the hardware specification that you can select is the disk: you can pick either 256 or 512 GB. As our mission is to deal with alleged "Big Data", we picked the larger option, which brings the price to $700 in the US or €800 in the EU. The amount of memory is fixed to 8 GB. And while there is only a single CPU option, it is quite an interesting one: this laptop is powered by the 6-core Apple A18 Pro, originally built for the iPhone 16 Pro.

It turns out that we have already tested this phone under some unusual circumstances. Back in 2024, with DuckDB v1.2-dev, we found that the iPhone 16 Pro could complete all TPC-H queries at scale factor 100 in about 10 minutes when air-cooled and in less than 8 minutes while lying in a box of dry ice. The MacBook Neo should definitely be able to handle this workload – but maybe it can even handle a bit more. Cue the inevitable benchmarks!

ClickBench

For our first experiment, we used ClickBench, an analytical database benchmark. ClickBench has 43 queries that focus on aggregation and filtering operations. The operations run on a single wide table with 100M rows, which uses about 14 GB when serialized to Parquet and 75 GB when stored in CSV format.

Benchmark Environment

We ported ClickBench's DuckDB implementation to macOS and ran it on the MacBook Neo using the freshly minted v1.5.0 release. We only applied a small tweak: as suggested in our performance guide, we slightly lowered the memory limit to 5 GB, to reduce relying on

the OS' swapping and to let DuckDB handle memory management for larger-than-memory workloads. This is a common trick in memory-constrained environments where other processes are likely using more than 20% of the total system memory.

We also re-ran ClickBench with DuckDB v1.5.0 on two cloud instances, yielding the following lineup:

- The star of our show, the MacBook Neo with 2 performance cores, 4 efficiency cores and 8 GB RAM
- c6a.4xlarge with 16 AMD EPYC vCPU cores and 32 GB RAM. This instance is popular in ClickBench with about 80 individual results reported.
- c8g.metal-48xl with a whopping 192 Graviton4 vCPU cores and 384 GB RAM. This instance is often at the top of the overall ClickBench leaderboard.

The benchmark script first loaded the Parquet file into the database. Then, as per ClickBench's rules, it ran each query three times to capture both cold runs (the first run when caches are cold) and hot runs (when the system has a chance to exploit e.g. file system caching).

Results and Analysis

Our experiments produced the following aggregate runtimes, in seconds:

Machine	Cold run (median)	Cold run (total)	Hot run (median)	Hot run (total)
MacBook Neo	0.57	59.73	0.41	54.27
c6a.4xlarge	1.34	145.08	0.50	47.86
c8g.metal-48xl	1.54	169.67	0.05	4.35

Cold run. The results start with a big surprise: in the cold run, the MacBook Neo is the clear winner with a sub-second median runtime, *completing all queries in under a minute!* Of course, if we dig deeper into the setups, there is an explanation for this. The cloud instances have network-attached disks, and accessing the database on these dominates the overall query runtimes. The MacBook Neo has a local NVMe SSD, which is far from best-in-class, but still provides relatively quick access on the first read.

Hot run. In the hot runs, the MacBook's *total runtime* only improves by approximately 10%, while the cloud machines come into their own, with the c8g.metal-48xl winning by an order of magnitude. However, it's worth noting that on *median query runtimes* the MacBook Neo can still beat the c6a.4xlarge, a mid-sized cloud instance. And the laptop's *total runtime* is only about 13% slower despite the cloud box having 10 more CPU threads and 4 times as much RAM.

TPC-DS

For our second experiment, we picked the queries of the TPC-DS benchmark. Compared to the ubiquitous TPC-H benchmark, which has 8 tables and 22 queries, TPC-DS has 24 tables and 99 queries, many of which are more complex and include features such as window functions. And while TPC-H has been optimized to death, there is still some semblance of value in TPC-DS results. Let's see whether the cheapest MacBook can handle these queries!

For this round, we used DuckDB's LTS version, v1.4.4. We generated the datasets using DuckDB's `tpcds` extension and set the memory limit to 6 GB.

At SF100, the laptop breezed through most queries with a median query runtime of 1.63 seconds and a total runtime of 15.5 minutes.

At SF300, the memory constraint started to show. While the median query runtime was still quite good at 6.90 seconds, DuckDB occasionally used up to 80 GB of space for spilling to disk and it was clear that some queries were going to take a long time. Most notably, query 67 took 51 minutes to complete. But hardware and software continued to work together tirelessly, and they ultimately passed the test, completing all queries in 79 minutes.

Should You Buy One?

Here's the thing: if you are running Big Data workloads on your laptop every day, you probably shouldn't get the MacBook Neo. Yes, DuckDB runs on it, and can handle a lot of data by leveraging out-of-core processing. But the MacBook Neo's disk I/O is lackluster compared to the Air and Pro models (about 1.5 GB/s compared to 3–6 GB/s), and the 8 GB memory will be limiting in the long run. If you need to process Big Data on the move and can pay up a bit, the other MacBook models will serve your needs better and there are also good options for Linux and Windows.

All that said, if you run DuckDB in the cloud and primarily use your laptop as a client, this is a great device. And you can rest assured that if you *occasionally* need to crunch some data locally, DuckDB on the MacBook Neo will be up to the challenge.

DuckDB.ExtensionKit: Building DuckDB Extensions in C

Publication date: 2026-03-20

Author: Giorgi Dalakishvili

TL;DR: DuckDB.ExtensionKit brings DuckDB extension development to the .NET ecosystem. By building on DuckDB's stable C Extension API and leveraging .NET Native AOT compilation, it lets C# developers define scalar and table functions, which can be shipped as native DuckDB extensions.

Introduction

DuckDB has a flexible extension mechanism that allows extensions to be loaded dynamically at runtime. This makes it easy to extend DuckDB's main feature set without adding everything to the main binary. Extensions can add support for new file formats, introduce custom types, or provide new scalar and table functions. A significant part of DuckDB's functionality is actually implemented using this extension mechanism in the form of core extensions, which are developed alongside the engine itself by the DuckDB team. For example, DuckDB can read and write JSON files via the `json` extension and integrate with PostgreSQL using the `postgres` extension.

DuckDB also has a thriving ecosystem of community extensions, i.e., third-party extensions, maintained by community members, covering a wide range of use cases and integrations. For example, you can expose additional cryptographic functionality through the `crypto` community extension.

How Extensions Are Built Today

Today, developers can use the same C++ API that the core extensions use for developing extensions. A template for creating extensions is available in the `extension-template` repository. While powerful, the C++ extension API is tightly coupled to DuckDB's internal APIs, so it can (and often will) change between DuckDB versions. Additionally, using it requires building the whole DuckDB engine and its documentation is not as complete as that of the C API.

To solve these issues, DuckDB also provides an experimental template for C/C++ based extensions that link with the **C Extension API** of DuckDB. This API provides a stable, backwards-compatible interface for developing extensions and is designed to allow extensions to work across different DuckDB versions. Because it is a C-based API, it can also be used from other programming languages such as Rust.

Even with the C API, writing extensions still means working at a low level, performing manual memory management, and writing a lot of boilerplate code. While the C API solves stability and compatibility, it doesn't solve *developer experience* for higher-level ecosystems. This is where DuckDB.ExtensionKit comes in, aiming to make extension development more accessible to developers working in the .NET ecosystem. By building on top of the DuckDB C Extension API and compiling extensions using the .NET Native AOT (Ahead-of-Time) compilation, DuckDB.ExtensionKit offers the best of both worlds: native DuckDB extensions that integrate like any other extension, combined with the productivity and rich library ecosystem of C# and .NET.

DuckDB.ExtensionKit

DuckDB.ExtensionKit provides a set of C# APIs and build tooling for implementing DuckDB extensions. It exposes the low-level DuckDB C Extension API as C# methods, and also provides type-safe, higher-level APIs for defining scalar and table functions, while still producing native DuckDB extensions. The toolkit also includes a source generator that automatically generates the required boilerplate code, including the native entry point and API initialization.

With DuckDB.ExtensionKit, building an extension closely resembles building a regular C# library. Extension authors create a C# project that references the ExtensionKit runtime and implements functions using the provided, type-safe APIs that expose DuckDB concepts.

At build time, the source generator emits the required boilerplate, including the native entry point and extension initialization. The project is then compiled using .NET Native AOT, producing a native DuckDB extension binary that can be loaded and used by DuckDB like any other extension, without requiring a .NET runtime.

To show a concrete example for this process, the following snippet shows a small DuckDB extension implemented using DuckDB.ExtensionKit that exposes both a scalar function and a table function for working with JWTs (JSON Web Token). At a high level, writing an extension with DuckDB.ExtensionKit involves defining a C# type that represents the extension and registering functions explicitly. In the example below, this is done by creating a `partial` class annotated with the `[DuckDBExtension]` attribute and implementing the `RegisterFunctions` method. The implementation makes use of the `System.IdentityModel.Tokens.Jwt` NuGet package, illustrating how extensions can easily take advantage of existing .NET libraries.

We'll add two functions, a scalar function for extracting *a single claim* from a JWT and a table function for extracting *multiple claims.*

```
public static partial class JwtExtension
{
  private static void RegisterFunctions(DuckDBConnection connection)
  {
    connection.RegisterScalarFunction<string, string, string?>("extract_claim_from_jwt",
ExtractClaimFromJwt);

    connection.RegisterTableFunction("extract_claims_from_jwt", (string jwt) => ExtractClaimsFromJwt(jwt),
                                     c => new { claim_name = c.Key, claim_value = c.Value });
  }

  private static string? ExtractClaimFromJwt(string jwt, string claim)
  {
    var jwtHandler = new JwtSecurityTokenHandler();
    var token = jwtHandler.ReadJwtToken(jwt);
    return token.Claims.FirstOrDefault(c => c.Type == claim)?.Value;
  }

  private static Dictionary<string, string> ExtractClaimsFromJwt(string jwt)
  {
    var jwtHandler = new JwtSecurityTokenHandler();
    var token = jwtHandler.ReadJwtToken(jwt);
    return token.Claims.ToDictionary(c => c.Type, c => c.Value);
  }
}
```

In just 25 lines, we have built an extension that adds `extract_claim_from_jwt` and `extract_claims_from_jwt` functions to DuckDB. We can call these functions just like any other function. For example, to extract the `name` field from a claim, we can run:

```
SELECT extract_claim_from_jwt(

    'eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6ImExZmIyY2NjN2FiMjBiMDYyNzJmNGUxMjIwZDEwZmZlIn0.eyJpc3MiOiJodHRwczo
    VuZg2fZFILVpzDyLvGITfUW_18XuDQ_
    SIm3VlfAuHOVHfruuvvSAfjUkTW2Jlrv3ihFYgusV58vjhcVFHssOGMEbtMNo10Jf62dczVVGNZXh_
    OOLS0nTLffhY94sZddqQIE56W8xhLK5YMO4gO8voMzhUwDwucnVvyNfui38MPDNdTSKjn3Ab0hG8jzOVhbYSCHf0eQsbxPzGtXUCJobScWDb78Ip
    noQi94NYjQg2P-AJ5FLCKzKA',
    'name'
);
```

This returns `Giorgi Dalakishvili`. Let's test the table function:

```
SELECT *
FROM extract_claims_from_jwt(

    'eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6ImExZmIyY2NjN2FiMjBiMDYyNzJmNGUxMjIwZDEwZmZlIn0.eyJpc3MiOiJodHRwczo
    VuZg2fZFILVpzDyLvGITfUW_18XuDQ_
    SIm3VlfAuHOVHfruuvvSAfjUkTW2Jlrv3ihFYgusV58vjhcVFHssOGMEbtMNo10Jf62dczVVGNZXh_
    OOLS0nTLffhY94sZddqQIE56W8xhLK5YMO4gO8voMzhUwDwucnVvyNfui38MPDNdTSKjn3Ab0hG8jzOVhbYSCHf0eQsbxPzGtXUCJobScWDb78Ip
    noQi94NYjQg2P-AJ5FLCKzKA'
);
```

This returns:

claim_name	claim_value
iss	https://idp.local
aud	my_client_app
name	Giorgi Dalakishvili
sub	5be86359073c434bad2da3932222dabe
admin	true
exp	1766591267
iat	1766590967

How DuckDB.ExtensionKit Works

DuckDB.ExtensionKit relies on several modern C# language and runtime features to efficiently bridge DuckDB's C extension API to managed code. These features make it possible to build native extensions in C# without introducing a managed runtime dependency at load time.

Function Pointers

DuckDB's C extension API is exposed as a **versioned function table**: a large struct (duckdb_ext_api_v1) whose fields are C function pointers (e.g., `duckdb_open`, `duckdb_register_scalar_function`, `duckdb_vector_get_data`, and so on). DuckDB.ExtensionKit mirrors this mechanism in C#. It defines a C# representation of the struct (`DuckDBExtApiV1`), where each field is declared as a C# function pointer (`delegate* unmanaged[Cdecl]<...>`). This maps the C ABI directly: calling into DuckDB becomes a simple indirect call through a function pointer field, rather than a delegate invocation with runtime marshaling.

Entrypoint

A DuckDB extension needs to expose an **entrypoint function** following the C calling convention (the entrypoint that should be exported from the binary is the name of the extension plus `_init_c_api`). This way, DuckDB can locate it when the extension is loaded. In the C extension template, this is handled with macros that generate the exported function and the surrounding boilerplate.

DuckDB.ExtensionKit follows the same model, but generates the boilerplate from C# instead of C macros. The source generator emits a native-compatible entrypoint that retrieves the API table (via the `access` object) and performs the required initialization, just like the C template does. The generated method is annotated with `[UnmanagedCallersOnly(EntryPoint = "...")]`, which instructs the .NET toolchain to export a real native symbol with that name and make it callable from C. With .NET Native AOT, this becomes an actual exported function in the produced binary – allowing DuckDB to load and call into the extension exactly as it would for a C implementation.

Native AOT

Finally, Native AOT is what makes this approach practical for DuckDB extensions. Once the extension code and generated sources are compiled, the project is published using .NET Native AOT. This step produces a native binary with no dependency on a managed runtime at load time. The resulting artifact is a native DuckDB extension that can be loaded and executed in the same way as extensions written in C or C++. From DuckDB's perspective, there is no difference between an extension built with DuckDB.ExtensionKit and one implemented in a traditional native language.

Current Status and Limitations

DuckDB.ExtensionKit, just like the C extension template, is currently experimental. The APIs are still evolving, and not all extension features supported by DuckDB are exposed yet.

The toolkit relies on .NET Native AOT, which means extensions need to be built for specific target platforms (for example, `linux-x64`, `osx-arm64`, or `win-x64`). As with other native extensions, binaries are platform-specific and need to be built accordingly.

Build Your Own Extension in C#

DuckDB.ExtensionKit is available as an open-source project on GitHub under the MIT license. The project includes example extensions that demonstrate how to define and build DuckDB extensions in C#. The repository contains a JWT-based example extension that showcases both scalar functions and table functions, as well as the full build and publishing workflow using .NET Native AOT.

Feedback, bug reports, and contributions are welcome through GitHub issues.

Closing Thoughts

DuckDB's extension mechanism has proven to be a flexible foundation for extending the system without complicating the core engine. DuckDB.ExtensionKit explores how this mechanism can be made accessible to a broader audience by leveraging the .NET ecosystem, while still producing native extensions that integrate directly with DuckDB.

Although C# is typically viewed as a high-level language, this project demonstrates that it can also be used to implement low-level, ABI-compatible components when needed. By combining modern C# features with DuckDB's existing extension interface, it is possible to write extensions in a high-level language without giving up control over native boundaries.

Announcing DuckDB 1.5.1

Publication date: 2026-03-23

Author: The DuckDB team

TL;DR: We are releasing DuckDB version 1.5.1, a patch release with bugfixes, performance improvements and support for the Lance lakehouse format.

In this blog post, we highlight a few important fixes in DuckDB v1.5.1, the first patch release in DuckDB's v1.5 line. You can find the complete release notes on GitHub.

To install the new version, please visit the installation page.

Data Lake and Lakehouse Formats

Lance Support

Thanks to the collaboration with LanceDB, DuckDB now supports reading and writing the Lance lakehouse format through the `lance` core extension.

```
INSTALL lance;
LOAD lance;
```

You can write to Lance as follows:

```
COPY (
    SELECT 1::BIGINT AS id, 'a'::VARCHAR AS s
    UNION ALL
    SELECT 2::BIGINT AS id, 'b'::VARCHAR AS s
) TO 'example.lance' (FORMAT lance);
```

And read it like so:

```
SELECT count(*) FROM 'example.lance';
```

```
┌──────────────┐
│ count_star() │
│    int64     │
├──────────────┤
│            2 │
└──────────────┘
```

> Lance support is also available for DuckDB v1.4.4 LTS and v1.5.0.

Iceberg Support

We extended support for Iceberg v3 tables, including:

- the `VARIANT` and `TIMESTAMP_NS` types
- default values
- deletion vectors (delete and update v3 tables)
- inserting into a partitioned table
- creating a partitioned table
- Parquet Copy options support

Configuration Options

The `httpfs` extension has a new setting:

```
SET force_download_threshold = 2_000_000;
```

This will force full file download on any file < 2 MB. The default value is 0, but we may revisit the setting default in the next release.

Fixes

Globbing Performance

There have been reports by users (thanks!) that S3 globbing performance degraded in certain cases – this has now been addressed.

Non-Interactive Shell

On Linux and macOS, DuckDB's new CLI had an issue executing the input received through a non-interactive shell. In practice, this meant that scripts piped into DuckDB were not executed. For v1.5.0, there was a simple workaround available. We fixed the issue in v1.5.1, so there is no need for a workaround.

Indexes

This release ships two fixes for ART indexes. If you are using indexes in your workload (directly or through key / unique constraints), we recommend updating to v1.5.1 as soon as possible.

Landing Page Improvements

We are shipping a new section of the landing page that showcases all the technologies DuckDB can run on... or in! Check it out!

Conclusion

This post is a short summary of the changes in v1.5.1. As usual, you can find the full release notes on GitHub.

DuckDB Now Speaks Dutch!

Publication date: 2026-04-01

Author: Daniël ten Wolde

TL;DR: DuckDB now speaks Dutch! Load the EendDB community extension and start writing your queries in het Nederlands.

Historically speaking, SQL queries have always been formulated in English. The initial name of the language was even Structured **English** Query Language (SEQUEL), before it became SQL. Now, what if the Dutch hadn't traded away New Amsterdam (present-day New York)? Would we all have been writing SQL in Dutch instead?

Well, wonder no longer. Today we're releasing **EendDB**: a DuckDB extension that brings you the **Gestructureerde Zoektaal,** or GZT for short.

Want joins? We've got SAMENVOEGEN. Aggregates? GROEP PER. Window functions? Those work too — though you'll have to look up the Dutch keywords in the repository yourself.

You can try it out right now in DuckDB v1.5.1:

```
INSTALL eenddb FROM community;
LOAD eenddb;
CALL enable_dutch_parser();

MAAK TABEL eend (
    id        GEHEEL_GETAL,
    naam      TEKST,
    leeftijd  GEHEEL_GETAL,
    gewicht   KOMMAGETAL,
    soort     TEKST
);

TOEVOEGEN AAN eend WAARDEN
    (1, 'Donald',  29, 1.2, 'Wilde eend'),
    (2, 'Daffy',   35, 1.5, 'Zwarte eend'),
    (3, 'Daisy',   27, 1.1, 'Wilde eend'),
    (4, 'Scrooge', 75, 1.8, 'Wilde eend');

SELECTEER *
VAN eend
WAARBIJ gewicht > 1.2 EN naam ZOALS '%D%'
VOLGORDE PER leeftijd;
```

id int32	naam varchar	leeftijd int32	gewicht float	soort varchar
2	Daffy	35	1.5	Zwarte eend

Of course, no query language is complete without joins and aggregates. Let's create a second table and count the ducks per *soort:*

```
MAAK TABEL soorten (soort TEKST, leefgebied TEKST);

TOEVOEGEN AAN soorten WAARDEN
    ('Wilde eend',  'Meren en rivieren'),
    ('Zwarte eend', 'Kustgebieden');

SELECTEER s.leefgebied, count(*) ALS aantal_eenden
```

```
VAN eend ALS e
LINKS SAMENVOEGEN soorten ALS s OP e.soort = s.soort
GROEP PER s.leefgebied
VOLGORDE PER aantal_eenden AFLOPEND;
```

leefgebied varchar	aantal_eenden int64
Meren en rivieren	3
Kustgebieden	1

After we are done playing around, we obviously have to clean up after ourselves. Rather than DROP a table, in Dutch we like to throw it away (“weggooien”):

```
GOOI_WEG TABEL eend;
GOOI_WEG TABEL soorten;
```

Under the hood, the parser is using DuckDB's new experimental parser, based on Parsing Expression Grammar.

For more examples, check out the repository on GitHub.

Data Inlining in DuckLake: Unlocking Streaming for Data Lakes

Publication date: 2026-04-02

Author: Pedro Holanda

TL;DR: DuckLake's data inlining stores small updates directly in the catalog, eliminating the "small files problem" and making continuous streaming into data lakes practical. Our benchmark shows 926× faster queries and 105× faster ingestion when compared to Iceberg.

Announcing DuckDB 1.5.2

Publication date: 2026-04-13

Author: The DuckDB team

TL;DR: We are releasing DuckDB version v1.5.2, a patch release with bugfixes and performance improvements, and support for the DuckLake v1.0 lakehouse format.

In this blog post, we highlight a few important fixes in DuckDB v1.5.2, the second patch release in DuckDB's v1.5 line. You can find the complete release notes on GitHub.

To install the new version, please visit the installation page.

Data Lake and Lakehouse Formats

DuckLake

We are proud to release a stable, production-ready lakehouse specification and its reference implementation in DuckDB.

We published a detailed blog post on the DuckLake site but here's a quick summary: DuckLake v1.0 ships dozens of bugfixes and guarantees backward-compatibility. Additionally, it has a number of cool features: data inlining, sorted tables, bucket partitioning, and deletion buffers as Iceberg-compatible Puffin files. More on this in the announcement blog post.

Iceberg

The Iceberg extension ships a number of new features. It now supports the following:

- `GEOMETRY` type
- `ALTER TABLE` statement
- Updates and deletes from partitioned tables
- Truncate and bucket partitions

Last week, DuckLabs engineer Tom Ebergen gave a talk at the Iceberg Summit titled "Building DuckDB-Iceberg: Exploring the Iceberg Ecosystem", where he shared his experiences with Iceberg.

Preliminary Jepsen Test Results

To make DuckDB as robust as possible, we started a collaboration with Jepsen. The preliminary test suite is available at https://github.com/duckdb/duckdb-jepsen.

The test suite has uncovered a bug that was triggered by `INSERT INTO` statements that perform conflict resolution on a primary key, and already shipped a fix in this release.

New Online Shell

The online WebAssembly shell at `shell.duckdb.org` received a complete overhaul. A highlight of the new shell is the ability to store and list files using the `.files` dot command and its variants.

Using the file storage feature, you can turn your browser session into workbench: you can drag-and-drop files from your local file system to upload them, create new ones using DuckDB's COPY ... TO statement and download the results. For more information on this feature, use the `.help` command.

The new shell comes with a few built-in datasets: you're welcome to try them out and experiment. Your old links to `shell.duckdb.org` should still work but if you experience any problems, please submit an issue in the duckdb-web repository.

Benchmarks

We benchmarked DuckDB using the Linux v7 kernel on an r8gd.8xlarge instance with 32 vCPUs, 256 GiB RAM, and an NVMe SSD. We first ran the scale factor 300 test on Ubuntu 24.04 LTS, then upgraded to Ubuntu 26.04 beta. We noticed that the composite TPC-H score shows a **~10% improvement**, jumping from 778,041 to 854,676 when measured with TPC-H's QphH@Score metric.

Coming Up

This quarter, we have quite a few exciting events lined up.

DuckCon #7. On June 24, we'll host our next user conference, DuckCon #7, in Amsterdam's beautiful Royal Tropical Institute.

AI Council Talk. On May 12, DuckDB co-creator Hannes Mühleisen will give a talk at AI Council 2026 titled "Super-Secret Next Big Thing for DuckDB". Well, at this point, we cannot tell you more than he will present the super-secret next big thing for DuckDB. But, if you cannot make it, don't worry: we'll publish the presentation afterwards.

Ubuntu Summit Talk. We already talked about performance on Ubuntu. In late May, Gábor Szárnyas of DuckLabs will give a talk titled "DuckDB: Not Quack Science" at the Ubuntu Summit.

Conclusion

This post is a short summary of the changes in v1.5.2. As usual, you can find the full release notes on GitHub.

DuckLake v1.0: The Lakehouse Format Built on SQL Reaches Production-Readiness

Publication date: 2026-04-13

Author: The DuckDB team

TL;DR: We released the DuckLake v1.0 standard!

The DuckLake Spec Is so Simple, Even a Clanker Can Build One for Dataframes

Publication date: 2026-05-04

Authors: Pedro Holanda, Dr. Peter van Holland

TL;DR: We are showcasing the simplicity of DuckLake's v1.0 specification by developing a dataframe reader/writer with AI.

Delta Grows Up: Writes, Unity Catalog and Time Travel

Publication date: 2026-05-07

Author: Ben Fleis

TL;DR: DuckDB's Delta and Unity Catalog extensions shed their experimental tags — now with writes, Unity Catalog and time travel support.

Welcome back! While we here at DuckLabs are typically of the quacking persuasion, we've been busy as beavers, shoring up our Delta to prepare for what's next… Unity Catalog! Let's look at how DuckDB's Delta and Unity Catalog extensions have grown up enough to shed the experimental tag, and see what has changed since our last update.

Time to Open the Delta

Before we jump in, let's review briefly. Delta is a foundational open table format and toolset for building and managing data lakes, related to Iceberg and other lakehouse formats. DuckDB supports Delta tables via its Delta Extension.

In that last update we highlighted performance wins, particularly file skipping via filter pushdowns, and metadata caching with snapshot pinning. Now we build on these, and add writes, time travel and Unity Catalog support, plus more performance gains!

Building Up the Delta (Lake): Writes

What fun are reads without writes? The big addition since we last chatted is `INSERT` support! It works as simply as you expect. Let's assume you have a Delta table ready to go. `INSERT` away, it's that simple:

```
-- Schema: (text VARCHAR, code BIGINT)
ATTACH './path/to/my_table' AS my_table (TYPE delta);

INSERT INTO my_table
VALUES ('Question 2', 2), ('The Answer', 42);

-- Bulk insert from a query
INSERT INTO my_table
FROM (SELECT text || ' (copy)', code + 100 FROM my_table);
```

Also worth calling out – multiple `INSERT`s within a `BEGIN` / `COMMIT` block are stored as a single Delta version: one atomic commit, one new log entry. And, as you'll see later, this works with catalogs too! `UPDATE`, `MERGE`, and `DELETE` are not yet supported, but on our future work list.

Time Travel

DuckDB's Delta extension now supports time travel. Any Delta table can be queried as of a particular version. DuckDB supports binding to a specific `VERSION` either at `ATTACH` time, or as part of an individual query.

Let's assume that we have built up the above `my_table` incrementally, with versions 0, 1, and 2 containing:

Version	Contents
0	`('Question 1', 1)`

Version	Contents
1	+('Question 2', 2),('The Answer', 42)
2	+('Question 1 (copy)', 101),('Question 2 (copy)', 102),('The Answer (copy)', 142)

You can attach normally and query arbitrary versions inline as needed. The most flexible approach:

```
ATTACH './path/to/my_table' AS my_table (TYPE delta);

SELECT count() FROM my_table AT (VERSION => 0); -- 1  (Question 1 only)
SELECT count() FROM my_table AT (VERSION => 1); -- 3  (after 1st insert)
SELECT count() FROM my_table;                   -- 6  (latest)
```

Or attach, pinned to a specific version, which is useful when you want a stable reference that never changes, regardless of future writes:

```
-- Always v1, no matter what gets written later
ATTACH './path/to/my_table' AS my_table_v1
    (TYPE delta, VERSION 1);

SELECT count() FROM my_table_v1;      -- → 3

-- Locked to whatever was latest at attach time
ATTACH './path/to/my_table' AS my_table_pinned
    (TYPE delta, PIN_SNAPSHOT);

SELECT count() FROM my_table_pinned;  -- → 6
```

Growing Up: No Longer a Kit

The DuckDB Delta extension is no longer a kit and has grown up quite a bit since a year ago. As you just saw, we added writes and time travel. These features open the door to something bigger: Unity Catalog coordination.

Unity Catalog Support atop the Delta

Data lake systems excel at scale. As your data assets multiply, you need a way to discover what exists, control who can access it, audit how it's being used, and coordinate writes across multiple engines. Data catalogs have evolved to address exactly these needs, sitting above the storage layer to manage the metadata, governance, and transactional bookkeeping that make large-scale data lakes effective. The OSS Unity Catalog team has a good overview if you'd like to go deeper; the concepts apply broadly regardless of which catalog you use.

What is Unity Catalog?

Unity Catalog (UC for short) is an open standard for governing data and AI assets, including tables, volumes, models, and functions, across engines and clouds. It turns your data lake into a lakehouse, and gives you a single place to discover, audit, and control access to your data, regardless of what's reading or writing it. DuckDB's Unity Catalog extension is built upon the Unity Catalog Open API. There are two main implementations: OSS Unity Catalog, which you can self-host (and Docker-ify in minutes), and Databricks Unity Catalog, the managed version. Like Delta, the DuckDB Unity Catalog extension has shed its experimental tag. Let's put both to work.

Getting Started: OSS Unity Catalog

We've set up a Docker image playground bundling OSS Unity Catalog and DuckDB together, so you can follow along with easy docker build-and-run setup. Grab it if you would like to walk through the samples or experiment on your own. (If you'd prefer to run OSS UC directly, the official image is the upstream of our playground.)

Let's start with Docker. Assuming you now have the image running, it already executed (roughly) the following steps in the build phase to prepare our playground:

```
# Create a schema
/home/unitycatalog/bin/uc schema create --catalog unity --name my_schema

# Create the "pets" table
/home/unitycatalog/bin/uc table create \
    --full_name        unity.my_schema.pets \
    --columns          "uuid STRING, name STRING, age INT, adopted BOOLEAN" \
    --format           DELTA \
    --storage_location file:///home/unitycatalog/etc/data/external/unity/my_schema/tables/pets
```

After that, we can test things out from DuckDB. To see for yourself, `docker exec -it duckdb-playground duckdb` will give you a DuckDB shell inside the container.

Before doing anything meaningful we'll need to set up a DuckDB secret. In this example the TOKEN value is ignored by local OSS UC server, but the field is required. Create the secret, then you can immediately attach and read:

```
LOAD unity_catalog;

CREATE SECRET (
    TYPE     unity_catalog,
    TOKEN    'demo-ignored-token',
    ENDPOINT 'http://unitycatalog:8080'
);

ATTACH 'unity' AS my_catalog
    (TYPE unity_catalog, DEFAULT_SCHEMA 'my_schema');

SELECT name, age, adopted FROM my_catalog.pets ORDER BY name;
-- returns a single 'Seed' row
```

That's it! You just queried Unity-Catalog-managed, Delta-stored pets data.

> **Tip.** Want to experiment with this on Databricks Unity Catalog? Setting up a Databricks Unity Catalog is out of scope for this blog, but if you have one ready to go, you will need these to get bootstrapped with DuckDB:
>
> - set ENDPOINT to your Workspace URL (typically: https://%7Binstance%7D.cloud.databricks.com/)
> - set TOKEN appropriately (e.g. create a PAT with `unity-catalog` scope); getting the correct token depends entirely on your setup. To dive in, see Access Control in Unity Catalog.
>
> With these in hand you can use DuckDB directly, or access the extensive UC Open API directly.

Next, let's complete the circle and write some data into our pets table:

```
INSERT INTO my_catalog.pets
    (uuid, name, age, adopted)
SELECT
    gen_random_uuid()::VARCHAR,
    ['Luna', 'Milo', 'Bella', 'Charlie', 'Max', 'Lucy', 'Cooper',
     'Daisy', 'Buddy', 'Lily', 'Rocky', 'Molly', 'Bear', 'Lola',
     'Duke', 'Sadie', 'Tucker', 'Zoe', 'Oliver', 'Stella'
    ][1 + (random() * 19)::INT],
    (1 + (random() * 14)::INT)::INT,
    random() > 0.5
FROM range(10);

SELECT count() FROM my_catalog.pets;
```

You can also easily find and see the created files; check the local `data` directory (also bind-mounted in Docker), and you should find both pre-existing files, and a new Parquet file containing the inserted rows. In my case it looks like this:

```
tree data
```

```
data
└── external
    └── unity
```

```
        └── my_schema
            └── tables
                └── pets
                    ├── _delta_log
                    │   ├── 00000000000000000000.json
                    │   ├── 00000000000000000001.json
                    │   └── 00000000000000000002.json
                    ├── duckdb-19cb47ae-9f35-4126-b67d-c94fcade68cc.parquet
                    └── duckdb-e3bb0336-f16a-4d21-9495-0fbf55c6cba8.parquet

7 directories, 5 files
```

Catalog Managed Tables

With the basics out of the way, we can talk about Catalog Managed Tables (CMT). This is available today in both OSS and Databricks Unity Catalog.

The big feature in CMT is Catalog Commits, which enables coordinated concurrent writes. Without Catalog Commits, DuckDB writes go directly to the Delta log. While modern storage backends prevent outright lost writes, UC is left out of the loop entirely. Its metadata, audit trail, and statistics fall out of sync with the actual table state, and other engines querying through UC may see a stale view.

Catalog Commits (CC) fixes this: every write is staged and registered through UC before it becomes visible. UC acts as the commit arbiter, preserving first writer commits, and sending a conflict error to later writers. This matters wherever multiple writers are appending simultaneously, e.g., parallel ETL pipelines, partitioned bulk loads, and concurrent analytical inserts. Each writer works independently; UC ensures exactly one commit lands per version and keeps its own catalog in sync with every one of them.

Consistent reads and audit history are already inherent to Delta and UC respectively. CC doesn't add functionality, it just ensures UC stays in sync with every commit. And Catalog Commits coordinate per table; there is no cross-table atomicity. If you write to two tables in the same `BEGIN` / `COMMIT` block, each table commits independently.

To opt a table into CMT (and therefore CC), set the `delta.feature.catalogManaged` table property at creation time. This is done via Spark or the UC CLI, as DuckDB's Unity Catalog extension does not yet support `CREATE TABLE` DDL:

```
-- Via Spark
CREATE TABLE my_catalog.my_schema.concurrent_tbl (
    uuid    STRING  NOT NULL,
    name    STRING  NOT NULL,
    age     INT     NOT NULL,
    adopted BOOLEAN NOT NULL
)
TBLPROPERTIES ('delta.feature.catalogManaged' = 'supported');
```

Once enabled, DuckDB writes go through UC's commit staging automatically — the `INSERT` syntax is unchanged:

```
INSERT INTO my_catalog.my_schema.concurrent_tbl
    (uuid, name, age, adopted)
VALUES (gen_random_uuid()::VARCHAR, 'Luna', 3, true);
```

Now each DuckDB writer stages its commit to a `_staged_commits/` directory and registers it with UC before that data becomes visible. UC arbitrates: exactly one writer wins each version in a race, the others get a conflict error and can retry. Next, let's look at how UC handles the race.

Deeper Dive

Racing Commits

To see how Catalog Commits arbitrates, we launched 20 concurrent DuckDB writers, 8 at a time, all inserting into the same managed table:

```
seq 1 20 | xargs -P 8 -I{} scripts/unity/05-cmc/write-single {}
```

```
[worker 6] OK - inserted 5 rows
[worker 5] CONFLICT - another writer won this version, retry needed
[worker 2] CONFLICT - another writer won this version, retry needed
[worker 8] CONFLICT - another writer won this version, retry needed
[worker 7] CONFLICT - another writer won this version, retry needed
[worker 3] CONFLICT - another writer won this version, retry needed
[worker 1] OK - inserted 5 rows
[worker 4] CONFLICT - another writer won this version, retry needed
[worker 16] OK - inserted 5 rows
[worker 13] CONFLICT - another writer won this version, retry needed
[worker 15] CONFLICT - another writer won this version, retry needed
[worker 11] CONFLICT - another writer won this version, retry needed
[worker 14] CONFLICT - another writer won this version, retry needed
[worker 12] OK - inserted 5 rows
[worker 9] CONFLICT - another writer won this version, retry needed
[worker 10] CONFLICT - another writer won this version, retry needed
[worker 17] CONFLICT - another writer won this version, retry needed
[worker 20] CONFLICT - another writer won this version, retry needed
[worker 18] OK - inserted 5 rows
[worker 19] CONFLICT - another writer won this version, retry needed
```

Here we see 5 successful writes, and 15 signaled conflicts. Let's confirm in the data:

```
SELECT count() AS total_rows FROM my_catalog.my_schema.concurrent_tbl;
```

total_rows
int64
35

10 seeded rows + (5 writes × 5 rows each) = 35 total rows. (In a real workload, you would retry the conflicted writes and land all 20 inserts.) Catalog Managed Table commits gave us clear signal and semantics during highly concurrent writes, as promised.

Travel in Time, Faster

DuckDB's Delta snapshot loading is getting a speed boost: snapshots will load incrementally when possible, making time travel across nearby versions significantly faster. Consider a table where some initial queries are made against version 16:

```
ATTACH './path/to/table' AS t (TYPE delta, VERSION 16);
SELECT count() FROM t;  -- → 17
```

And now some work needs to be done against version 20. If we peek under the hood (warning: sneaky code follows), we'll see that none of the previously loaded Delta log metadata files were re-loaded:

```
SET enable_logging = true;
SET delta_kernel_logging = true;
CALL enable_logging('DeltaKernel', level = 'trace');

ATTACH './path/to/table' AS t (TYPE delta, VERSION 20);
SELECT count() FROM t;  -- → 21

-- Delta kernel logs 'Provisionally selecting ... <version>.json'
-- whenever it reads a log file from scratch. We search for any such
-- message referencing a zero-padded log filename; zero matches
-- means the cached v16 snapshot was reused rather than rebuilt.
SELECT count() FROM duckdb_logs
WHERE type = 'DeltaKernel'
  AND message LIKE '%00000000000000000%.json%';
-- → 0
```

In Delta lakes with thousands or millions of snapshots, incremental loading provides a big win when working across multiple versions.

> At time of writing, incremental snapshot loading is supported in nightly builds. You can install it using:
>
> ```
> FORCE INSTALL delta FROM core_nightly;
> ```
>
> Please be aware that nightly builds are not intended for production use. The implementation will be included in the next stable release, v1.5.3.

Conclusions

A year ago, DuckDB could read Delta tables. Today it can insert data into them, travel through their history, and query and write through a governed catalog — without the experimental caveat on any of it. The combination of Delta for open storage, Unity Catalog for governance and coordination, and DuckDB for fast analytical queries is a stack you can build on.

There's more to come: DDL support to create and manage tables directly, delete/update/merge support, and multi-table atomicity for writes that span more than one table. In the meantime, the playground image linked above has everything you need to kick the tires. As always, feedback and bug reports are welcome on GitHub.

Announcing the Program of DuckCon #7 Amsterdam

Publication date: 2026-05-08

Author: Gábor Szárnyas

TL;DR: We are hosting DuckCon #7 in Amsterdam on June 24, 2026. Join us at the Royal Tropical Institute for talks, lightning sessions, and a borrel.

We are excited to announce the program of **DuckCon #7 Amsterdam**, DuckDB's user conference. The event will be held on **Wednesday, June 24, 2026**, at the Royal Tropical Institute. The program runs from **15:00 to 20:00 CEST**.

See the registration link and the full program on the DuckCon #7 event page.

Quack: The DuckDB Client-Server Protocol

Publication date: 2026-05-12

Author: The DuckDB team

TL;DR: DuckDB instances can now talk to each other using the Quack remote protocol. This lets you run DuckDB in a client-server setup with multiple concurrent writers. In DuckDB's spirit, Quack is simple to set up and builds on proven technologies such as HTTP. It's also fast, which allows it to support workloads ranging from bulk operations to small transactions.

Background: Database Architectures

When databases first emerged, there was no distinction between a 'client' and a 'server', the whole database just ran on a single computer. Somewhere in the 80s, Sybase was the first to introduce the concept of a database 'server' and a 'client' running on different computers. Ever since, it was just assumed that every database system used a client-server architecture along with a communication protocol to talk between those. This was convenient, because the single mutable state stays in a single place under the control of a server, and there can be many clients at the same time reading and writing data. There are of course drawbacks to this method, most notably, those protocols can add a significant amount of overhead. If you're curious to read more, we wrote a research paper on database protocols a while back.

Of course, there were always dissenters to the client-server architecture, most notably the ubiquitous SQLite in 2000, and of course DuckDB, first released in 2019. We made quite a lot of noise about implementing an in-process architecture, where there is no client-server, no protocol, just low-level API calls. This worked really well for interactive use cases in e.g., data science, where analysts would interact with their data for example in a Python notebook and their data was managed in a DuckDB instance running in the very same process. It also worked really well for the many use cases where DuckDB was just "glued" to an existing application to provide SQL functionality on data living in that application.

Being an in-process system works "less well" for use cases when trying to modify the same database file from multiple processes at the same time. There are a lot of use cases where this is relevant, for example, when inserting into the same database from a bunch of processes collecting telemetry while at the same time querying the same tables to drive a dashboard. There are very good technical reasons why we could not make this work, most notably, the fact that DuckDB keeps a bunch of state in main memory and would have to synchronize that state if multiple processes start making changes simultaneously.

And yes, there were workarounds. Of course you can whip up a custom Remote Procedure Call (RPC) solution where there is a process that holds the DuckDB database instance and offers a service to other processes to query and insert data. There are also multiple projects out there that retrofit client-server abilities to DuckDB, for example using the Arrow Flight SQL protocol. MotherDuck has their own custom client-server protocol. And of course, you can always (gasp) switch to a more traditional database system that had client-server support, for example the also-ubiquitous PostgreSQL. You can then even proceed to run a so-called "EleDucken", DuckDB in said PostgreSQL using one of the various extensions out there that enable this, for example pg_duckdb.

The vast number of workarounds people built to bolt a client-server solution onto DuckDB has at the very least convinced us that this is something people cared about. We see DuckDB as a universal data wrangling tool. If this means having a client-server protocol in addition to the in-process capabilities – fine. If this ends up unlocking a vast new set of cases in which DuckDB can be useful – excellent! In the end we care deeply about user experience and perhaps less about having the last word on architecture. So we bit the bullet, eventually, finally, and today we are very happy to announce the result:

Introducing the Quack Protocol for DuckDB

What do two (or more) ducks do if they want to talk to each other? They quack! So it is quite natural that we need to call the protocol that two DuckDB instances can use to talk to each other "Quack", too! We had the opportunity to design a database protocol from scratch

in 2026 without having to consider any legacy, which is quite a luxury. We were able to learn from the existing protocols, including the more recent Arrow Flight SQL and others. Before we dive into how Quack works internally, let's see how it works from a user perspective. First, you need two DuckDB instances. That's right, DuckDB will act both as a client and as a server! The two instances can be on different computers a world apart (or in space) or just two different terminal windows on your laptop. First, we need to install the Quack extension in both DuckDB instances:

This should show the content of the remote table hello, `world` in DuckDB #2. Witchcraft! We can also copy data from the local instance to the remote one:

Similarly, you should see `world2` in the output on DuckDB #1. Obviously those are the most basic examples we can think of. Tables can be much more complex, queries can be much more complex, data volumes can be quite vast (see below). There is also a way to just ship an entire verbatim query to the remote side using the `query` function, which is better for very complex queries on large datasets and offers more control over what exactly is executed remotely:

Of course there is much more to see here. Please consult our documentation for more details.

Protocol Design

HTTP-Based

Quack is built straight on the venerable HTTP, the Hypertext Transfer Protocol. From its humble beginnings at CERN, HTTP has become a de-facto protocol layer on top of TCP and all the stuff below. The entire stack is optimized to transmit HTTP message streams efficiently. The protocol has surprisingly low overhead if implemented properly. Everyone and their little brother knows how to deal with HTTP in load balancing, authentication, firewalls, intrusion detection etc. It would be rather misguided not to build a database protocol on top of HTTP in 2026. HTTP also allows the DuckDB-Wasm distribution to speak Quack natively! So DuckDB running in a browser can e.g., directly connect to a DuckDB instance running in an EC2 server using Quack.

Request-Response Pattern

Interactions on Quack are always driven by the client in a request-response pattern. Quack messages are for example connection requests, to authenticate with a token as seen above. See below on how authentication and authorization are handled in detail. Subsequent messages are requests to execute a query and return the first part of the response and follow-up fetch messages to retrieve large results, possibly from multiple threads in parallel.

Serialization

Requests and responses are encoded using the new MIME type application/duckdb. This encoding leverages DuckDB's internal efficient serialization primitives for complex structures like data types and result sets. We have been using the same primitives for example in our Write-Ahead Log (WAL) files for years, meaning they are fairly well-optimized and battle-tested.

Encryption

While we want Quack to "just work" we also are wary of the security nightmares of attaching a database directly to the evil internet, as has happened before. This is why Quack will by default generate a random authentication token at server start-up, which then has to be given to the client. In addition, the Quack server will by default only bind to localhost (which can of course be overridden). Quack does not use SSL by default, because it is a bit silly to bring all that infrastructure and add dependencies just for localhost communication. We do not recommend opening up a DuckDB Quack endpoint directly to the Internet. Instead we strongly recommend that you use a common HTTP endpoint like nginx if you should choose to expose Quack to the World Wide Web and have that proxy terminate SSL (e.g., with Let's Encrypt). The Quack client will assume SSL is enabled for non-local connections, this can be overridden. We provide a guide for this in our documentation.

Round-Trips

We have been careful to optimize the number of protocol round trips or request/response pairs for queries. Once connected, a query can be completely handled with a single round trip. This is a critical optimization for latency-sensitive environments. At the same time, we have seriously optimized Quack for efficient bulk response transfer. As far as we know, Quack is currently the fastest way to shove tables through a socket, and millions of rows can be transferred in a few seconds. Below are a few benchmark results.

Authentication and Authorization

Authentication and authorization of database queries are an endless source of joy and complexity. We are likely unable to capture everyone's use case, certainly not in a first release. The smart thing is therefore not to try. For Quack, we have chosen an auth model that ties into DuckDB's philosophy of extensibility. There are hundreds of DuckDB extensions out there already. Quack ships with a default Authentication method and no authorization restrictions, but both can be overridden by user-supplied code. As you have seen above, the Quack server generates a default random authentication token on startup. When a client connects, it provides an authentication string. The server side will call an authentication callback. By default, it will compare the client-supplied token with the one that was randomly generated before. But this callback can be changed through configuration! You can bring your own authentication function that for example queries an LDAP directory, reads a text file, or just rolls the dice. Up to you. Similarly, the authorization function can be changed. The default authorization function just says "yes" to everything, but you can inspect each query a client attempts to execute, correlate the query to the previously used authentication string etc. Those callbacks can even be plain SQL macros! Please see our documentation for more details.

Default Port

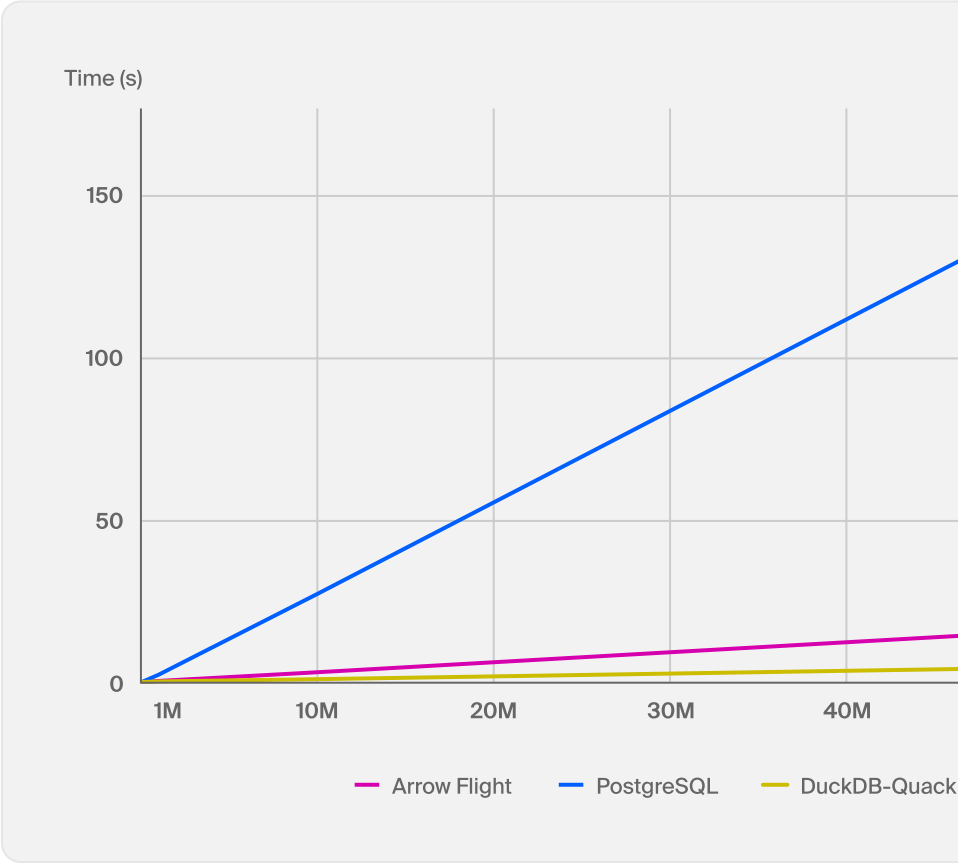
By default, a Quack server listens on port `9494`, the number 94 being easy to remember as the year Netscape Navigator was released.

Benchmarks

We have set up two benchmarks to showcase the Quack protocol. Those benchmarks were run on AWS virtual machines running Ubuntu on Arm. We picked the m8g.2xlarge instance type, which has 8 vCPUs and 32 GB of RAM and, importantly, "up to 15 Gbps" network bandwidth. We recreated a real-world scenario where client and server are in the same data center, but on different machines. We made sure both instances were in the same "availability zone". Ping time between the instances averaged around 0.280 ms.

Bulk Transfer

The first benchmark tests bulk transfer, the case where a fairly large number of rows should be transferred over the database protocol. If you've read the paper we linked above, you know that this is a case where traditional database protocols were struggling. We compare Quack with two systems: the widespread PostgreSQL protocol and the newer Arrow Flight SQL protocol. Arrow Flight is provided by the GizmoSQL server that also uses DuckDB internally. We transfer an increasing number of rows of the TPC-H lineitem table, all the way up to a whopping 60 million rows (76 GB in CSV format!) and report the median wall clock time over 5 runs. We expect the modern bulk-oriented protocols to far outclass the PostgreSQL protocol. Here are the results:



Runtimes of bulk transfer operations (lower is better)

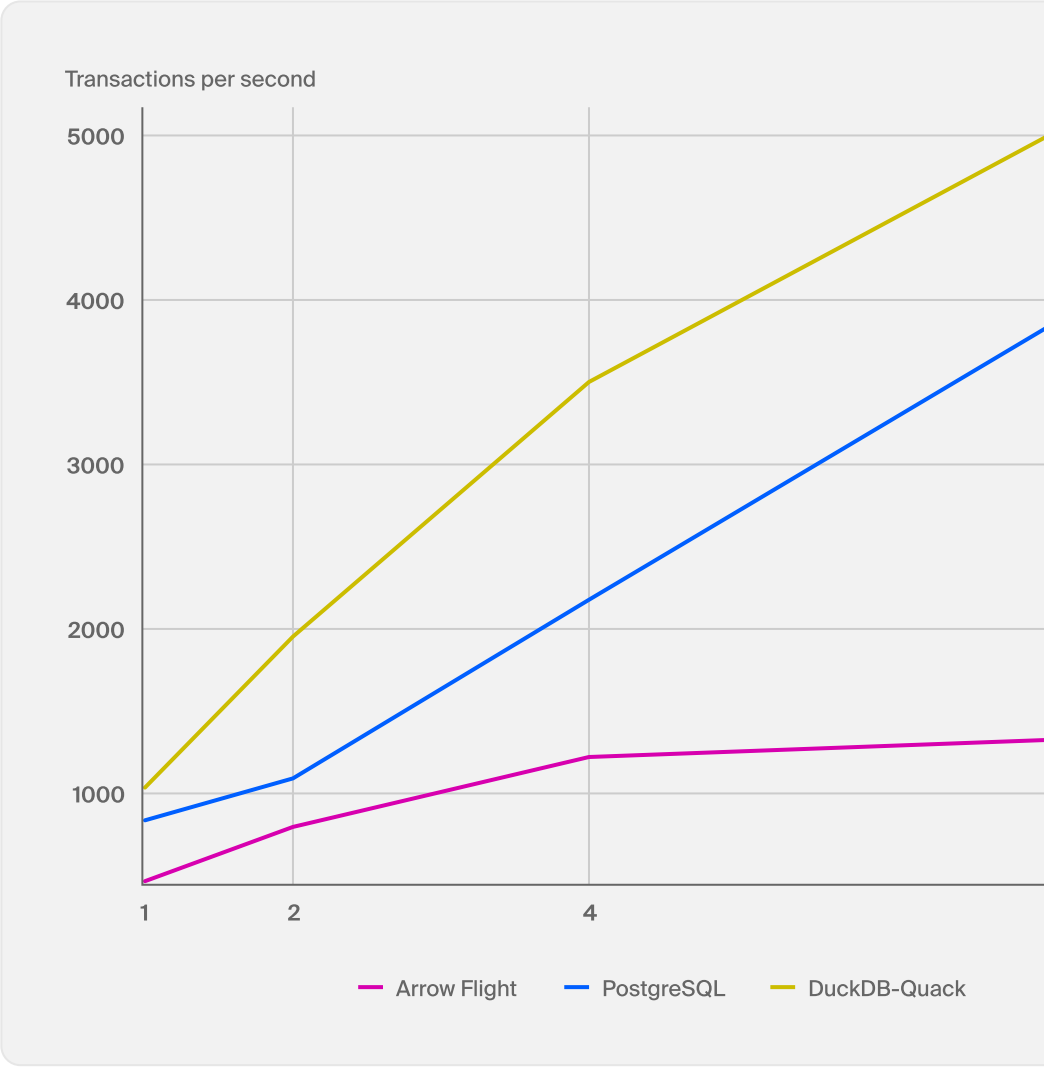
We can see how Quack is doing great for bulk result set transfer, transferring the 60 million rows in under 5 seconds! Even the purpose-built Arrow Flight SQL protocol can't compete here, and Postgres' row-based protocol is rather hopeless in general.

In fairness we have to mention that the standard PostgreSQL clients do not parallelize reads over multiple threads, but Quack and Arrow can. Shameless plug: DuckDB's PostgreSQL client can also do that in some cases!

Small Writes

The second benchmark tests small appends. This is a common use case to, for example, centralize observability data in a single central DuckDB instance. This stresses the database protocol in a different way, for example, multiple round trips between client and server to complete a single transaction will be a disadvantage. We test this by creating an empty table with the same structure as the TPC-H lineitem table, and then insert randomized values into it, each row in its own `INSERT` transaction. The inserted values somewhat follow the distribution of the usual benchmark generator. We ran an increasing amount of parallel threads for five seconds. We repeated this experiment five times and reported the median transactions per second.

We expect a highly transaction-optimized system like PostgreSQL to dominate this benchmark. We also expect the bulk-optimized Arrow Flight to not do particularly well.



Throughput of small writes (higher is better)

Quite surprisingly, we see Quack outperforming PostgreSQL up to 8 parallel threads to a maximum transaction rate of around 5,500 transactions per second. Beyond that, we hit a current limitation of DuckDB itself in concurrent insertions per second into the same table. PostgreSQL scales better here, which is something to look into for us in the near future. Arrow Flight is not doing too well, being roughly half as fast as Postgres, as expected.

Benchmark scripts are available online.

Conclusion

Today we released Quack, a client-server protocol for DuckDB along with an initial implementation as a DuckDB extension. Quack unlocks a full multiplayer experience with DuckDB, where multiple separate processes – locally or remote – can now modify contents of tables in parallel without locking each other out. And while part of this could also already be achieved with DuckLake, Quack makes this far simpler and provides far higher performance.

Use Cases

With Quack, DuckDB can now be useful in a wide range of new use cases, where centralizing state is more important than hyper-local querying. We have already had to learn that data is not always local with the rise of data lakes. Speaking of lakes, Quack is also going to be integrated into DuckLake so that DuckDB itself can be a remotely-accessible Catalog server. This will unlock new capabilities, e.g., for data inlining. If you have more questions on this, please consult the Quack FAQ.

Overall, DuckDB is moving further out of its initial niche of an in-process database for interactive analytics into a core building block of modern data architecture. We have been playing with Quack for a while now, and are quite excited to hear what you are going to build with it. If you have any suggestions on how Quack could be improved, let us know! And hey, the MythBusters have already proven that a duck's quack echos, so let's see what kind of noise this leads to.

Next Steps

There are of course a lot of things still to do. First off, we are going to integrate Quack into DuckLake, so that it becomes possible to use a remote DuckDB server as a DuckLake catalog! We expect this to greatly improve performance, especially with inlining. Next, we are going to polish Quack over the coming months and release a first production release together with DuckDB v2.0 when it's coming in fall this year. We plan for example to enable auto-installation and auto-loading of the Quack extension whenever it is needed. Using our new parser, we are also planning to improve on the syntax for talking to remote SQL databases from DuckDB. On the core DuckDB side, we plan to work on greatly increasing the transactions per second achievable, so we can scale transactions far beyond eight parallel threads.

Further on, we are thinking about allowing extensions to the Quack protocol beyond authentication and authorization, for example, by allowing DuckDB extensions to add new protocol messages and the code to handle them. And we are also thinking about adding a replication protocol on top of Quack so that changes to a DuckDB instance can be replicated to other servers, for example to set up a cluster of read replicas.

If you want to learn more about Quack – and hear about its initial adoption – join our community conference, DuckCon #7, on June 24. DuckCon will start with the "State of the Duck" talk presented by the co-creators of DuckDB. You can either join in-person or watch the online stream on YouTube.

PS: We have a separate page for the Quack project, make sure you give it a visit.

Acknowledgements

We would like to thank Boaz Leskes from MotherDuck for sharing their lessons learned from building the MotherDuck protocol with us. We also want to thank Philip Moore from GizmoSQL / GizmoData, who has blazed this trail for us already and shown that client-server DuckDB is a very worthwhile thing.

Appendix: Why Not Arrow Flight SQL?

We also have to address one of the few elephants in the room: why on earth did we not use the existing Arrow Flight SQL protocol? It's there. It's available. There are existing implementations. We see the value in Arrow and related projects like ADBC: they are interchange APIs like ODBC and JDBC before them aimed at reducing friction in exchanging data between systems. And that works pretty well.

However, we are also wary of using interchange formats like Arrow inside DuckDB. And while DuckDB's internal structures for query intermediates are in some ways close to Arrow, in other ways they are quite different. We feel that in order to be able to keep innovating in data systems, we cannot allow ourselves to be restricted by formats that are controlled externally. This is why we use our own serialization in Quack. If we want to add a new data type or protocol message, we can ship tomorrow.

Deep down, there is also one fateful design decision in Arrow Flight SQL: every single query requires at least two protocol round trips, `CommandStatementQuery` and `DoGet`. This is not ideal for small updates like in our second experiment above, especially in higher-latency environments. As mentioned, we designed Quack to be able to do single-round trip query execution and result fetching for small queries.

DuckDB 1.5.3: Not an Ordinary Patch Release

Publication date: 2026-05-20

Author: The DuckDB team

TL;DR: We are releasing DuckDB version v1.5.3. While updates in DuckDB itself are limited bugfixes, the upgraded extensions shipped with v1.5.3 bring a ton of new features. These include the Quack client-server protocol, which is now available as a core extension, support for Quack in DuckLake, and several new features for Iceberg, AWS and HTTPS.

In this blog post, we highlight a few important features shipped in DuckDB v1.5.3, the third patch release in DuckDB's v1.5 line. You can find the complete release notes on GitHub.

To install the new version, please visit the installation page.

What's New

While DuckDB v1.5.3 is a patch release, its extensions brings various new features. We list these below.

Quack as a Core Extension

On May 12, we introduced Quack, our new remote protocol that turns DuckDB into a client-server database. If you are new to Quack and don't know where to start, check out the following resources:

- For a high-level overview, see the Quack explainer page.
- For the rationale and history behind Quack, along with an introduction of the protocol and its features, see the announcement blog post.
- For the reference manual and setup guide, check out the Quack documentation.

Starting from DuckDB v1.5.3, we ship Quack as a core extension. This means that you can now start using Quack right away from any client running DuckDB: it will be transparently autoinstalled and autoloaded on first use.

Please note that Quack is still in beta state and breaking changes may happen in the protocol, in function names, etc. We plan to release the production-ready version of Quack together with DuckDB v2.0 in fall 2026.

DuckLake with Quack

DuckLake now supports DuckDB with Quack as its catalog database (ducklake#1151). Let the example speak for itself!

AWS Extension Features

The AWS extension now supports the `web_identity` chain type for IAM Roles for Service Accounts (IRSA) support. This was made possible through a contribution by community member Marcel Steinbach (@mst).

The AWS extension now also supports IAM authentication for managed PostgreSQL databases running on RDS/Aurora. For more details, see the AWS RDS IAM Authentication section in the documentation.

HTTP_PROXY Variable for the HTTPS Extension

Setting the HTTP_PROXY environment variable now sets the `http_proxy` DuckDB configuration option (duckdb#22541). This option makes sure that extensions installs are also passing through the proxy, which may come in handy in e.g. environments that use firewalls.

Note that since the introduction of `curl` into DuckDB's network stack, `curl` automatically uses HTTP_PROXY and HTTPS_PROXY, so now implicitly also DuckDB handles those parameters when the `httpfs` extension is loaded with the default `curl` backend.

Iceberg

The DuckDB-Iceberg extension has shipped a number of features between DuckDB v1.5.2 and v1.5.3. Most notably:

- `MERGE INTO` is now supported for Iceberg tables (iceberg#788)
- The `INSERT` and `UPDATE` statements are now supported on partitioned Iceberg tables with a `truncate` or `bucket` transform (iceberg#879)
- CTAS statements in DuckDB-Iceberg using ADBC are now possible (iceberg#974)
- We added the `iceberg_schema_properties`, `set_iceberg_schema_properties`, and `remove_iceberg_schema_properties` functions to allow getting, setting, and removing Iceberg schema properties (iceberg#960)
- `ALTER TABLE` support has been added for Iceberg tables (iceberg#932, iceberg#928, iceberg#924, iceberg#912, iceberg#904, iceberg#853, iceberg#985, iceberg#981)
- Support for the `GEOMETRY` type has been added for Iceberg tables (iceberg#968, iceberg#902)

Development and Internals

Shipping jemalloc as a Statically Linked Library

The jemalloc allocator is now part of core DuckDB (duckdb#22603) as a static third-party library which is included and linked by default on Linux. Previously jemalloc was a statically-linked extension – the new packaging is cleaner since other DuckDB extensions can be loaded dynamically.

`DISABLE_EXTENSION_LOAD` Flag

The `DISABLE_EXTENSION_LOAD` compile-time flag was fixed in duckdb#22019. When compiling DuckDB with this flag, loading extensions is disabled.

Coming Up

We have two events coming up in the next few weeks:

DuckCon #7. On June 24, we'll host our next user conference, DuckCon #7, in Amsterdam's beautiful Royal Tropical Institute.

Ubuntu Summit Talk. Next week, Gábor Szárnyas of DuckLabs will give a talk titled "DuckDB: Not Quack Science" at the Ubuntu Summit. Yes, his talk will include the new Quack protocol.

Conclusion

This post is a short summary of the changes in v1.5.3. As usual, you can find the full release notes on GitHub.

Test-Driving the Lance Lakehouse Format in DuckDB

Publication date: 2026-05-21

Authors: LanceDB team, Guillermo Sanchez

TL;DR: Lance is an open lakehouse format with a design geared toward AI workloads. LanceDB and DuckLabs have partnered to bring you fast vector and hybrid search directly from DuckDB SQL, without leaving your analytical workflow. In this post, we explain what Lance is, how to use it in DuckDB, and, of course, show some benchmark results.

With the `lance` extension, DuckDB users can query Lance datasets with the same familiar SQL interface (via the CLI or SDKs), while adding capabilities for AI and retrieval workloads. This blog post highlights how Lance is a good option for workloads that need to support storage and querying of vectors, rich table operations, and AI-oriented access patterns, while also supporting scan-friendly analytical workloads at scale. And with DuckDB, it becomes trivial to query those kinds of datasets in SQL.

> In this blog, there will be some mentions of "retrieval workloads" and "AI data patterns" or "AI datasets". By "retrieval workloads," we mean queries that find rows by similarity or keyword relevance, such as vector search and full-text search, rather than by exact filters or aggregations. By "AI data patterns," we mean datasets that mix embeddings, images, or audio alongside scalar metadata.

What is Lance?

Lance is an open lakehouse format designed for modern ML and AI workloads. Unlike Parquet, Lance is a file format, a table format, and a lightweight catalog spec all at once. At the table format level, Lance supports versioning, schema evolution, indexes, and transactional updates through MVCC and ACID-style semantics. In practice, this means Lance is built for datasets that change over time and need more than read-only scans.

This matters because many AI datasets are no longer just rows of scalar values. They often contain embeddings, long-form text, images, audio, metadata for filtering, and indexes used for retrieval. A format that works well for these workloads needs to do more than store and scan columns efficiently. It also needs to support search, updates, and lifecycle operations without forcing users into managing multiple different systems.

The mental model is still familiar to users coming from Parquet: columnar data in an open format, queried with standard analytical tools. Lance's fragment-based layout stores data in small columnar chunks. This design enables efficient random access without trade-offs in scan performance or memory utilization, something that has historically been difficult to achieve in columnar formats, and that the Lance team examines in this 2025 paper on adaptive structural encodings.

On the data evolution side, adding columns or backfilling existing rows with new data only writes new files without touching existing ones. That makes schema changes lightweight in practice, which is useful for workflows where columns are added incrementally, such as appending derived features or embeddings to an existing dataset.

The Lance DuckDB Extension

The `lance` extension brings Lance into DuckDB as part of the SQL-based workflow. You can read Lance datasets directly, write to them via COPY, attach them as table namespaces, build indexes, and query them with regular DuckDB SQL. On top of that, the extension exposes Lance-native search functionality through SQL table functions.

This fits naturally with how DuckDB is already used: as a single, embedded SQL query engine that operates on many different data sources and file formats. With the Lance extension, DuckDB remains the familiar query engine, while Lance provides the storage, indexing, and search capabilities underneath, which is especially beneficial when your data is multimodal and includes embeddings.

Example Usage

Installing and using the extension is straightforward:

```
INSTALL lance;
LOAD lance;

SELECT *
FROM 'path/to/dataset.lance'
LIMIT 10;
```

DuckDB can also write Lance datasets directly:

```
COPY (
    SELECT *
    FROM (
        VALUES
            (1::BIGINT, 'duck', [0.9, 0.7, 0.1]::FLOAT[3]),
            (2::BIGINT, 'horse', [0.3, 0.1, 0.5]::FLOAT[3]),
            (3::BIGINT, 'dragon', [0.5, 0.2, 0.7]::FLOAT[3])
        ) AS t(id, animal, vec)
) TO 'path/to/out.lance' (FORMAT lance, mode 'overwrite');
```

Once the data is in Lance, DuckDB can query it with Lance-native search operators. For example, hybrid search combines vector similarity and keyword relevance in one SQL query:

```
SELECT id, text, _hybrid_score, _distance, _score
FROM lance_hybrid_search(
    'path/to/dataset.lance',
    'vec',
    [0.1, 0.2, 0.3, 0.4]::FLOAT[4],
    'text',
    'puppy',
    k = 10,
    prefilter = false,
    alpha = 0.5,
    oversample_factor = 4
)
ORDER BY _hybrid_score DESC;
```

The extension also exposes `lance_vector_search(...)` for vector similarity search and `lance_fts(...)` for full-text search, so users can choose the retrieval mode that fits their workload.

If you want table-style access instead of path-based access, you can attach a directory as a Lance namespace:

```
ATTACH 'path/to/dir' AS ns (TYPE lance);

SELECT count(*)
FROM ns.main.my_table;
```

Index creation also happens through SQL. For example, a vector index can be created directly on a Lance dataset:

```
CREATE INDEX vec_idx ON 'path/to/dataset.lance' (vec)
USING IVF_FLAT WITH (num_partitions = 1, metric_type = 'l2');
```

The extension surface goes well beyond read-only scans. In the current implementation, DuckDB can:

- Read Lance datasets with direct path scans
- Write and append Lance datasets with `COPY ... TO ... (FORMAT lance)`
- Run vector, full-text, and hybrid search with SQL functions
- Attach local directories or custom catalogs via REST namespaces
- Create, update, delete, merge, and alter tables in attached namespaces
- Create and manage vector, scalar, and full-text indexes
- Run maintenance operations such as compaction, cleanup, and index optimization

Note that this extension is not just a file reader, but it also gives DuckDB users a way to work with Lance as an operational table format from inside SQL.

Why Lance and DuckDB?

The combination of Lance and DuckDB is compelling for three reasons.

First, it gives users one SQL surface for analytics plus retrieval. The same DuckDB workflow can scan a dataset, filter it, join it with other tables, compute aggregates, and then run vector search or hybrid search over the result set. That is a good fit for AI applications where retrieval is only one step in a larger analytical pipeline.

Second, Lance is a table format for more than traditional analytics. Many AI pipelines need versioned datasets, updates, deletes, `MERGE`-style changes, index management, and schema evolution. The DuckDB extension exposes these capabilities through SQL, which means users do not need to leave the DuckDB environment just because their dataset is doing more than serving analytical reads.

Third, the workflow scales from local files to remote storage without changing the mental model. You can start with a local `lance` dataset, then move to object storage.

The extension also supports REST namespaces, so DuckDB can connect to a remote Lance catalog (including LanceDB Enterprise) and treat it like an attached database. That makes the local-to-remote storage progression feel incremental rather than disruptive.

To sum up, DuckDB remains the familiar SQL engine, while Lance adds storage and indexing features that are especially useful when the same dataset powers both analytics and retrieval.

Performance Experiment

LAION is an open dataset of image/caption pairs scraped from the public web, originally released to support research on models like CLIP, which learn a shared embedding space for images and text. The full release spans billions of pairs. For this experiment, we used the `lance-format/laion-1m` subset on Hugging Face Hub, which is easy to reproduce locally.

Each row carries a caption, a 768-dimensional CLIP image embedding, the raw image bytes, and scalar metadata like width, height, and NSFW flags. This mix of scalar, text, vector, and blob data in a single table makes it a useful workload for comparing formats, and it is structurally different from the wide-but-flat schemas like TPC-H or ClickBench that are traditionally used for analytical benchmarks.

The public Hugging Face export used by the benchmark currently materializes 69,632 rows locally, not the full million-row source dataset. The runner first downloads the public Parquet shards, then builds all local artifacts from that same baseline: an LZ4-compressed Parquet file, an indexed DuckDB database, and a Lance dataset. Generated files are reused across runs, so the initial download is the only networked step.

> The experiments were run on an Apple MacBook Pro with a 10-core M1 Max CPU and 32 GB of RAM, running DuckDB 1.5.2.

The benchmark was run using DuckDB as the query engine for the following three storage formats:

- **Parquet**: DuckDB scanning the LZ4-compressed Parquet baseline directly, with no auxiliary indexes.
- **DuckDB indexed**: the same baseline loaded into a DuckDB table, with DuckDB's `vss` (HNSW) and `fts` extensions layered on top, plus scalar indexes on filter columns. This is the typical "build it yourself in DuckDB" stack.
- **Lance native**: the same baseline written to a Lance dataset with a vector index, a full-text index, and native blob storage, queried through the DuckDB lance extension.

The workloads are aligned by task across the three paths, even though the exact SQL differs by storage/indexing backend:

- `fts`: find rows by keyword search over the caption text.
- `vector_exact`: run nearest-neighbor search over the CLIP embedding column without using an approximate vector index.
- `vector_indexed`: run nearest-neighbor search over the same embedding column using the available vector index.
- `hybrid`: combine text search and vector search into one retrieval query, returning the best-ranked matches from both signals.
- `blob_read`: fetch image bytes for selected rows, which exercises random access to large binary values rather than just scalar or vector columns.

Each workload is run five times by default, and the tables below report the average. The full scripts and SQL queries are in the `laion_1m` benchmark directory.

Cold Results

The table below runs each workload cold, in a fresh DuckDB process, so it captures process startup, file open, and first-query cost. It's closest to what a one-off script or a cron job would see.

Workload	Parquet	DuckDB indexed	Lance native
`fts`	12 ms	11 ms	21 ms
`vector_exact`	695 ms	61 ms	89 ms
`vector_indexed`	761 ms	104 ms	12 ms
`hybrid`	465 ms	80 ms	17 ms
`blob_read`	1559 ms	271 ms	278 ms

In the cold run, Lance stands out in the `vector_indexed` and `hybrid` workloads. DuckDB's own format does well in `vector_exact` and `fts`, while the `blob_read` workload is pretty much on par. Parquet is not well optimized for vector searches or blob reads, but does well on a simple text search powered by regex.

Warm Results

The warm results are from running all workloads in a single DuckDB session after a silent warmup pass, so caches, memory-mapped pages, and loaded indexes are already primed.

Workload	Parquet	DuckDB indexed	Lance native
`fts`	12 ms	10 ms	7 ms
`vector_exact`	703 ms	30 ms	50 ms
`vector_indexed`	755 ms	2 ms	5 ms
`hybrid`	471 ms	11 ms	8 ms
`blob_read`	1484 ms	266 ms	276 ms

When caches and indexes are already warm, both DuckDB and Lance are significantly faster than using Parquet on retrieval workloads.

Conclusion

Lance is a relatively new addition to the world of open lakehouse formats. It is designed for datasets that change over time, contain more than scalar values, and need to support both search and retrieval alongside traditional scan workloads. From DuckDB, the extension makes these capabilities available through SQL, while preserving the familiar embedded workflow. The benchmark results reflect, particularly in cold runs, how Lance is a good alternative to DuckDB's own format for vector and hybrid search.

> The Lance support in DuckDB was made possible through a collaboration between DuckLabs and LanceDB.

New DuckDB-Iceberg Features in v1.5.3

Publication date: 2026-05-29

Authors: Tom Ebergen, Thijs Bruineman

TL;DR: DuckDB-Iceberg now has a number of new features supporting Iceberg Tables and Iceberg REST Catalogs: `MERGE INTO`, `ALTER TABLE`, partition transforms, V3 support, and others!

Despite the work required to develop the features needed for DuckLake v1.0 and Quack, the DuckLabs team is still hard at work on the DuckDB-Iceberg extension. In this blog post, we will demonstrate some of the features that are available in DuckDB v1.5.3. Many of these features were earmarked for a future release in our last Iceberg-themed blog post "Writes in DuckDB-Iceberg" – you can think of this post as "Part 2" to that blog post.

Getting Started

To experiment with the new DuckDB-Iceberg features, you will need to connect to your favorite Iceberg REST Catalog. There are many ways to do so: please have a look at the Connecting to REST Catalogs page, which has instructions for catalogs such as Apache Polaris and Lakekeeper. If you would like to connect to Amazon S3 Tables, please consult the Connecting to S3 Tables page. In any case, your `ATTACH` command will look something like this:

```
ATTACH '<warehouse_name>' AS my_datalake (
    TYPE iceberg,
    <other options>
);
```

`MERGE INTO` Support

DuckDB's `MERGE INTO` statement is the recommended way to express upserts when the target table does not have a primary key – which is the case for all lakehouse formats. With v1.5.3, `MERGE INTO` is now fully supported against Iceberg tables. You can apply a change set to an Iceberg table in a single statement, deciding per row whether to insert, update or delete.

Let's take this table for example:

```
CREATE TABLE my_datalake.default.people (
    id INTEGER,
    name VARCHAR,
    salary FLOAT
);
INSERT INTO my_datalake.default.people
    VALUES (1, 'John', 92_000.0), (2, 'Anna', 100_000.0);
```

id int32	name varchar	salary float
1	John	92000.0
2	Anna	100000.0

Let's run an update on this table with two records, one increasing person 1's salary and another adding a new person with id 3.

```
MERGE INTO my_datalake.default.people AS target
    USING (
        FROM (VALUES
            (1, 'John', 105_000.0),
            (3, 'Sarah', 95_000.0)
        ) t(id, name, salary)
    ) AS upserts
    ON (upserts.id = target.id)
    WHEN MATCHED THEN UPDATE
    WHEN NOT MATCHED THEN INSERT;
```

When querying the result, we get the following:

```
SELECT *
FROM my_datalake.default.people
ORDER BY id;
```

id int32	name varchar	salary float
1	John	105000.0
2	Anna	100000.0
3	Sarah	95000.0

You can also combine matched and unmatched branches with `WHEN MATCHED THEN DELETE` to express a delete set in the same statement. As with `UPDATE` and `DELETE`, `MERGE INTO` uses merge-on-read semantics and writes positional deletes to the Iceberg table.

`ALTER TABLE` Support

In DuckDB v1.4's Iceberg extension, the lack of schema evolution of Iceberg tables was a documented limitation. In v1.5.3, the `ALTER TABLE` statement is now supported against Iceberg tables, covering the most common schema-evolution operations.

```
-- Create the table
CREATE TABLE my_datalake.default.simple_table AS
    FROM (VALUES
        (1, 'Andy'),
        (2, 'Bob'),
        (3, 'Claire'),
        (4, 'Mr. Duck')) t(col1, col2);

-- Rename the table
ALTER TABLE my_datalake.default.simple_table
    RENAME TO renamed_table;

-- Add a column
ALTER TABLE my_datalake.default.renamed_table
    ADD COLUMN col3 DOUBLE;

-- Rename a column
ALTER TABLE my_datalake.default.renamed_table
    RENAME COLUMN col2 TO name;

-- Drop a column
ALTER TABLE my_datalake.default.renamed_table
    DROP COLUMN col3;

-- Set the format-version
ALTER TABLE my_datalake.default.renamed_table
    SET ('format-version' = 3);
```

If we query the table after the schema changes, we get the following:

```
SELECT *
FROM my_datalake.default.renamed_table
ORDER BY col1;
```

col1	name
int32	varchar
1	Andy
2	Bob
3	Claire
4	Mr. Duck

In the background, each `ALTER TABLE` statement updates the `current-schema-id` of the Iceberg table. The changes are visible to other Iceberg-aware engines the next time they query the `LoadTableInformation` endpoint. Iceberg schema evolution is metadata-only, so no data files are rewritten.

`truncate` and `bucket` Support

The Iceberg specification defines several partition transforms that determine how data files are laid out on disk. In v1.5.3, DuckDB-Iceberg supports creating, inserting into, and updating tables that use the `bucket` and `truncate` partition transforms.

The `bucket(N, col)` transform hashes the column's value into `N` buckets, which is useful when you want stable partitioning on a high-cardinality column. `truncate(W, col)` groups rows by the first `W` characters (or by the column's value rounded down to a multiple of `W` for numeric columns), which is useful for prefix-based partitioning.

```
CREATE TABLE my_datalake.default.events (
    event_id BIGINT,
    user_id BIGINT,
    country VARCHAR,
    payload VARCHAR
)
PARTITIONED BY (bucket(16, user_id), truncate(2, country));

INSERT INTO my_datalake.default.events
    VALUES
        (1, 1001, 'United States', 'click'),
        (2, 1002, 'United Kingdom', 'view'),
        (3, 1003, 'Germany', 'click'),
        (4, 1004, 'Netherlands', 'view');
```

You can inspect the resulting data files to verify the partitioning:

```
SELECT file_path, record_count
FROM iceberg_metadata(my_datalake.default.events)
WHERE content = 'EXISTING';
```

Updates and deletes against bucket- and truncate-partitioned tables are also supported, using positional deletes under merge-on-read semantics.

Iceberg Schema Properties

Iceberg catalogs allow arbitrary key-value properties to be attached at the schema (namespace) level. These properties are typically used to record ownership, descriptions, default storage locations, or any other metadata that applies to every table in a schema.

- `iceberg_schema_properties`
- `set_iceberg_schema_properties`
- `remove_iceberg_schema_properties`

You can use them as follows:

```
-- to set schema properties
CALL set_iceberg_schema_properties(my_datalake.default, {
    'owner': 'analytics-team',
    'description': 'Default analytics schema'
});
-- to read schema properties
SELECT * FROM iceberg_schema_properties(my_datalake.default);
```

key varchar	value varchar
owner	analytics-team
description	Default analytics schema

```
-- to remove schema properties
CALL remove_iceberg_schema_properties(
    my_datalake.default,
    ['description']
);
```

Schema properties are written through the Iceberg REST Catalog, so any other Iceberg-aware engine attached to the same catalog will see the updates immediately. The returned value is the number of remaining schema properties.

V3 Support

The Iceberg v3 specification introduces several new features that DuckDB-Iceberg now supports for both reads and writes:

- `VARIANT` and `TIMESTAMP_NS` data types
- Schema-level default values for columns
- Binary deletion vectors
- Row lineage tracking

The biggest change in practice is binary deletion vectors. In v2 tables, DuckDB-Iceberg writes positional deletes as Parquet files; in v3 tables, the same information is encoded as a much more compact binary deletion vector (Puffin file). DuckDB picks the right format automatically based on the table's `format-version`.

You can create a v3 table by setting the `format-version` table property at creation time:

```
CREATE TABLE my_datalake.default.v3_table
WITH ('format-version' = 3) AS
    FROM (VALUES
        (1, {'kind': 'click', 'x': 10}::VARIANT, TIMESTAMP_NS '2026-05-20 12:00:00.123456789'),
        (2, {'kind': 'view'}::VARIANT, TIMESTAMP_NS '2026-05-20 12:00:00.987654321')
    ) t(id, payload, event_time);

-- Deletes against a v3 table are written as binary deletion vectors
DELETE FROM my_datalake.default.v3_table
WHERE id = 1;

SELECT * FROM my_datalake.default.v3_table;
```

id int32	payload variant	event_time timestamp_ns
2	{"kind": "view"}	2026-05-20 12:00:00.987654321

Looking at the metadata for the table confirms that the delete was written as a deletion vector rather than as a positional-delete Parquet file:

```
SELECT manifest_content, content, file_format
FROM iceberg_metadata(my_datalake.default.v3_table);
```

manifest_content varchar	content varchar	file_format varchar
DATA	EXISTING	parquet
DELETE	POSITION_DELETES	puffin

> The Geography type and Unknown type are not yet supported in DuckDB-Iceberg; we are planning to add those in DuckDB v2.0.0.

Conclusion and Future Work

With these features, DuckDB-Iceberg has closed many of the gaps called out in the previous blog post: partitioned writes, schema evolution, `MERGE INTO`, and many Iceberg v3 features are now available. There is still more to come, and as always, if you would like to see a specific feature prioritized, please reach out to us in the DuckDB-Iceberg GitHub repository or get in touch with our engineers.

Announcing DuckDB 1.4.5 LTS (Andium)

Publication date: 2026-06-17

Author: The DuckDB team

TL;DR: Today we are releasing DuckDB 1.4.5 LTS with bugfixes and performance improvements.

> We are simultaneously releasing two DuckDB versions: v1.4.5 LTS (Andium) and v1.5.4 (Variegata). **This blog post is about the LTS version, v1.4.5 (Andium).** Unless you are using specifically using the LTS version, you are likely looking for the latest stable version, v1.5.4 (Variegata).

In this blog post, we highlight a few important fixes in DuckDB v1.4.5, the fifth patch release in DuckDB's 1.4 LTS line. The release ships bugfixes, performance improvements and security patches. You can find the full release notes on GitHub.

To install the new version, please visit the installation page.

Fixes

Correctness

- #21482 – Correctly detect overflows when decoding integers from storage
- #21177 – Backport support for the `* NOT SIMILAR TO 'pattern'` from #19232

Crashes and Internal Errors

- #20804 – Backport various race condition fixes from #20803

Generic Bugfixes

- #20760 – Fix escape handling in `AddEscapes` function
- #21316 – Fix unbounded row group growth for indexed tables on repeated load and insert cycles
- #21577 – Fix for CSV reader buffer-boundary value read
- #20686 – Secret Manager: Ensure secrets are created exactly once
- #23197 – Backport out-of-bounds security fixes from #23100

Performance

- #21090 – AsOf Simple Joins improvements
- #21178 – Backport zstd improvements from #20943

Miscellaneous

- #23130 – Backport `GetLocalFileSystem` improvements from #21983
- #20727 – Use `FileSystem` APIs in `QueryProfiler::Write`

Conclusion

This post was a short summary of the changes in v1.4.5. As usual, you can find the full release notes on GitHub. We would like to thank our contributors for providing detailed issue reports and patches. Stay tuned for future DuckDB releasese, including v2.0.0 in the fall!

PS: Next week, we'll host DuckCon #7 in Amsterdam. Join us in-person or on the online stream.

Announcing DuckDB 1.5.4 (Variegata)

Publication date: 2026-06-17

Author: The DuckDB team

TL;DR: Today we are releasing DuckDB 1.5.4 (Variegata) with bugfixes and performance improvements.

> We are simultaneously releasing two DuckDB versions: v1.4.5 LTS (Andium) and v1.5.4 (Variegata). **This blog post is about the latest non-LTS version, v1.5.4 (Variegata).** If you are looking for the LTS version, check out the v1.4.5 (Andium) announcement.

In this blog post, we highlight a few important fixes in DuckDB v1.5.4, the fourth patch release in DuckDB's 1.5 (Variegata) line. The release ships bugfixes, performance improvements and security patches. You can find the full release notes on GitHub.

To install the new version, please visit the installation page.

Fixes

This version ships a number of performance improvements and bugfixes.

Correctness

- #23031 – Fix VARIANT cast reading wrong rows under a filter
- #23014 – MERGE INTO: only consider target table when binding `WHEN NOT MATCHED` and source table when binding `WHEN NOT MATCHED BY TARGET`
- #22825 – Fix case-insensitive column match in INSERT ... SELECT ON CONFLICT
- #22911 – Use non-deleted row count in `RowGroupReorderer`
- #23194 – Fix variant shredding analysis logic discrepancy with shredded writing
- #23234 – Fix problem with re-use of cached transform data for differently shredded files
- #22844 – Window Self-Join Limits: don't apply the self-join optimisation more than once

Crashes and Internal Errors

- #21854 – Fix double free and memory leak in Arrow GeoArrow CRS serialization
- #22836 – Fix progress bar output and crash when piping SQL
- #23174 – Fix crash when storage path is not set
- #23232 – Fix gzip compression write overflow
- #23156 – Avoid trying to bind an expression that doesn't exist in `UNPIVOT`
- #23189 – Guard against null row group reorder stats

Generic Bugfixes

- #22855 – Fix json_keys with wildcard paths
- #23144 – Fix json argument order affecting result
- #23116 – Reject NULL json key
- #23137 – Fix `ignore_errors` silently accepting invalid JSON
- #22882 – Fix geometry stats checkpointing when no changes are detected

- #22815 – Render MAP values as valid SQL in `Value::ToSQLString()`
- #23254 – Fix NULL propagation for date parts of infinite dates
- #23190 – Fix selection vector use in Arrow extension callbacks

Performance

- #23253 – Trim the system heap in the allocator flush path on jemalloc builds
- #23140 – Fix native geometry Parquet stats pruning and add `OPERATOR_ROW_GROUPS_SCANNED` to Parquet reader

Miscellaneous

- #23246 – Add explicit `-dark-mode` and `-light-mode` options to the CLI, and improve terminal background color detection
- #23100 – Add hardening to many DuckDB/Parquet decompression and deserializing paths
- #22690 – Add `vacuum_rebuild_indexes` as an (experimental) ATTACH option

Conclusion

This post was a short summary of the changes in v1.5.4. As usual, you can find the full release notes on GitHub. We would like to thank our contributors for providing detailed issue reports and patches. Stay tuned for future DuckDB releases, including v2.0.0 in the fall!

PS: Next week, we'll host DuckCon #7 in Amsterdam. Join us in-person or on the online stream.

Announcing DuckDB 1.5.5

Publication date: 2026-07-22

Author: The DuckDB team

TL;DR: Today we are releasing DuckDB 1.5.5 with bugfixes and performance improvements.

In this blog post, we highlight a few important fixes in DuckDB v1.5.5, the fifth patch release in DuckDB's 1.5 (Variegata) line. The release ships bugfixes, performance improvements and security patches. You can find the full release notes on GitHub.

To install the new version, please visit the installation page.

Fixes

Here are the most important fixes from the DuckDB v1.5.5 release, organized by category:

Correctness

- #23693 – Fix swapped min/max for multi-row-group 128-bit `DECIMAL` in `RETURN_STATS`

Crashes and Internal Errors

- #23517 – Fix min/max aggregate stats when row groups are filtered, which caused a crash
- #23351 – Fix deadlock in `TemporaryMemoryManager`
- #23566 – Fix C API scalar bind subquery crash
- #23757 – Fix segfault in external hash aggregate when radix bits grow after going external
- #23861 – Fix concurrent `ALTER` and `INSERT` crash
- #23593 – Fixed unsafe iteration when parent is `NULL` in string cast
- #23341 – Add additional guards to `DICT_FSST` to prevent exception during compression with small block sizes

Generic Bugfixes

- #23458 – Fix false RLE corruption error
- #23534 – Fix Arrow type extension bugs
- #23507 – Fix `ALTER TABLE ADD COLUMN IF NOT EXISTS ... DEFAULT` regression
- #23714 – Fix `DROP COLUMN` corrupting per-column metadata block bookkeeping
- #23808 – Fix `ALTER` dependency preservation in DependencyManager
- #23790 – `TryLookupEntry` now uses default schema as fallback
- #23354 – Prevent `NULL MAP` keys in `MultiFileReader` due to missing default values
- #23803 – Fix malformed JSON when rendering via duckbox
- #23879 – Fix reset of `empty_range` in the `TIMESTAMP range()` table function
- #23479 – Fix eviction node memory leak when using external files

Performance

- #23483 – Enable `ALP` and `ALP_RD` for storage version v1.5.0 and up with smaller block sizes

Miscellaneous

- #21293 – Add support for `duckdb://` URI scheme in ADBC
- #23230 – Support the ADBC Statistics API
- #23316 – Add request body length to HTTP logs
- #23327 – Show transport errors in HTTP log
- #23752 – Include extension header in libduckdb archives

Conclusion

This post was a short summary of the changes in v1.5.5. As usual, you can find the full release notes on GitHub. We would like to thank our contributors for providing detailed issue reports and patches. Stay tuned for future DuckDB releases, including v2.0.0 in the fall!

Asynchronous I/O in DuckDB: Work, Thread, Work

Publication date: 2026-07-31

Author: Pedro Holanda

TL;DR: Starting with v2.0, scheduled for fall 2026, DuckDB will support asynchronous reads of Parquet and CSV files. This can significantly speed up queries when synchronous I/O does not saturate the available bandwidth, as is typical in EC2/S3 compute-storage setups.

It doesn't matter how fast query operators are in a database system if we can't pull in the data quickly. For most of DuckDB's history, however, this problem was largely avoided by pruning data early. By pushing down filters and projections, we could ensure that we only read what we actually needed.

This worked particularly well because DuckDB primarily ran locally, with its main use case being as a quick-draw database engine for querying data directly from your machine's SSD. We could split the data into several partitions, such as row groups for Parquet files or fixed-size buffers for CSV files, and load them with low latency and high bandwidth. As a result, the main bottlenecks were elsewhere: subqueries, joins, aggregations, and so on. The actual data access path received less attention because synchronous access was perfectly suitable for this use case.

As usual, things changed. We realized that DuckDB's architecture was a great fit for querying remotely stored large-scale datasets, such as data lakes (e.g., DuckLake). Since May this year, we can even run DuckDB as a server using the Quack protocol. The original expectation of data files sitting on a local SSD therefore no longer always holds.

The practical implication of these changes is that many current DuckDB setups need to transfer files from remote storage to the machine that will actually process them. For data lakes, for example, a typical setup is to store the data in blob storage, such as S3, and process it on an EC2 machine in the same region. In this setup, latency and bandwidth play a much more significant role. If we cannot issue enough concurrent requests to use the available network bandwidth, performance can suffer drastically, with threads spending a large amount of their time waiting for remote reads instead of processing data.

As an example, let's consider a simple query over a remote Parquet file. For simplicity, let's assume we only have a single thread executing.

```
FROM read_parquet('s3://bucket/file.parquet');
```

A Parquet scan is partitioned into row-group-based jobs, with each job containing one or more fetch tasks that issue byte-range requests. With synchronous I/O, the worker thread will be blocked, waiting for the data to arrive at the machine before performing actual work, such as decoding, aggregating, and so on. You can see a visual depiction in the figure below, where the thread is blocked from doing any work while it waits for the read to finish.

{: .lightmode-img }

Synchronous read{: .caption }

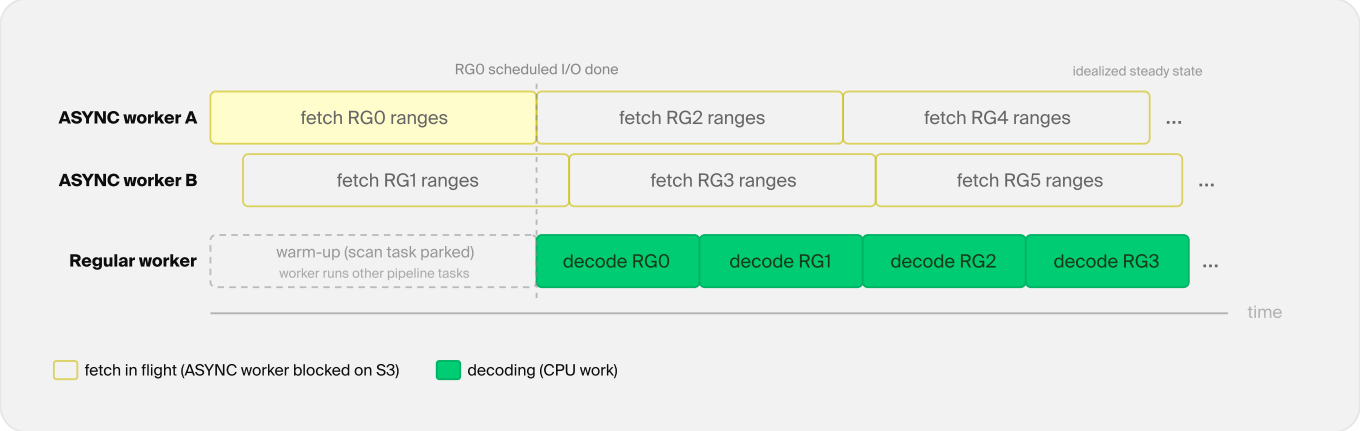
To address this, we have been implementing asynchronous I/O pipelines in DuckDB. They are currently implemented for Parquet and for uncompressed, seekable UTF-8 CSV files, with support for other formats, such as DuckDB's native format and JSON, still to come. In the

remainder of this blog post, we will give a simple explanation of how asynchronous I/O is implemented in DuckDB and provide benchmarks for both Parquet and CSV files.

> If you would like to try asynchronous I/O now, you can do so by using DuckDB's v2.0.0-dev preview builds. Asynchronous I/O will be used by default from the next major DuckDB version, v2.0, released in the fall.

Asynchronous I/O

The conceptual idea of asynchronous I/O is rather simple: we should be able to start an I/O operation without blocking the worker thread that requested it. Applied to our Parquet example, the same picture would look like the following:



{: .lightmode-img }

Asynchronous read{: .caption }

In this example, we have two ASYNC threads and one regular worker thread. The ASYNC threads keep *fetch tasks* in flight while the worker thread decodes data. During the initial warm-up, the scan task parks, leaving the worker thread free to run other pipeline tasks. Once the first job is ready, fetching and decoding can overlap.

In DuckDB, we implemented something similar. We have two separate thread pools:

- REGULAR – This pool contains our worker threads (by default: one for each available CPU thread). These are the ones that do real work, like decoding, joins, and aggregations. They prioritize regular work but can also perform I/O tasks when idle.
- ASYNC – A pool of threads intended for asynchronous tasks, primarily blocking I/O.

The main reason we have these two different pools is that, for remote I/O, these threads can spend almost all their time blocked, waiting for an HTTP response, for example, and hence have very little CPU utilization. Because of that, we have many more ASYNC workers than system threads, with the default setting being `4 * system threads` and the total being capped at 256.

It's of utmost importance to keep as many of our ASYNC threads busy as possible. To ensure that, we implement a read-ahead strategy instead of issuing reads on demand. This means scheduling fetch tasks ahead of what our regular worker threads currently need.

One thing we need to be attentive to is that read-ahead buys throughput by holding memory. If decoding is slow and the network is fast, prefetched data can accumulate and lead to out-of-memory issues. To mitigate this, we also implemented asynchronous memory governance. Both read-ahead and memory governance will be explained in more detail in the following sections.

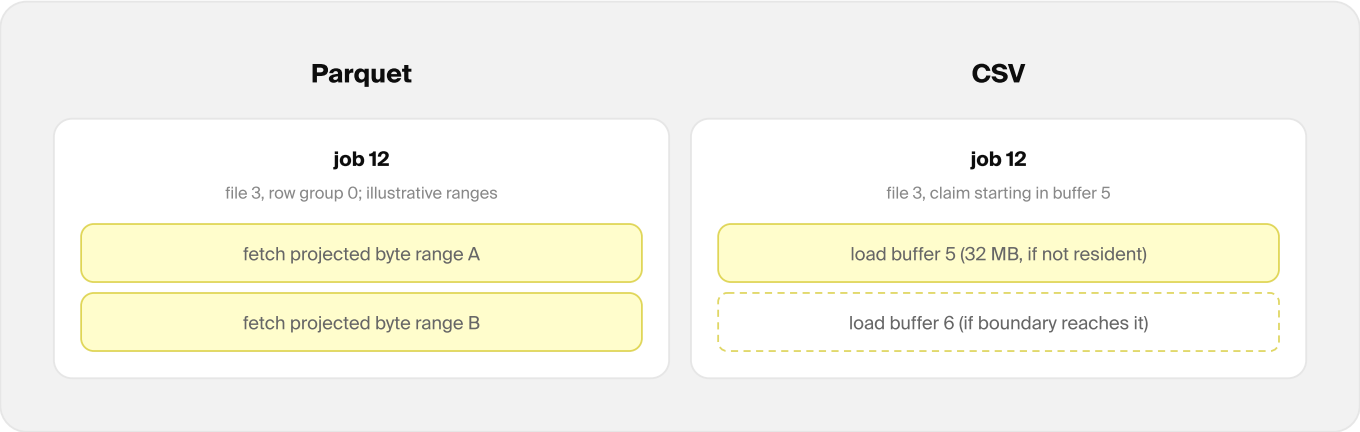
Read-Ahead Queue

The idea of read-ahead is also straightforward. Instead of starting a read at the exact moment a regular worker needs the data, we schedule fetch tasks for work that lies further ahead. While a regular worker decodes the current job, the ASYNC threads are already pulling in data for the next jobs. The goal is to keep enough fetch tasks in flight to hide the latency of remote storage.

The jobs are units of work that can be scheduled and processed independently, and they can be different depending on the underlying file format. For a Parquet file, a job is one row group of one file. For a CSV file, a job is a scan boundary that generally covers a fixed byte range within the file.

A Parquet job might be broken down into multiple fetch tasks depending on the query projections, filter pushdowns, physical column locations, and which nearby byte ranges can be combined. The two fetch tasks in the figure below are illustrative, as their exact grouping and sizes depend on the file and query.

For CSV files, we don't have the same granularity of information as we do for Parquet files. A job's fetch tasks load its starting buffer if it is not already in memory and, when the scan boundary reaches the end of that buffer, the following buffer as well (e.g., to handle lines that are split across two buffers).



{: .lightmode-img }

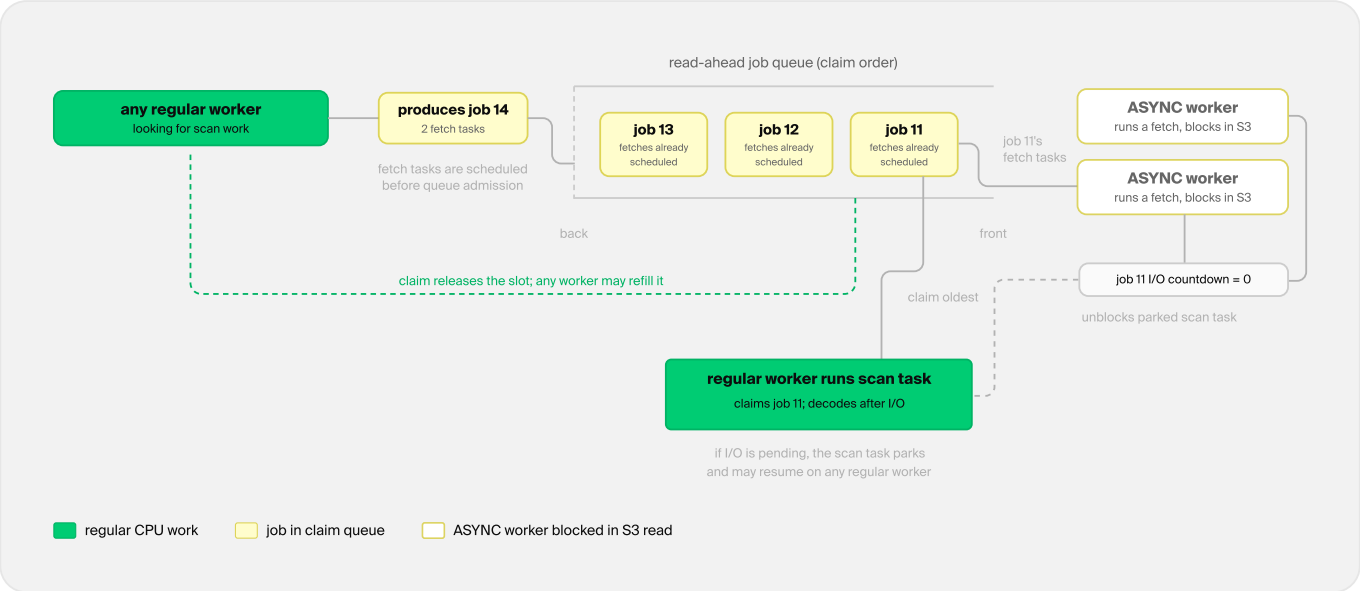
Jobs{: .caption }

Filling the queue requires no dedicated producer thread. Any regular worker that comes looking for scan work first tops up the queue as far as it is allowed to. The limit is either given by a user-specified number of slots or by a memory budget. If there is space, a job and its fetch tasks are created. The fetch tasks are scheduled immediately on the ASYNC pool, while the job is admitted to the read-ahead queue in batch order.

ASYNC threads execute individual fetch tasks independently of the job queue's claim order. Fetch tasks from the same job can run concurrently, although no particular assignment to ASYNC threads is guaranteed. All fetch tasks of a job share a countdown, and the fetch task that brings it to zero completes the job's I/O.

A worker thread claims the oldest job in the queue and checks that countdown. If I/O is done, the worker starts decoding the job. If not, it parks the scan task and is free to run other pipeline tasks. The last fetch task then unblocks the scan task, which may resume on any regular worker.

Claiming the job also immediately frees a queue slot, allowing any regular worker looking for scan work to produce a replacement job at the back of the queue. The figure below depicts this cycle:



{:

.lightmode-img }

Read-ahead cycle{: .caption }

Memory Management

Keeping more fetch tasks in flight consumes more memory. To determine a budget and avoid out-of-memory issues, we introduced the `read_ahead_depth` configuration option. It can have three types of values:

- `-1` (default): unlimited depth, bounded by memory.
- `N > 0`: at most N jobs ahead, with no memory budget.
- `0`: read-ahead is off, each scan task schedules I/O only for its own job.

To configure it, use the SET clause, e.g.:

```
SET read_ahead_depth = 5;
```

In the default mode, the budget is negotiated with the temporary memory manager, which is the same manager that splits memory between concurrent joins, sorts, and window operators. When there is a lot of memory pressure, for example, because an operator is using a large amount of memory, queue reservations might instantly be over budget. In practice, this means that the queue will only allow one job at a time, and the scan will behave close to a synchronous scan.

When the memory-heavy operator finishes, the memory manager has more budget to give, and the queue fills back up.

Benchmarks

Asynchronous I/O should have the largest effect when the latency of synchronous requests prevents us from using the available remote bandwidth. To measure this effect, we ran TPC-H Query 6 at SF100, with the data sitting on S3, and compared the results against DuckDB v1.5.5, our latest stable release. The SF100 dataset was written as a single file per table for both the Parquet and CSV benchmarks, with the `lineitem` table containing 600,037,902 rows.

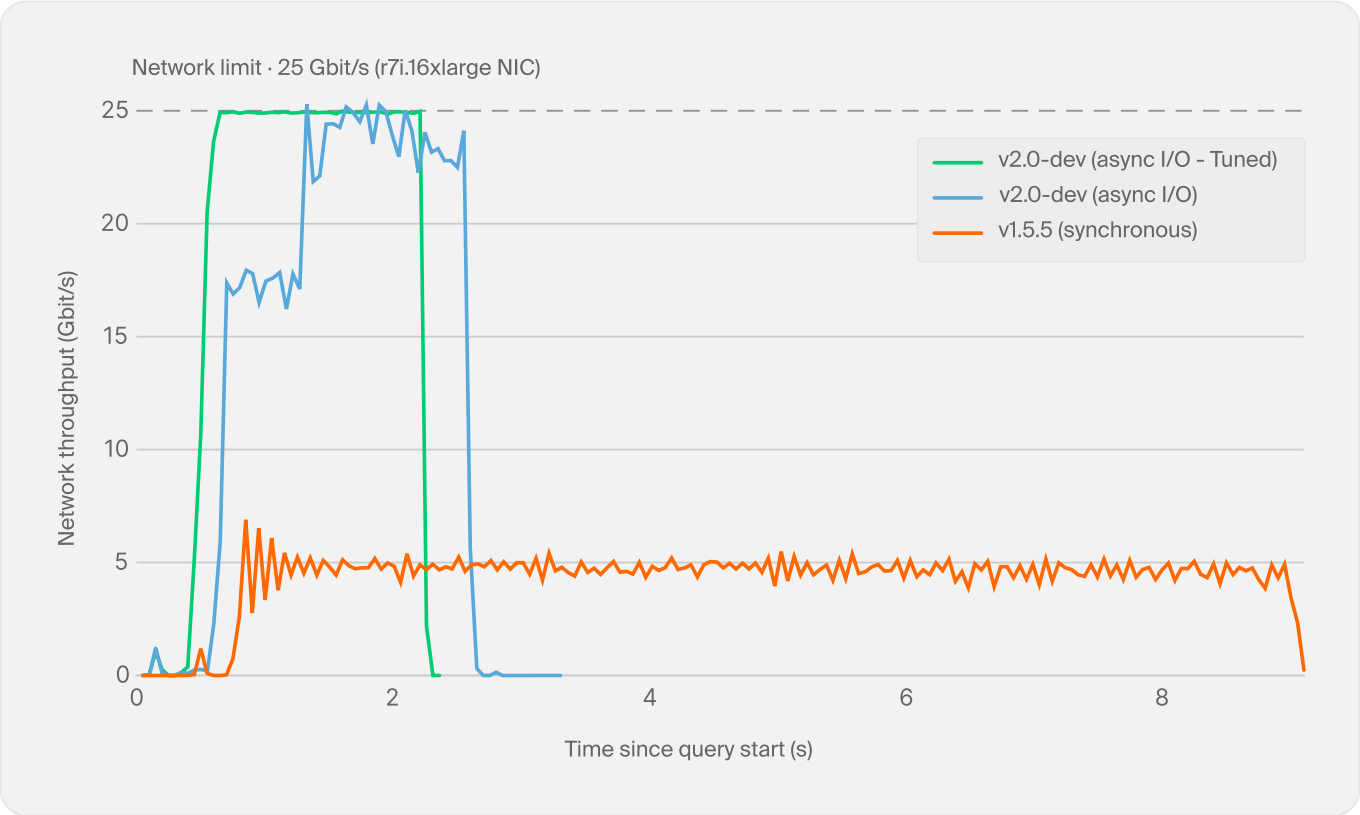
For compute, we used an EC2 r7i.16xlarge machine (64 vCPUs and 512 GB of RAM), with both the machine and the S3 bucket with the data located in the same region. We executed the query five times and report the mean execution time. The files were never cached (i.e., `SET enable_external_file_cache = false;`{:.language-sql .highlight}), meaning that every execution read the data straight from S3.

Parquet

The Parquet file is approximately 22 GB and has around 4,880 row groups, with each row group containing approximately 122,880 rows. With asynchronous I/O, the mean runtime drops from 8.230 seconds to 2.844 seconds, making the query almost 3× faster.

Version	Q6 runtime
v1.5.5 (synchronous)	8.230 s
v2.0.0-dev (asynchronous I/O)	2.844 s

Below we also show the network throughput over the course of the query:



{: .lightmode-img }

Network throughput{: .caption }

In it, we run DuckDB v1.5.5 and two variations of DuckDB v2.0.0-dev. One with the read-ahead depth determined by the memory governor, and one tuned for this machine, where we cap the read-ahead at 64 in-flight jobs and adjust the I/O settings (`SET async_threads = 48; SET http_retries = 8; SET http_retry_wait_ms = 50; SET http_retry_backoff = 2`{:.language-sql .highlight}). We can see that v2.0.0-dev uses the available bandwidth much more effectively, approaching the network limit and reaching it at several points. The tuned version goes further. With fewer, hotter connections and cheap retries, the throughput variance drops to a minimum and the 25 Gbit/s network stays almost fully saturated. Its query time was 2.227 seconds, reducing the runtime of the untuned v2.0.0-dev run by 21.7% and making it about 3.7× faster than DuckDB v1.5.5. In comparison, v1.5.5 stays around 5 Gbit/s because its synchronous reads do not keep enough requests in flight to saturate the network.

One other detail worth noting is that, in all experiments, a few hundred milliseconds pass before the first bump in network traffic, followed by another few hundred milliseconds before the main data transfer begins. The first gap is the time needed to open a DuckDB connection, perform the first TLS handshake, and open the file. The bump corresponds to downloading the file footer, while the second gap comes from processing the information in the footer before executing the query. We believe this is an area we can investigate and optimize further before the v2.0 release.

> We sampled the NIC's received-byte counter every 50 ms and calculated the throughput from the change in bytes between samples. We independently confirmed that the machine can access the network at 25 Gbit/s with both a DuckDB full-file read and the `s5cmd` tool.

Local Disk Remote storage is the main target for asynchronous I/O, but cold local reads give us a useful contrast. To measure them, we ran TPC-H Query 6 over the SF100 Parquet file, this time with the file sitting on the local disk of a MacBook Pro (Apple M4 Max, 14 cores, and 36 GB of RAM). Since the benefit of asynchronous I/O on local disks comes from cold reads, we cleared the OS caches (with the macOS `purge` command) between runs, making sure every execution actually read the file from disk.

Version	Q6 runtime
v1.5.5 (synchronous)	1.321 s
v2.0.0-dev (asynchronous I/O)	0.883 s

We can see that, for cold runs, asynchronous I/O is approximately 1.5× faster, reducing runtime by about 33%. The performance difference is much smaller than in the cases presented above, due to the SSD having much lower latency and much higher bandwidth than the EC2/S3 network. For hot runs, the difference is negligible, as there is no disk access happening if data is properly cached.

Small Files Partitioned datasets are a particularly relevant use case here, as partitioning can easily spread the data across many small files. To see how asynchronous I/O behaves in this setup, we also performed a Parquet run using the same TPC-H SF100 dataset. Instead of using one file, we generated 976 files with five row groups each. Each file contains approximately 615,000 rows and is around 22 MB in size.

Version	Q6 runtime
v1.5.5 (synchronous)	9.344 s
v2.0.0-dev (asynchronous I/O)	2.945 s

We can see that v2.0.0-dev delivers a similar performance improvement here as it does for the single-file benchmark, running around 3× faster. This shows that read-ahead can also parallelize across multiple files without becoming bottlenecked by opening files or fetching their footers.

Large Row Groups We also wanted to see what happens at the other extreme, when a Parquet file has only a few very large row groups. For this experiment, we generated six versions of the same TPC-H SF100 `lineitem` table as a single file, changing only the requested row group (RG) size, and ran Q6 using DuckDB v2.0.0-dev. The table below reports the runtime for each version.

Rows / RG	RGs	Approx. RG size (MB)	Total file size (MB)	Time
122,880	4,886	~4 MB	~21,600 MB	2.74 s
1,966,080	306	~70 MB	~21,400 MB	2.11 s
9,375,593	64	~320 MB	~20,500 MB	2.27 s
62,914,560	10	~1,500 MB	~14,700 MB	3.69 s
150,009,476	4	~3,200 MB	~12,800 MB	8.01 s
600,037,902	1	~12,300 MB	~12,300 MB	25.26 s

At first, larger row groups decrease query times. As we increase the row-group size, request latency is amortized over much larger transfers. However, beyond a certain point, the available parallelism starts to fall. A row group is DuckDB's unit of Parquet scan parallelism, so ideally a scan should expose at least one row group per system thread. On this 64-vCPU machine, the version with 64 row groups provides exactly that and finishes in 2.27 seconds, while the fastest run comes from the version with 306 row groups, at 2.11 seconds.

However, with fewer row groups than threads, we lose parallelism and can no longer saturate the network. For Q6, the projections and the physical location of the columns result in two fetch requests per row group. Four row groups therefore expose only about eight concurrent S3 streams, raising the runtime to 8.01 seconds. For the largest configuration, the file contains a single row group, and its I/O is effectively reduced to two giant streams, pushing the runtime to 25.26 seconds. This happens even though better compression makes the file a little over half the size of the version with 4,886 row groups. In this case, the extra bandwidth required by smaller row groups is cheaper than the parallelism lost with extremely large ones.

Concurrent Queries The effect becomes even clearer when several queries run at the same time. For this experiment, we ran TPC-H queries 1, 6, 9, and 18 concurrently against the same SF100 Parquet dataset on S3, using a single DuckDB instance. We picked these queries because they cover a mix of scans, aggregations, and joins, with different CPU and memory requirements. We repeated the experiment with the default memory configuration and with memory limits of 16 GB and 8 GB. The total runtime is the wall-clock time until all four queries finish. We report the average and peak CPU utilization (number of cores utilized), the peak bandwidth (bw.) and the peak resident set size (RSS).

Version	Mem. limit	Runtime	Avg. CPU	Peak CPU	Peak bw.	Peak RSS
v1.5.5	default	35.8 s	5.9	35.7	10.7 Gbit/s	14.5 GB
v2.0.0-dev	default	15.6 s	48.1	64.0	24.9 Gbit/s	20.1 GB
v1.5.5	16 GB	35.6 s	6.1	25.7	17.4 Gbit/s	14.1 GB
v2.0.0-dev	16 GB	22.7 s	35.2	63.4	24.8 Gbit/s	15.7 GB
v1.5.5	8 GB	35.9 s	6.9	38.8	16.8 Gbit/s	10.4 GB
v2.0.0-dev	8 GB	24.2 s	30.3	63.7	25.0 Gbit/s	11.5 GB

With the default memory configuration, DuckDB v1.5.5 keeps an average of only about 6 of the 64 cores busy. In other words, around 90% of the machine sits idle waiting for synchronous S3 reads. DuckDB v2.0.0-dev, on the other hand, averages 48 busy cores, reaches all 64 at its peak, and saturates the 25 Gbit/s network. As a result, all four queries finish in less than half the time.

The memory results are also interesting. As we lower the limit, the memory governor reduces the read-ahead backlog, while memory-heavy operators such as those in Q18 can spill to disk. This lowers the peak physical memory used by the DuckDB process (i.e., RSS) of DuckDB v2.0.0-dev from 20.1 GB with the default configuration to 15.7 GB with a 16 GB limit and 11.5 GB with an 8 GB limit. The additional spilling and reduced read-ahead also lower average CPU utilization and increase the runtime, but v2.0.0-dev continues to saturate the network and remains substantially faster than v1.5.5 in both cases.

> One might notice that the 8 GB result still peaks at 11.5 GB of RSS. This is because jemalloc keeps recently freed pages resident for about one second so they can be reused. This memory is no longer counted by DuckDB's memory manager, and v1.5.5 shows the same allocator behavior.

CSV

The effect is larger on CSV files. The CSV file is 80.89 GB, and asynchronous I/O reduces the mean runtime from 878 seconds to just 45 seconds, making the query almost 20× faster. CSV is row-oriented, so the scan transfers substantially more data and performs fixed-size buffer reads, making concurrent remote reads especially valuable.

Version	Q6 runtime
v1.5.5 (synchronous)	877.563 s
v2.0.0-dev (asynchronous I/O)	45.264 s

As in the other experiments, we used the default memory-governed read-ahead depth, so this run was not tuned to keep the 25 Gbit/s network saturated on average.

Conclusion

In this blog post, we presented the recent work on asynchronous I/O for Parquet and CSV files. Most of its benefit comes from accessing remote data, but cold local reads can also benefit, albeit less. Next, we plan to add async reads for JSON and DuckDB-native files, as these are the two other formats most relevant to DuckDB core. Formats that live in out-of-tree extensions are not on the roadmap yet. We will also

investigate `io_uring`, Linux's asynchronous I/O interface, which could reduce system-call overhead and the number of threads blocked on I/O. If it proves beneficial in practice, we will integrate it into DuckDB. One important thing to notice is that any of the data lake solutions supported in DuckDB can already benefit from asynchronous I/O automatically as long as the underlying data format is Parquet (or CSV, if you are brave enough).

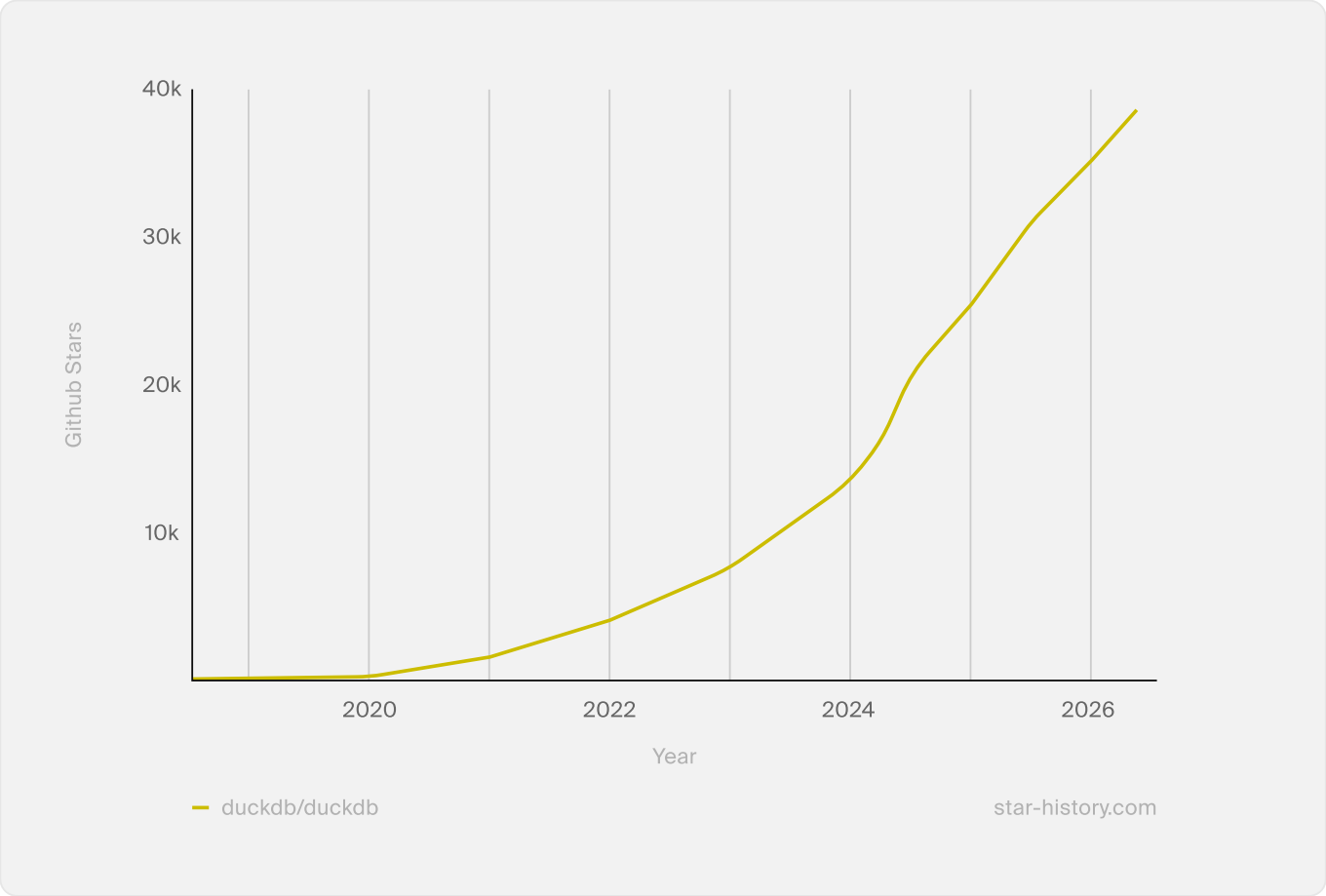
Thank You for 40 000 Stars on GitHub

Publication date: 2026-08-05

Author: The DuckDB team

TL;DR: DuckDB just reached 40 000 stars on GitHub! Here's what happened since the last 10 000-star milestone.

The duckdb/duckdb GitHub repository has just passed 40 000 stars!



{: .lightmode-img }

We'd like to use this milestone to stop for a moment and revisit recent developments in the "Duck Stack" ecosystem since last summer (2025), when we surpassed 30 000 stars.

Releases

- Since the last milestone, the DuckLabs team has released DuckDB 1.4.0, our first long-term support release, and DuckDB 1.5.0. Both releases are packed with many new features, performance optimizations, and bugfixes.
- The DuckLabs team also published the DuckLake 1.0 standard, our production-ready format for SQL-as-a-lakehouse.
- Finally, we released the Quack remote protocol, which lets you run DuckDB in a client-server setup with multiple concurrent writers. That's right: DuckDB instances can now talk to each other!

Community Metrics

Besides the GitHub stars, the DuckDB community has also experienced a lot of growth in other metrics:

- Each month, `duckdb.org` receives traffic from over 8 million unique visitors – more than double last summer's numbers. We also see over 2 PB (!) in traffic from millions of extension downloads. Thanks again to Cloudflare for sponsoring the project.
- DuckDB now has 50M+ monthly downloads in PyPI, more than double the 20M we reported last time.

(As usual, we'd like to emphasize that while we're happy to see these metrics grow, we are not glorifying them and they are not a target per se in accordance with Goodhart's law.)

Events

Since 30 000 stars, we ran DuckDB meetups in Amsterdam, Berlin, and London. We also hosted a Developer Meeting focused on DuckDB internals and extension development.

Finally, we hosted DuckCon #7, our largest DuckCon so far, which we also streamed online. These events allowed us to connect with our community of engineers, builders, and scientists, and learn about the many ways they are using DuckDB.

Next up, we'll host the DuckDB Paris meetup with Altertable and we are setting up a meetup in Boston. Stay tuned!

DuckDB in the Wild

Here are just a few of the projects, extensions, and announcements that happened in the DuckDB community since the last 10 000-star milestone:

- **Blog:** Petrica Leuca shows you how to use DuckDB for keyword, full-text, and semantic similarity search with embeddings for lightweight text analytics.
- **Video:** GizmoEdge: A distributed DuckDB engine for IoT (from the first DuckDB Developer Meeting) by Philip Moore.
- **Blog:** DuckDB for streaming analytics? Yes, says Guillermo Sanchez, you can use DuckDB to refresh your data at near real-time speed.
- **Use Case Walkthrough:** Uncovering Financial Crime with DuckDB and SQL/PGQ graph syntax that's part of SQL:2023 by Daniël ten Wolde.
- **For Fun:** DuckDB co-creator Hannes Mühleisen shows you how to store and even process video in DuckDB.
- **Blog:** Iceberg in the Browser: learn how to read and write tables in Iceberg catalogs without needing to manage any infrastructure – directly from your browser!
- **Presentation:** Denis Hirn from University of Tübingen introduces you to DuckPL: A procedural language in DuckDB.
- **Hardware benchmarks:** Hannes Mühleisen tests DuckDB on a Loongson CPU and Gábor Szárnyas put DuckDB to work on a MacBook Neo.
- **Video:** Barry Smart walks you through 20 years of wind data while auditing UK energy policy without a cluster (DuckCon).
- **Blog:** Interested in building DuckDB extensions in C#? Giorgi Dalakishvili shows you how to build DuckDB extensions in C# with DuckDB.ExtensionKit.
- **Talk:** Kian Mehrabani takes you on a tour of how Spotify is using DuckDB as a SQL layer over user listening history in this video from DuckCon.
- **Preview:** In DuckDB v2.0, scheduled for fall 2026, DuckDB will support async reads of Parquet and CSV files. Pedro Holanda walks you through the details of asynchronous I/O.

Closing Thoughts

We would like to extend a heartfelt **thank you** to the DuckDB community of users and contributors. We look forward to growing the community around the "DuckStack" and hope the GitHub stars follow. We'll report back when we reach 50 000 stars.

A Preview of DuckDB v2.0

Publication date: 2026-08-17

Authors: Mark Raasveldt, Hannes Mühleisen

TL;DR: DuckDB v2.0 is coming this fall. In this post, we preview its headline features: DuckDB as a server, triggers, the VARIANT type, asynchronous I/O, a new SQL parser, a new storage format, and much more.

> **Update.** DuckDB v2.0.0 was released on October 21, 2026. Some details changed since this preview; see the release announcement and the migration guide.

DuckDB v2.0 will be named "Cinnamon Teal" after the cinnamon teal *(Spatula cyanoptera),* a strikingly reddish-brown duck found in the western Americas.

A major version bump is not something we do lightly, and it is not just ceremony: v2.0 ships a new SQL parser, a new default storage format, a reworked C API, and a small number of carefully chosen breaking changes. But above all, it is a feature release, built from over 10,000 commits since we released v1.5 in March. Where last year was the year of the lakehouse, this release kicks off the year of DuckDB as a server. We previewed many of these features in the "State of the Duck" talk at DuckCon #7, if you prefer to watch instead of read.

DuckDB is moving rather quickly, and we can only cover a small fraction of the changes here. Condensing all new features down to a shortlist is always a fight over what gets in, and yes, we know that what follows is technically a listicle *(Ten Things Coming to DuckDB v2.0, Number Eight Will Shock You).* We are not proud of the format, but it works, so here it is, starting with the SQL-level features and working down into the engine.

1. DuckDB as a Server: Quack and CONNECT

DuckDB has been an in-process database since day one. But people have asked us – very persistently – for a client/server mode, and we have finally caved. The `quack` extension implements DuckDB's native protocol for talking to other DuckDBs. It was released as a preview shortly before DuckCon #7, and it is a big part of where DuckDB is headed: any DuckDB process can serve its databases over the network, and any other DuckDB can attach to it and route queries there using the new CONNECT statement. For example:

CONNECT is the successor to the `remote.query($$...$$)` workaround we showed when Quack was first revealed – we looked at that syntax and said: no, this cannot be it. And CONNECT is not limited to Quack: it points your session at any remote database that supports it, and the new remote pushdown optimizer (#22914) ships SQL directly to PostgreSQL and MySQL instead of pulling tables over the wire:

```
CONNECT 'postgres://localhost/mydb';
SELECT count(*) FROM orders; -- runs on the PostgreSQL server
DISCONNECT;
```

If you have worked with analytical systems in the past, you may assume that DuckDB cannot handle transactional workloads. But DuckDB has been built as a transactional, multi-connection database with full MVCC and transaction isolation since day one. Most users just never needed that in a single-user scenario. It turns out DuckDB handles transactions well: it's fast enough to compete with general-purpose databases like PostgreSQL on quite a few workloads, and the client/server pattern finally lets that machinery shine in multi-tenant, long-running deployments.

Running DuckDB long-term also comes with new challenges, which is why v2.0 pushes on better metrics, logs, and observability (see, e.g., the metrics layer rework in #22799) that let you look at a DuckDB instance and see what it is actually doing. People even built standalone clients for the Quack protocol within weeks of the preview. We thought we were extending DuckDB to talk to other DuckDBs; the world said no, no, no, and built their own clients. Who would have thought.

2. VARIANT Becomes a First-Class Citizen

The VARIANT type shipped in DuckDB v1.5, and the way to think about it is JSON on steroids. Basically, imagine if JSON were fast. Like JSON, a VARIANT column can store differently-shaped data in every row. Unlike JSON, it is not a text format: DuckDB automatically detects the common structure hidden in your semi-structured data and "shreds" it, so it compresses well in storage and executes fast in queries, all without you ever declaring a schema. This makes VARIANT a natural fit for real-time log ingestion, where streams of JSON-ish records share structure but evolve over time.

In v2.0, this pipeline works end to end: shredded execution straight from storage (#20912), extraction pushdown into scans (#22478), shredded VARIANT reading *and* writing for Parquet, and a family of `variant_*` functions:

```
CREATE TABLE events (payload VARIANT);
INSERT INTO events
VALUES ('{"user": {"id": 42, "tags": ["a", "b"]}}'::JSON::VARIANT);

SELECT variant_type(payload), variant_keys(payload)
FROM events;

SELECT *
FROM events
WHERE variant_contains(payload, {'user': {'id': 42}}::VARIANT);
```

Longer term, likely soon after v2.0 (but don't hold us to it), we plan to back the regular JSON type with VARIANT, so existing JSON workloads get all of these benefits without changing a single query.

3. Triggers

Triggers have been a long-standing feature request, and DuckDB v2.0 delivers them in full: BEFORE and AFTER triggers, FOR EACH ROW and FOR EACH STATEMENT, transition tables via REFERENCING OLD/NEW TABLE, multiple triggers per event, RETURNING on triggered tables, and DROP TRIGGER.

The classic use case is audit tables: something happens in the system, and a trigger records what changed. For example:

```
CREATE TABLE target (id INTEGER, val INTEGER);
CREATE TABLE audit (id INTEGER, old_val INTEGER, new_val INTEGER);

CREATE TRIGGER trg_audit AFTER UPDATE ON target
REFERENCING OLD TABLE AS o NEW TABLE AS n
FOR EACH STATEMENT
    INSERT INTO audit
    SELECT n.id, o.val, n.val
    FROM o
    JOIN n ON o.id = n.id;

INSERT INTO target VALUES (1, 10), (2, 20);
UPDATE target SET val = val * 10 WHERE id <= 2;
SELECT * FROM audit;
```

id	old_val	new_val
1	10	100
2	20	200

Triggers fit naturally with long-running DuckDB services, and we are also planning to use them internally to build several upcoming features. They are fully exposed at the SQL level too, so you can build your own cool stuff with them.

4. SQL Dialect Additions

As always, DuckDB's SQL dialect keeps growing. A few favorites from this release cycle:

With **NEAREST joins** (#24137), top-k similarity search becomes a join clause, handy for vector and embedding workloads:

```
SELECT q.user_id, t.product_id
FROM users q
    INNER JOIN products t APPROX NEAREST 2
    BY SIMILARITY array_cosine_similarity(q.embedding, t.embedding);
```

DML inside CTEs (#21634, #21997, #24217) lets you use `INSERT`, `UPDATE`, `DELETE`, and `COPY` as pipeline steps:

```
WITH moved AS MATERIALIZED (
    DELETE FROM staging RETURNING *
)
INSERT INTO archive SELECT * FROM moved;
```

Nested schemas (#23492, #24222) allow schemas within schemas:

```
CREATE SCHEMA finance;
CREATE SCHEMA finance.reports;
CREATE TABLE finance.reports.q3 (revenue DECIMAL);
```

The new **variable syntax** (#21194) lets you write `$x` anywhere an expression is allowed, no more `getvariable(...)` verbiage:

```
SET VARIABLE threshold = 100;
SELECT * FROM orders WHERE amount > $threshold;
```

The **JSON mutation functions** `json_set`, `json_insert`, `json_replace`, and `json_remove` (#23786) finally let you modify JSON documents in place:

```
SELECT json_set('{"a":1}', '$.b', '2');
```

json_set('{"a":1}', '$.b', '2')
{"a":1,"b":2}

And **recursive CTEs with `USING KEY` aggregation** (#19481) enable iterative algorithms in pure SQL, backed by the rewritten recursive CTE engine described below:

```
WITH RECURSIVE tbl(a, b) USING KEY (a, avg(b)) AS (
    SELECT 1, 5
    UNION ALL
    SELECT a, b - 1 FROM tbl WHERE b > 0
)
TABLE tbl;
```

a	b
1	2.5

There is more: SQL-standard `FETCH FIRST 2 ROWS ONLY` (#23533), `OVERLAY()` (#22456), `UNNEST` in `GROUP BY` (#23644), and well-defined `MERGE` / `UPDATE ... FROM` semantics for multi-matched rows (#24058).

5. Asynchronous I/O

Interacting with object stores like S3 is central to the DuckDB experience: your data has to come from somewhere, and it often sits in object storage. DuckDB has long been able to read from object stores in parallel, but synchronous access placed a limit on how fast this could go. DuckDB v2.0 introduces asynchronous I/O throughout the engine. We described the design in detail in a dedicated blog post.

Thanks to asynchronous access, the I/O layer now scales independently from the query processing layer, which means far more parallelism for remote reads and dramatically faster queries on network storage. Parquet support came first (#23662), with CSV (#23961) and DuckDB's own file format (#24654) following, along with asynchronous Parquet writes (#23283) and new `MMAP` and `DIRECT_IO` modes (#22988). Local storage benefits a little too, but network storage is where you will see the big gains.

6. Faster Queries Across the Board

As with every release, a lot of work went into making your existing queries faster without you doing anything. To pick some highlights: partial aggregates are now pushed below joins (#22572) and redundant aggregations are reused (#24543), the recursive CTE engine has been rewritten (#22211), aggregations now spill to disk when they outgrow memory (#24499), and the Windows CLI got approximately 2.2× faster at multi-threaded result materialization (#24036).

How much faster can this get? Here is a microbenchmark you can run on a laptop: single-source reachability over a graph with one million edges, written as a plain recursive CTE.

```
CREATE TABLE edges AS
    SELECT (range % 100_000)::INTEGER AS src,
           ((range * 13 + 7) % 100_000)::INTEGER AS dst
    FROM range(1_000_000);

WITH RECURSIVE reachable(node) AS (
    SELECT 0
    UNION
    SELECT dst FROM edges, reachable WHERE src = node
)
SELECT count(*) FROM reachable;
```

Version	Run time
DuckDB v1.5.4	4.90 s
DuckDB v2.0 (preview)	0.12 s

As you can see, DuckDB v2.0 is about 40× faster (!) for the same recursive query.

> **Update.** We published a blog post on how we are speeding up recursive CTEs in DuckDB v2.0.

Row-group pruning has been massively expanded: min-max indexes (zone maps) and Parquet Bloom filters now skip data for structs, lists, decimals, UUIDs, `IN` filters, and even function predicates:

```
-- these now prune row groups instead of scanning them:
SELECT * FROM logs WHERE contains(message, 'ERROR');
SELECT * FROM t WHERE substr(code, 1, 3) = 'NL-';
SELECT * FROM 'data/*.parquet' WHERE id IN (1, 5, 9);
```

Query planning also becomes **partition-aware** (#22336). Lakehouse formats (DuckLake, Iceberg and plain Hive-partitioned Parquet on S3) are all partitioned, and exploiting that partitioning is often the difference between scanning a dataset and skipping most of it. In v2.0, the planner and optimizer take full advantage of existing partitioning, and partitioned writes have been reworked as well (#22225, #22620).

7. Storage Format v2.0

DuckDB v2.0 bumps the default storage format version to v2.0.0 (#22875).

Column metadata is now loaded lazily (#22333), so wide tables open faster too. The `DICT_FSST` string compression method is enabled by default (#23733), deletes are stored compactly (#24336), and the storage layer performs much stronger corruption validation on read. In short: databases with big indexes and wide tables open faster and use far less memory. The new storage format also allows checkpoint vacuuming to compact tables with ART indexes by incrementally remapping entries whose row IDs change, avoiding a full index rebuild (#23653).

Later this year, ART indexes will be buffer-managed (#21458, #23605). This will allow ART index buffers to be evicted under memory pressure, and remove the restriction on the ART index fitting completely within memory.

8. A Brand New SQL Parser

DuckDB has famously always used a parser derived from PostgreSQL's. We have decided that enough is enough: v2.0 ships our own modern, extensible PEG-based parser (#22194), an idea we first explored in our 2024 post on runtime-extensible parsers. This change ties into the extension ecosystem: extensions can now hook into the grammar itself, so expect extensions that expose entirely new SQL syntax. It also brings better error messages with precise source locations.

You should not actually notice anything from the parser swap as we designed it to be compatible with the old one. If you do notice, please file an issue.

> **Update.** We published a new blog post on the PEG parser.

9. Timezones, Calendars, and Collations Without ICU

Timezone-aware timestamps, calendars, and collations in DuckDB have always been powered by the ICU library. ICU is a fine library, but we only ever used a small slice of it, while still carrying it around in every DuckDB distribution. In v2.0, the ICU library is gone entirely: the `icu` extension now implements timezones, calendars, and collations itself (#24463, #24403), with the timezone data built directly from the IANA database and compressed down to around 45 kB. Everything keeps working exactly as before:

```
SELECT '2026-08-14 12:00:00'::TIMESTAMPTZ AT TIME ZONE 'Europe/Paris';
SELECT * FROM names ORDER BY name COLLATE de;
```

Besides being much smaller and easier to keep up to date, the new implementation is also simply faster. Here's a quick microbenchmark on a MacBook that converts 25 million timestamps to a timezone and filters 5 million strings with a German collation:

Query	v1.5.4 (ICU)	v2.0 (native)	Speedup
`ts AT TIME ZONE 'Europe/Paris'`, 25 M rows	0.24 s	0.11 s	2.2×
Filter with `COLLATE de`, 5 M rows	0.15 s	0.06 s	2.6×

10. Write Extensions Once, Host Them Yourself

Extensions are one of the best things about DuckDB, but today, most of them, including our own, build against the unstable C++ API. Extension authors are required to rebuild and publish their extensions for every DuckDB release, even if the extension itself does not change. This is often a non-trivial operation for the extension author and a non-trivial coordination effort for the DuckDB team. Not rebuilding an extension means it can't be installed on the latest version of DuckDB.

Developing Extensions

DuckDB v2.0 will ship with a revamped C API (see #24702 for part 1). The API will have a versioned specification expressed in YAML that uses an also versioned specification *schema*, and tooling for code generation (#24135). The schema lets us tag every symbol with its lifecycle and stability guarantees (#24435). A large part of the API will be marked stable and frozen, providing a stable ABI across DuckDB versions.

The API itself improves over the pre-2.0 C API in a number of ways. It provides coherent error handling and a unified set of conventions around naming and ownership. It covers a much larger feature surface, including scalar, aggregate, table, cast, and copy functions with named parameters and varargs, parsing and inspecting SQL statements, prepared statements, replacement scans, custom filesystems, direct access to vector buffers, values and types, and more. It supports streaming query result consumption. With this API we want to provide the primitives to access DuckDB's most powerful features.

You won't need to program against the C API directly, though. We also provide a C++ API on top of it. It is a thin layer that compiles into your extension and talks only to the stable C ABI, so the binary you ship stays independent of DuckDB versions. It gives you a convenient, typed, well-documented way to access DuckDB's API. We're also working on bindings that provide the same for extension writers using Rust.

So what will an extension using the C++ API look like? Here is a complete example, though the details may still change: a single file that registers a vectorized scalar function.

```
#include "duckdb_cpp.hpp"

using namespace duckdb_api;

DUCKDB_CPP_EXTENSION_ENTRYPOINT(extension) {
    // add_numbers(a BIGINT, b BIGINT DEFAULT 2): adds two BIGINTs, one vector at a time
    ScalarFunction function;
    function.SetName("add_numbers")
        .SetSignature(FunctionSignature::Create()
                        .AddParameter("a", LogicalType::BIGINT())
                        .AddParameterDefault("b", LogicalType::BIGINT(), Value::Bigint(2))
                        .SetReturnType(LogicalType::BIGINT()))
        .SetExecCallback([](ScalarFunction::ExecInput &input) {
            auto chunk = input.GetInputChunk();
            auto a = chunk.GetVector(0).GetView();
            auto b = chunk.GetVector(1).GetView();
            auto out = input.GetResultVector().GetDataMutable<int64_t>();
            for (idx_t i = 0; i < chunk.GetRowCount(); i++) {
                out[i] = a.Data<int64_t>()[a.SelAt(i)] + b.Data<int64_t>()[b.SelAt(i)];
            }
        })
        .Register(extension);
}
```

You can use it as follows:

```
LOAD add_numbers;
SELECT add_numbers(40, 2);           -- 42
SELECT add_numbers(40);              -- 42: b falls back to its default
SELECT add_numbers(b := 2, a := 40); -- 42: named arguments work too
SELECT add_numbers(40, NULL);        -- NULL, without the function doing anything
```

You do not need to re-target or rebuild it every time a new DuckDB version comes out. And nowadays, with all the AI tooling around, building an extension has never been easier.

Distributing Extensions

So you have written your extension. But how should you distribute it? Until now, DuckDB could only install extensions from the built-in repositories (`core`, `core_nightly`, `community`, ...). In v2.0, you will be able to register your own trusted repositories (#24777, currently work-in-progress), so an organization can host and sign its own extensions and have them install and load just like the built-in ones:

```
SET allow_extension_repositories = 'allowed';
```

```
CREATE EXTENSION REPOSITORY my_repo
    WITH PREFIX 'https://extensions.example.org';

INSTALL my_ext FROM my_repo;
LOAD my_ext FROM my_repo;
```

A repository is a name, a URL prefix, and one or more RSA public keys that are trusted to sign the extensions served from it. The prefix can point at anything DuckDB can read: a local path, `https`, `s3`, you name it. At `CREATE` time, DuckDB fetches the repository's public keys and pins them into the repository definition, printing each key's SHA-256 fingerprint so you can compare it against one published out of band. If you would rather not trust the network at all, you can pass the key directly:

```
CREATE EXTENSION REPOSITORY my_repo
    WITH PREFIX 'https://extensions.example.org'
    USING PUBLIC KEY '-----BEGIN PUBLIC KEY----- ...';
```

Pinned repositories survive restarts, support key rotation by trusting multiple keys, and can be audited at any time through the `duckdb_extension_repositories()` table function, or removed again with `DROP EXTENSION REPOSITORY`. Together with the stable C API, the extension story rounds out nicely: write your extension once, sign it, host it wherever you like, and `INSTALL` it anywhere.

> **Update.** This feature is available in DuckDB v2.0.0. For details, see the Extension Repositories page.

Bonus: DuckDB Foundation – Technical Advisory Board

Starting this fall, we will add a Technical Advisory Board to the DuckDB Foundation. The board will provide input on the development roadmap of DuckDB, DuckLake, and Quack. This allows key stakeholders to have a say in the projects' direction.

Final Thoughts

These are only a few highlights, and this post is only a preview. Some details may still shift before the release this fall, and there are many more features and improvements that we could not cover here. DuckDB v2.0 will also come with a small set of breaking changes, including the new default storage format and the completed lambda syntax transition, which we will cover in detail in the release announcement.

There have been more than 10,000 commits by many contributors since we released v1.5. We would like to thank our community for the detailed issue reports, feedback, and contributions that shaped this release. DuckDB v2.0.0 is now available on the installation page is.

Reconciling JSON in DuckDB, One Patch at a Time

Publication date: 2026-08-18

Author: Mustafa Khan

TL;DR: DuckDB v2.0 will ship JSON functions that allow you to easily apply JSON patches.

> Guest blog post by Mustafa Khan (Atlan).

DuckDB's JSON extension already covers reading JSON, extracting paths, applying RFC 7396 merge patches, and the usual scalar accessors. The upcoming v2.0 release will extend it with four new scalar functions: `json_merge_patch_diff` computes the inverse of an RFC 7396 merge patch, `json_deep_merge` applies patches with skip-on-`null` semantics, `json_normalize` canonicalizes key order, and `json_strip_nulls` recursively removes `null`-valued keys. You can try these primitives today by installing the preview version of DuckDB v2.0-dev.

These primitives address a handful of recurring problems in data pipelines that reconcile state between systems. The examples in this post come from Atlan. Atlan is a *context layer* for AI: a platform that gathers metadata, lineage, and semantics from a company's data systems so that data teams and AI agents can find and understand the data. Keeping that layer up to date means constantly reconciling entity documents from many upstream sources. Two upstream services may describe the same entity with different key orderings. Partial updates may carry `null` in a field to indicate either deletion or absence of data. Most events change only one or two fields out of dozens. Handling these cases in SQL previously required leaving the database. The four new functions allow you to perform the work in a single query.

In the rest of this post, we walk through each function, then chain them into an end-to-end reconciliation example. Finally, we present the results of a simple benchmark against equivalent Python implementations on 500,000 synthetic change data capture (CDC) events.

`json_merge_patch_diff`: The Inverse of `json_merge_patch`

`json_merge_patch_diff(orig, modified)` returns the minimal RFC 7396 patch such that `json_merge_patch(orig, patch) = modified`. `json_merge_patch` already exists in DuckDB. It applies an RFC 7396 patch to a document: a `null` value in the patch deletes the key, nested objects are merged recursively, and any other value overwrites the original. Deleted keys appear as `null` in the patch, changed and added keys appear with their new values, and unchanged keys are omitted entirely.

```
SELECT json_merge_patch_diff('{"a":1,"b":2,"c":3}', '{"a":1,"b":99,"d":4}');
```

```
{"c":null,"b":99,"d":4}
```

a is unchanged, so it is omitted. b changed. c was removed, so it becomes `null`. d is new.

The function recurses into nested objects, producing patches that only touch the changed paths:

```
SELECT json_merge_patch_diff(
    '{"user":{"name":"Alice","age":30}}',
    '{"user":{"name":"Alice","age":31}}'
);
```

```
{"user":{"age":31}}
```

The round trip with `json_merge_patch` holds by construction:

```
SELECT json_merge_patch(
    '{"a":1,"b":2,"c":3}',
    json_merge_patch_diff('{"a":1,"b":2,"c":3}', '{"a":1,"b":99,"d":4}')
) = '{"a":1,"b":99,"d":4}' AS round_trips;
```

```
true
```

Equality between subtrees is computed with `yyjson_equals`, which compares two `yyjson` values structurally without serializing either side. Subtrees that compare equal contribute nothing to the output and short-circuit the recursion:

```
SELECT json_merge_patch_diff(
    '{"a":{"b":{"c":{"d":1}}}}',
    '{"a":{"b":{"c":{"d":1}}}}'
);
```

```
{}
```

This function is particularly useful for CDC pipelines, where most events on a metadata catalog touch one or two fields out of dozens. Shipping the output of `json_merge_patch_diff(prev_state, new_state)` downstream, instead of the full new state, cuts the change payload to a small fraction of its previous size. The diff is the change itself, so there is nothing extra to filter on the consumer side, and a `json_merge_patch` on the other end reconstructs the new state from `prev_state` and the patch.

`json_deep_merge`: Recursive Merge where `null` Means "Skip"

The patches from the previous section are applied with `json_merge_patch`, which strictly follows RFC 7396: `null` in the patch deletes the key. That is the right rule for the round trip with `json_merge_patch_diff`, but it is the wrong rule when you are reconciling fragments from multiple upstreams that emit `null` to mean "I do not have a value for this field on this message." `json_deep_merge` covers that case.

`json_deep_merge` follows the same recursive merge structure as `json_merge_patch`, with one difference: a `null` value in the patch means "keep the original value" instead of "delete the key". Everything else, including handling of non-null values, nested objects, and the variadic argument shape, matches `json_merge_patch`. In the following example, we apply a patch with the key b set to `null` with `json_merge_patch` and `json_deep_merge`, respectively, resulting in two different results:

```
SELECT json_merge_patch('{"a":1,"b":2}', '{"b":null}') AS rfc_7396;
```

```
{"a":1}
```

```
SELECT json_deep_merge('{"a":1,"b":2}', '{"b":null}') AS deep_merge;
```

```
{"a":1,"b":2}
```

Both functions replace the original value with the patch value when the patch contains a non-null value, and both recurse into nested objects when the original and patch are both objects. The handling of `null` is the only divergence:

```
SELECT json_deep_merge(
    '{"a":{"x":1,"y":2}}',
    '{"a":{"y":null,"z":3}}'
);
```

```
{"a":{"x":1,"y":2,"z":3}}
```

The nested `y` is preserved because the patch contained `null`. `z` is added because the patch had a real value. The same patch through `json_merge_patch` deletes `y` instead:

```
SELECT json_merge_patch(
    '{"a":{"x":1,"y":2}}',
    '{"a":{"y":null,"z":3}}'
);
```

```
{"a":{"x":1,"z":3}}
```

The function is variadic: pass any number of patches and they apply left to right.

```
SELECT json_deep_merge(
    '{"a":1}',
    '{"a":null}',
    '{"a":2}'
);
```

```
{"a":2}
```

The first patch skips (keeps `a=1`), the second overwrites (`a` becomes 2).

The use case that motivates the skip-on-null rule is reconciling fragments from multiple sources. One source knows the column name (`columnName`) but not its parent table (`parentColumn`). Another knows the parent but not the column name. Each emits a fragment with the unknown fields as `null`:

```
SELECT json_deep_merge(
    '{"columnName":"user_id","parentColumn":null}',
    '{"columnName":null,"parentColumn":"accounts.id"}'
);
```

```
{"columnName":"user_id","parentColumn":"accounts.id"}
```

Note that passing SQL `NULL` as an argument and using JSON `null` values result in different behaviors. A SQL `NULL` patch makes the result `NULL`, and a SQL `NULL` original is ignored, matching the behavior of `json_merge_patch`. JSON `null` inside a patch means "keep the original value for this key".

`json_normalize`: Canonical Form for Hashing

The third primitive answers the first of the questions this post opened with: when two services emit the same JSON object with different key orders, are they the same document? Lexically they are not, which means comparing the content hashes of the raw strings will conclude they differ. `json_normalize` fixes that by recursively sorting the keys of every object, including objects nested inside arrays. Array elements keep their order, since element order is meaningful in JSON and key order is not. Scalar values are returned unchanged.

```
SELECT json_normalize('{"z":1,"a":2,"m":3}');
```

```
{"a":2,"m":3,"z":1}
```

Recursion descends into nested objects, and into objects nested inside arrays:

```
SELECT json_normalize('{"c":{"b":{"z":1,"a":2},"a":3},"a":4}');
```

```
{"a":4,"c":{"a":3,"b":{"a":2,"z":1}}}
```

```
SELECT json_normalize('[{"z":1,"a":2},{"y":3,"b":4}]');
```

```
[{"a":2,"z":1},{"b":4,"y":3}]
```

Once two semantically-equal documents normalize to the same bytes, hashing them collapses duplicates trivially:

```
SELECT
    md5(json_normalize('{"z":1,"a":2}')) =
    md5(json_normalize('{"a":2,"z":1}')) AS same_hash;
```

```
true
```

A `GROUP BY md5(json_normalize(payload))` collapses duplicate emissions to a single row, deduplicating entities that differ only in key ordering. This is useful for a metadata catalog where the same entity may be described by two different upstream sources with slightly different key orderings.

Internally, the function walks the mutable doc, sorts each object's key-value pairs with `std::sort`, and rebuilds the object in order.

`json_strip_nulls`: Recursively Drop Null-Valued Keys

`json_strip_nulls` is a cleanup helper. It removes every key whose value is JSON `null`, recursively. Array elements are not modified (since `null` is a valid array element), and scalar values are returned unchanged.

```
SELECT json_strip_nulls('{"a":1,"b":null,"c":null,"d":2}');
```

```
{"a":1,"d":2}
```

It descends into nested objects:

```
SELECT json_strip_nulls('{"a":{"x":1,"y":null},"b":2}');
```

```
{"a":{"x":1},"b":2}
```

And into objects inside arrays, while leaving the array structure intact:

```
SELECT json_strip_nulls('{"a":[{"x":1,"y":null},{"z":null}]}');
```

```
{"a":[{"x":1},{}]}
```

The motivating case is cleaning patches before storing them. After computing a `json_merge_patch_diff`, the patch contains `null` entries for deleted keys, which is correct under RFC 7396. If you only want to ship the additions and updates, `json_strip_nulls(json_merge_patch_diff(orig, modified))` removes those `null` entries from the patch. The resulting patch can add and update fields but never delete them. The same idea applies to trimming verbose API responses before hashing or storing them.

Under the hood, this uses `yyjson_mut_obj_iter_remove`, which deletes keys during iteration without invalidating the iterator. Calling `json_strip_nulls` on an already-clean document reduces to a parse, one pass, and a serialize.

Composing Them: An End-to-End Reconciliation

The four functions compose into a small reconciliation workflow. Suppose an upstream system emits partial JSON documents on each change. Some fields are present with new values, some are absent because the upstream did not include them, and some are explicitly `null` because the upstream has no data for them on this event. The goal is to clean the event, compute the minimal patch against the previous state, ship that patch, and store a canonical hash of the resulting document so identical entities collapse on deduplication.

Let's see this workflow through a concrete example:

```
CREATE TABLE catalog (id INTEGER, state JSON);
INSERT INTO catalog VALUES (
    1,
    '{"typeName":"Column","name":"user_id","description":"primary
    key","dataType":"BIGINT","ownerEmail":"alice@example.com"}'
);

CREATE TABLE incoming_events (id INTEGER, event JSON);
INSERT INTO incoming_events VALUES (
    1,
    '{"name":"user_id","description":"primary key for the users
    table","dataType":"BIGINT","ownerEmail":null,"team":null}'
);
```

Step 1: strip the `null` placeholders out of the incoming event so they do not get treated as deletes:

```
SELECT json_strip_nulls(event) AS clean_event
FROM incoming_events
WHERE id = 1;
```

```
{"name":"user_id","description":"primary key for the users table","dataType":"BIGINT"}
```

Step 2: produce a minimal patch against the current state. Deep-merge the cleaned event onto the original (so missing fields keep their previous value), then diff back against the original:

```
SELECT json_merge_patch_diff(
    c.state,
    json_deep_merge(c.state, json_strip_nulls(e.event))
) AS patch
FROM catalog c
JOIN incoming_events e USING (id);
```

```
{"description":"primary key for the users table"}
```

Only the changed field appears. `dataType` was unchanged, `ownerEmail` and `team` were `null` and got stripped, so neither are returned. The patch is what you would ship downstream.

Step 3: apply the patch back and hash the normalized result:

```
SELECT
    md5(json_normalize(
        json_merge_patch(
            c.state,
            '{"description":"primary key for the users table"}'
        )
    )) AS content_hash
FROM catalog c
WHERE id = 1;
```

```
21ceaced24d8958d71d537f6fcebe8a2
```

Each function performs a single transformation, and the whole chain runs in a single query.

Benchmarks: DuckDB vs. Python on 500,000 CDC Events

The benchmark below compares each function and the full composed chain against equivalent Python implementations on 500,000 synthetic catalog-event pairs (roughly 600 bytes per document). The Python implementations parse with `json.loads`, apply the equivalent transformation in pure Python (the Python output matches DuckDB's exactly on 100 sample documents), and serialize back with `json.dumps`. That is the closest apples-to-apples comparison to what DuckDB does end-to-end on a JSON column.

Function	Python (s)	DuckDB (s)	Speedup
`json_normalize`	7.02	0.15	46.8x
`json_deep_merge`	8.45	0.47	18.0x
`json_merge_patch_diff`	6.66	0.19	35.1x
`json_strip_nulls`	6.17	0.05	123.4x
Full chain (composed)	11.11	1.03	10.8x

The benchmark methodology is as follows. Each measurement is the median of five iterations. Python timings include parse, transform, and serialize per row, with data pre-loaded into memory so disk I/O is excluded. For DuckDB, we timed the whole run from the command line and subtracted a no-op `SELECT json` run, so only the transformation is measured. Hardware is an Apple M3 Pro (arm64) with 18 GB RAM, Python 3.11. DuckDB runs multi-threaded by default, while the Python script is single-threaded. The benchmark code and synthetic data generator are available on GitHub.

The speedup comes from two sources: DuckDB operates on `yyjson` trees in place, while the Python equivalent spends most of its cycles in `json.loads` and `json.dumps`. The result is a speedup of about 1-2 orders of magnitude, depending on how much transformation work the function does. `json_strip_nulls` is the extreme case, since the work after parsing reduces to a single iterator pass with an in-place delete. On the composed chain, the per-function gap narrows because the absolute work goes up on both sides, but DuckDB still finishes the full reconciliation of 500,000 event pairs in roughly one second.

Performance Notes

There are a few characteristics worth pointing out:

- All four functions operate on `yyjson` mutable values directly.
- `json_merge_patch_diff` uses `yyjson_equals` for deep equality, and subtrees that compare equal stop the recursion.
- `json_normalize`'s sort is per-object, not global. The cost scales with the size of each object, not the document.
- All four are plain scalar functions that process one vector of rows at a time, so they fit DuckDB's vectorized execution model.

Trying the Functions

The four functions will ship in DuckDB v2.0. To try them today, install a v2.0-dev preview build, then load the JSON extension:

```
LOAD json;

SELECT json_normalize('{"z":1,"a":2}');
```

Wrapping Up

These four functions extend the DuckDB JSON extension with primitives for diffing, merging, normalizing, and stripping `null` values. They compose into reconciliation pipelines that previously required transformations outside SQL.

That's not all. The functions `json_set(json, path, value)`, `json_remove`, `json_insert`, and `json_replace` are also available in DuckDB v2.0 preview (#23786), completing the set of edit primitives defined in the SQL/JSON specification. For the implementations, see the JSON extension source in `extension/json`.

Feedback and bug reports are welcome on Discord or GitHub.

DuckDB v2.0: Your Database Deserves a Better Parser

Publication date: 2026-08-20

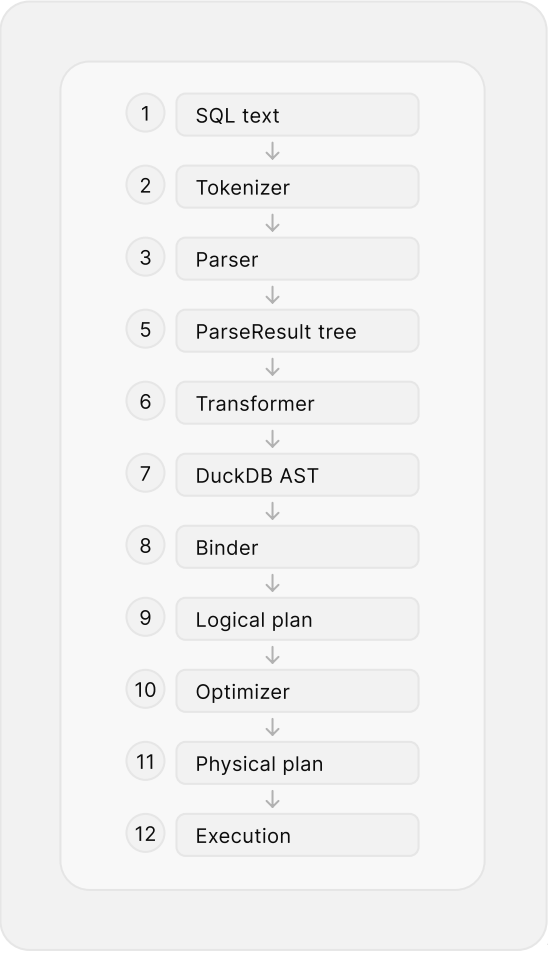
Author: Daniël ten Wolde

TL;DR: DuckDB v2.0 replaces its PostgreSQL-derived SQL parser with a PEG-based parser that is easier to evolve and can be extended at runtime.

At DuckDB, one of our goals is to make working with a database system as easy as possible. Users interact with the system through the widely understood Structured Query Language (SQL). Previous blog posts have covered DuckDB's friendly SQL, including `GROUP BY ALL` and column selection using `SELECT * EXCLUDE (...)`. Before DuckDB can execute a query using these features, however, it first has to determine whether its syntax is valid. That is the job of the *parser*, and in DuckDB v2.0 we are completely replacing it without you noticing.

What is the Role of a Parser?

At a high level, DuckDB processes a SQL query through the following stages:



{: .lightmode-img }

In this blog, we focus on the tokenizer, parser, and transformer:

- Tokenizer: This is the first step and is responsible for splitting up the raw input string into *tokens*. These can be of various categories, for example: `KEYWORD`, `NUMBER`, or `IDENTIFIER`. It is also where comments, in SQL denoted with either `--` or `/* */`, are recognized and skipped.
- Parser: The parser determines whether these tokens follow DuckDB's grammar and produces a `ParseResult` tree.
- Transformer: Converts the generic parse results into DuckDB's internal abstract syntax tree (AST), forming structures such as `SQLStatement`, `TableRef`, and `ParsedExpression`. The resulting AST is passed on to the binder.

The parser determines whether a query is syntactically valid, while the binder determines whether the tables, columns, and functions it refers to actually exist.

Consider the following query:

```
SELECT *
WHERE true
FROM range(1);

Parser Error:
syntax error at or near "FROM"

LINE 3: FROM range(1);
        ^^^^
```

Every individual *token* in this query is valid, but the clauses occur in an order that DuckDB's grammar does not accept. Friendly SQL allows both `SELECT`-first and `FROM`-first syntax, but it does not allow the clauses to appear in an arbitrary order.

By comparison, the following query is syntactically valid, so it passes the parser and transformer. However, it fails later in the binder because the table `missing_table` does not exist.

```
FROM missing_table;

Catalog Error:
Table with name missing_table does not exist!

LINE 1: FROM missing_table;
             ^^^^^^^^^^^^
```

The DuckDB SQL Dialect

Although a SQL standard exists, every database system supports different parts of the standard and adds its own syntax and behavior. The resulting variants are commonly referred to as SQL dialects. Examples include the dialects supported by PostgreSQL, Oracle, GoogleSQL for BigQuery, MySQL, MariaDB, SQLite, Spark SQL, and, of course, DuckDB.

DuckDB's SQL closely follows PostgreSQL conventions, but it has evolved considerably over the years. We have added features of our own, such as `GROUP BY ALL`, as well as features inspired by other database systems. At the same time, DuckDB does not implement every aspect of PostgreSQL's behavior. DuckDB therefore speaks its own SQL dialect, which we will refer to as **DuckSQL** in this post, even though it remains strongly influenced by PostgreSQL.

This distinction is important when talking about the parser. The SQL dialect that DuckDB accepts and the implementation used to parse that SQL are two separate things. For DuckDB v2.0, we are replacing the parser implementation and rewriting its grammar. What we are **not** replacing is DuckSQL itself.

Outgrowing the PostgreSQL-Derived Parser

When DuckDB started out, it made a lot of sense to use the PostgreSQL-derived parser and grammar. This parser was already part of the first commit to DuckDB in 2018. It gave DuckDB a mature, battle-tested SQL grammar based on syntax that many users were already familiar with. We adapted the parser to our needs and added a `Transformer` that converted the resulting PostgreSQL-style parse tree into DuckDB's internal AST.

However, over the years this parser also came with some downsides. Extending DuckSQL meant modifying the underlying YACC/Bison grammar. Because Bison generates an LALR(1) parser, seemingly small additions to the grammar can interact with existing rules and introduce `shift/reduce` or `reduce/reduce` conflicts. As DuckSQL grew, making changes to the grammar therefore became increasingly difficult.

This was one of the motivations behind our earlier blog post on runtime-extensible SQL parsers. In that post and the accompanying CIDR paper, we explored whether Parsing Expression Grammars (PEGs) could provide a better foundation for an extensible database parser. At the time, the PEG parser was still an experimental prototype capable of parsing only a subset of SQL.

A Primer on PEG Parsers

Before looking at how we turned the prototype into a production parser, let us briefly revisit how a PEG describes a language.

A PEG consists of named rules that describe how an input should be matched. Consider the following rules from DuckDB's new grammar:

```
SelectFrom <- SelectFromClause / FromSelectClause
SelectFromClause <- SelectClause FromClause?
FromSelectClause <- FromClause SelectClause?
```

The `<-` operator defines a rule, `/` specifies a choice between alternatives, and `?` makes an element optional. Together, these rules state that DuckSQL accepts both a traditional `SELECT`-first query:

```
SELECT *
FROM range(1);
```

And DuckDB's Friendly SQL `FROM`-first equivalent:

```
FROM range(1)
SELECT *;
```

A PEG evaluates alternatives in order. When matching `SelectFrom`, the parser first attempts `SelectFromClause`. If that does not match, it attempts `FromSelectClause`. The first successful alternative is selected. As a result, PEG grammars do not have the same `shift/reduce` and `reduce/reduce` conflicts as LALR grammars. Instead, alternatives are ordered explicitly, and that order forms part of the grammar's behavior.

We are not the only ones changing to a PEG-based parser. Python switched from its LL(1) parser to a PEG-based parser in Python 3.9, also motivated by the additional flexibility PEG provides to evolve the language.

In DuckDB, these rules operate on the tokens produced by the tokenizer. The matcher applies the grammar rules to those tokens and constructs a generic `ParseResult` tree, which is subsequently transformed into DuckDB's internal AST.

Going from Prototype to Production

The research prototype demonstrated that a PEG-based SQL parser was feasible. Replacing DuckDB's existing parser, however, required considerably more than parsing a subset of SQL. The new parser had to accept all of DuckSQL and produce the same AST expected by DuckDB's binder.

The PEG grammar was first introduced in DuckDB `v1.2`, where it handled autocomplete in the CLI. Later, in DuckDB `v1.5`, we introduced the complete PEG parser as an experimental, opt-in feature. We also used it for an April Fools' joke that made DuckDB speak Dutch. Since then, the grammar, matcher, and transformer have been steadily improved to make the PEG parser the default for DuckDB v2.0.

Among other things, the parser had to support:

- **Every statement and expression type:** Supporting the complete DuckSQL dialect includes both common syntax as well as the less frequently used statements and expressions.
- **Operator precedence and associativity:** For example, `SELECT true OR true AND false;` must be interpreted as `(true OR (true AND false))`, because `AND` binds more tightly than `OR`.
- **Correct keyword classification:** Some keywords, such as `SELECT`, are `RESERVED` and cannot be used as unquoted table or column names. Other keywords may be used as identifiers depending on their context.

- **Compatibility with DuckDB's internal AST:** The PEG transformer must produce the same DuckDB AST structures as the transformer for the PostgreSQL-derived parse nodes wherever the language behavior is intended to remain unchanged.
- **Correct error reporting:** For an invalid query, the parser should report where parsing failed and, where possible, provide context and a useful indication of what went wrong. Ideally, it should do so without pointing to a manual.
- **Performance on unusual inputs:** Besides keeping normal parsing fast, we also had to make sure that malformed queries do not suddenly take a long time to parse.

Avoiding Repeated Work with Packrat Parsing

One issue we encountered was repeated work during backtracking. A naïve PEG matcher can evaluate the same grammar rule at the same token position many times while trying different alternatives. For certain malformed inputs, the amount of repeated work can grow exponentially.

We encountered this with queries containing a large number of unmatched opening parentheses:

```
SELECT ((((((((((((((((((;
```

With the experimental PEG parser shipped in `v1.5`, adding one more opening parenthesis approximately doubled the parsing time:

```
18 opening parentheses:  5.303 seconds
19 opening parentheses: 10.640 seconds
```

We addressed this using **packrat parsing**, a memoization technique commonly used with PEG parsers. For each memoized matcher, we store the result of applying it at a particular token position. If the parser later attempts the same matcher at the same position, it reuses the cached result instead of evaluating it again.

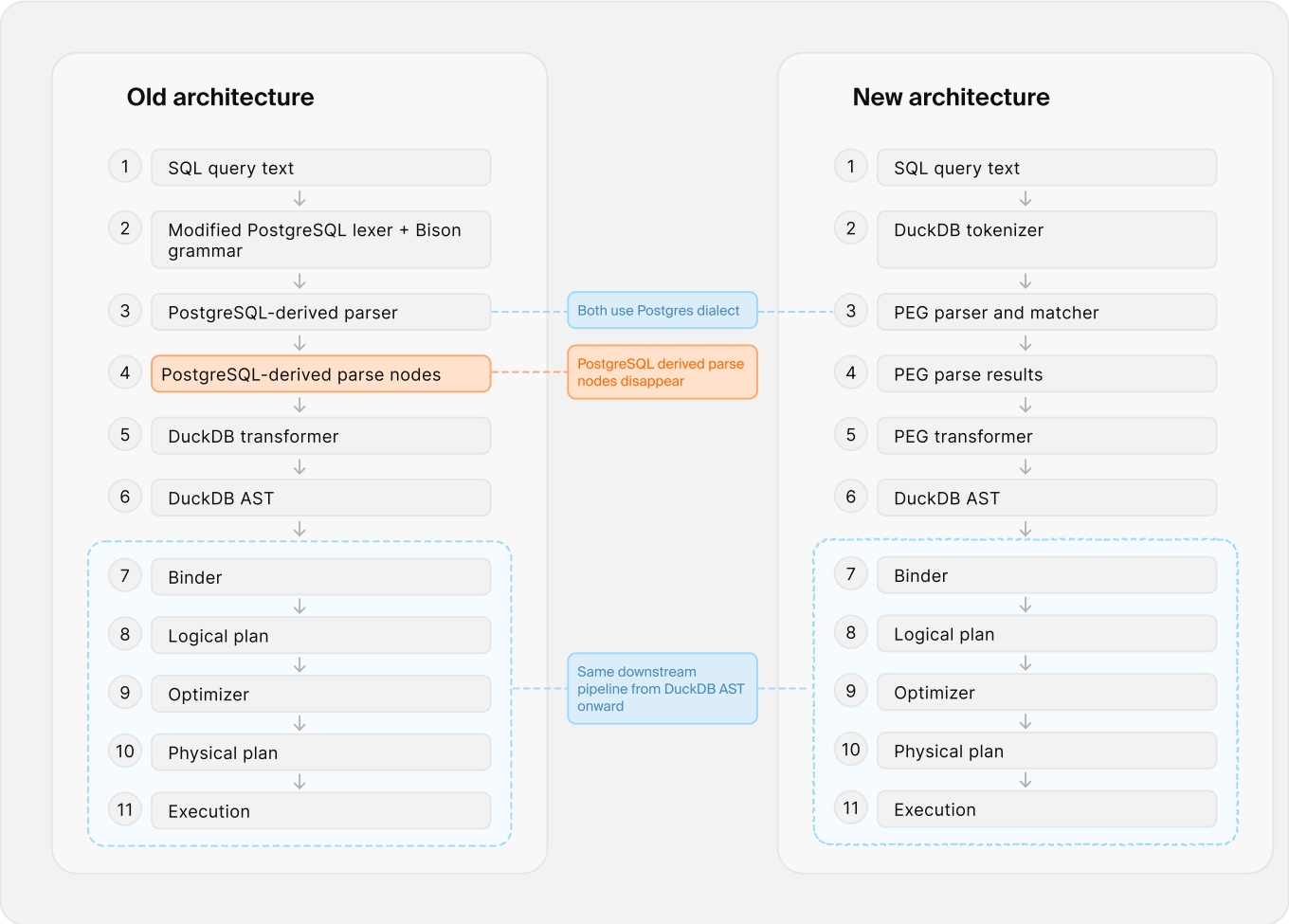
With packrat parsing enabled, the same malformed query was rejected almost instantly:

```
19 opening parentheses: 0.001 seconds
```

As a result, a memoized matcher is evaluated at most once at a particular token position, removing the repeated work that caused the exponential behavior in this example. This requires additional memory while parsing, but that is a worthwhile trade-off for avoiding cases such as this.

Turning the prototype into a production parser involved much more than translating the grammar. The new parser had to cover the complete DuckSQL dialect, preserve DuckDB's existing AST, remain compatible with existing queries, and handle both valid and malformed input efficiently.

The resulting architecture replaces the PostgreSQL-derived parser front end, while the binder and the remainder of DuckDB's query-processing pipeline continue to operate on the same internal AST.



{: .lightmode-img }

DuckDB Parser architecture. Key idea: replace the PostgreSQL-derived parse front end while keeping the rest of DuckDB's execution pipeline the same.{: .caption }

Evolving DuckSQL

With the PEG parser now in place for DuckDB v2.0, we have also continued to extend DuckSQL with new syntax.

One example is the new expression-statement syntax. Until now, executing a query consisting only of expressions always required writing a `SELECT`:

```
SELECT date: current_date(), time: current_localtime();
```

With an expression statement, the `SELECT` can be omitted:

```
date: current_date(), time: current_localtime();
```

As a bonus, this also works with prefix aliases.

Another example is the new `CONNECT` statement, introduced for Quack. It allows you to connect to a remote database and route subsequent queries to it until you run `DISCONNECT`:

```
CONNECT 'postgres://localhost/mydb';
SELECT count(*) FROM orders; -- Runs on the PostgreSQL server
DISCONNECT;
```

There will also be new syntax for working with external resources. This will allow you to manage resources that live outside DuckDB through an extension. You will be able to create, register, inspect, connect to, or destroy a resource all from within DuckDB:

```
CREATE EXTERNAL RESOURCE '<resource-type>' AS <name> (...);
REGISTER EXTERNAL RESOURCE '<resource-type>' AS <name> FROM <handle>;

SHOW EXTERNAL RESOURCES;

CONNECT TO EXTERNAL RESOURCE <name>;

DESTROY EXTERNAL RESOURCE <name>;
```

We have also extended `COPY TO` with `PARTITION BY` and `ORDER BY` syntax:

```
COPY orders TO 'orders'
(
    FORMAT parquet,
    PARTITION BY (year, month),
    ORDER BY (order_date)
);
```

These additions would also have been possible with the old PostgreSQL-derived parser, but adding them would have been considerably more cumbersome. The PEG grammar makes it easier for us to continue evolving DuckSQL.

So far, these rules are all part of DuckSQL itself. The next step is allowing extensions to add rules of their own.

Extending the Parser

Extensions are a central part of DuckDB. They can already add scalar and table functions, optimizer rules, query-plan rewrites, and even custom physical operators.

Extensions that add new syntax already exist, such as `psql` and `duckpgq`, but under the hood they work as fallback parsers. DuckDB first tries to parse the query itself and only calls the extension if that fails. This works well for self-contained syntax, but an extension that wants to add syntax inside SQL also has to parse the surrounding SQL itself. These fallback parsers also make it impossible to combine the syntax of multiple extensions.

With the PEG parser, extensions can instead extend individual parts of DuckDB's parser. They can extend the tokenizer, add grammar rules, and register custom matchers while continuing to reuse the rest of DuckSQL.

> **Warning.** The API shown below is still a preview and may change before DuckDB v2.0. You can follow the ongoing development on GitHub.

To make this concrete, we use Google's pipe query syntax. This is an extension to SQL that adds piped data flow syntax. Pipe syntax expresses a query as a sequence of operators, where each operator consumes the result of the previous one.

```
FROM produce
  |> WHERE
       item != 'bananas'
       AND category IN ('fruit', 'nut')
  |> AGGREGATE count(*) AS num_items, sum(sales) AS total_sales
     GROUP BY item
  |> ORDER BY item DESC;
```

A simplified PEG grammar for this needs a handful of rules:

```
PipeSelectAtom <- PipeSource PipeStage+
PipeSource <- FromClause / SelectStatementType / SelectParens
PipeStage <- '|>' PipeOperator
PipeOperator <- PipeAggregate / PipeAggregateGroupOnly / PipeWhere / PipeSelect / PipeExtend / PipeDistinct
/ PipeOrderBy / PipeLimit
PipeWhere <- WhereClause
PipeSelect <- 'SELECT' TargetList
PipeExtend <- 'EXTEND' TargetList
PipeDistinct <- 'DISTINCT'
PipeOrderBy <- OrderByClause
PipeLimit <- LimitClause OffsetClause?
```

```
PipeAggregate <- 'AGGREGATE' TargetList GroupByClause?
PipeAggregateGroupOnly <- 'AGGREGATE' GroupByClause
```

Here, + means that `PipeStage` must occur one or more times, so a pipe query must contain at least one pipe operator.

This grammar can reuse existing rules, such as `GroupByClause`, to reduce the amount of grammar the extension needs to define. An extension can still define its own rule where DuckDB's existing syntax does not fit.

Registering the Grammar

Defining just the PEG rules does not yet make them part of DuckDB's grammar. The extension must also specify (1) the existing grammar rule it wants to extend and (2) the transformer rules that convert the new syntax into DuckDB's AST.

In the current prototype, certain grammar rules expose extension points. Pipe SQL registers `PipeSelectAtom` as an additional alternative for `SelectAtom`, together with the new keywords `AGGREGATE` and `EXTEND`.

```
static void LoadInternal(ExtensionLoader &loader) {
    ParserExtension extension;
    extension.grammar_extension.grammar = PIPE_SQL_GRAMMAR;
    extension.grammar_extension.select_atom_rule = "PipeSelectAtom";

    extension.grammar_extension.RegisterSelectAtomTransformer(
        "PipeSelectAtom",
        TransformPipeSelectAtom
    );

    loader.RegisterKeyword(
        "aggregate",
        ExtensionKeywordCategory::RESERVED
    );
    loader.RegisterKeyword(
        "extend",
        ExtensionKeywordCategory::RESERVED
    );

    loader.RegisterParserExtension(std::move(extension));
}
```

By registering this alternative, the resulting grammar is effectively:

```
SelectAtom <-
    PipeSelectAtom /
    SelectParens /
    SelectStatementType
```

The extension alternative is now tried first. If no pipe syntax is present, it fails without consuming any tokens and the query is parsed with the built-in alternatives.

Transforming the Result

Adding a grammar rule only gets us as far as a `ParseResult`. The extension still needs to transform that result into the DuckDB AST that is expected by the binder. Since `PipeSelectAtom` extends `SelectAtom`, its transformer returns a `SelectStatement`:

```
static unique_ptr<SelectStatement>
TransformPipeSelectAtom(PEGTransformer &transformer, ParseResult &parse_result) {
    auto &pipe = parse_result.Cast<ListParseResult>();

    // PipeSelectAtom <- PipeSource PipeStage+
    auto statement = TransformPipeSource(transformer, pipe.GetChild(0));

    auto &stages = pipe.Child<RepeatParseResult>(1);
    for (auto &stage : stages.GetChildren()) {
```

```
        ApplyPipeStage(transformer, stage.get(), *statement);
    }

    return statement;
}
```

The shape of the `ParseResult` follows the grammar rule we defined earlier. `PipeSelectAtom` contains a `PipeSource` and one or more `PipeStages`. We first transform the `PipeSource` into a DuckDB `SelectStatement`. Each `PipeStage` is then applied to that statement in order. The resulting `SelectStatement` is then returned and can continue through the rest of the parser's pipeline and eventually on to the binder.

This is where reusing DuckDB's existing grammar becomes especially useful. The extension only needs to transform the new syntax it introduced. When it reuses an existing DuckDB grammar rule, such as `GroupByClause`, it can also reuse the corresponding transform function instead of having to implement GROUP BY itself.

This is an important difference from the fallback parsers that are available today. An extension no longer needs to implement expressions, table references, GROUP BY clauses, and the rest of SQL itself. Instead, it can add only the syntax it needs and reuse DuckDB's grammar and transformations for everything else.

Executing Pipe SQL

With the extension registered, we can now execute queries using the new pipe syntax. For example, we can combine the pipe operators added by the extension with existing DuckSQL features such as `range()` and prefix aliases:

```
FROM range(6) t(i)
  |> WHERE i % 2 = 0
  |> SELECT i, doubled: i * 2
  |> ORDER BY i DESC;
```

i int64	doubled int64
4	8
2	4
0	0

The extension only defines the pipe-specific syntax. Expressions, table references, `WHERE`, `SELECT`, `ORDER BY`, and other reused rules are still parsed and transformed by DuckDB itself. This means that new syntax can be combined with DuckSQL without the extension having to implement the rest of SQL again.

To Conclude

With DuckDB v2.0, we are replacing the PostgreSQL-derived parser with a new PEG parser. Existing DuckSQL queries should continue working as before. Under the hood, however, the new parser gives us something that is easier to evolve and designed for runtime extensibility.

The runtime grammar extension API shown in this post is still a preview and may change before v2.0 is released. However, the underlying idea is already working. Extensions can add their own syntax directly to DuckDB's grammar while reusing its existing rules and transformations. This means they no longer need to parse the rest of SQL themselves.

We are excited to see what new syntax the community will create. In the meantime, we will continue evolving DuckSQL and improving the parser.

If you do find an existing query that behaves differently with the PEG parser, please let us know by filing an issue.

Chunked Query Results in the DuckDB Java Driver

Publication date: 2026-08-21

Authors: Geertjan Wielenga, Alex Kasko

TL;DR: The DuckDB Java driver can now return query results as a lazily fetched sequence of columnar data chunks, avoiding JDBC's row-at-a-time ResultSet and its per-value overhead.

DuckDB is a columnar, vectorized database. Every operator inside the engine works on *data chunks:* batches of column vectors, 2,048 rows at a time. That is a big part of why DuckDB is fast: the engine amortizes interpretation overhead over thousands of values instead of paying it once per value.

JDBC, on the other hand, was designed in 1997 around a very different idea. The `ResultSet` API hands you one row at a time (`next()`) and one value at a time (`getInt(1)`, `getString(2)` and so on). It is a solid, familiar API, supported across the Java ecosystem, but it forces the data into a form that DuckDB never used internally.

The DuckDB Java driver has to bridge the two. It embeds the native DuckDB library and talks to it over JNI. When the engine has already produced a columnar chunk of 2,048 rows, the JDBC specification requires the driver to slice it back into rows and cells before you can read it. If your application is going to put those values right back into columnar form, such as arrays, Arrow buffers or a machine learning feature matrix, the data is split into rows only to be reassembled into columns, wasting work on both sides.

Version 1.5.3.0 of the Java driver adds an alternative.

> **Tip.** We have thoroughly updated the Java client documentation, which now covers the driver's full API surface across dedicated pages: defining connections, running queries, handling results (including Arrow methods and streaming results), importing data with the appender and batch writer, and defining functions. The chunked query results discussed in this post are documented under Chunked Results.

Reading Results Row by Row

A typical JDBC read loop looks like this:

```
try (ResultSet rs = stmt.executeQuery("SELECT a, b FROM measurements")) {
    while (rs.next()) {
        long a = rs.getLong(1);
        double b = rs.getDouble(2);
        // ...
    }
}
```

Calling `next()` advances the cursor, and a `get*` call then locates the value inside the current native chunk and converts it to a Java representation. For some non-primitive types that conversion has to cross the JNI boundary on every value, and a few conversions are more involved, such as fetching a timestamp against a caller-supplied `java.util.Calendar` (the pre-`java.time` class the JDBC specification still mandates here). None of these steps is expensive on its own, but multiplied by millions of rows times a handful of columns, they add up. This is exactly the per-value interpretation overhead that vectorized execution is designed to avoid.

The loop also carries fixed overhead. So that a single connection can keep several result sets open at once, result streaming is off by default, which means a query's entire result is read into memory unless you set `jdbc_stream_results`, an option most users never set. The driver also assembles a full set of result metadata. That happens once per result rather than once per row, so it is cheap next to the row loop, but it is not free, and most queries never look at it.

The `ResultSet` path is the standard your ORM and the rest of the ecosystem already use, and for the vast majority of queries, the ones returning hundreds or thousands of rows, it is the right choice. The chunked API is for the other case: a query returns *a lot* of data, you control both ends of the pipe, and reading it row by row provides no advantage.

Reading Chunks with `DuckDBChunkedResult`

The Java driver now exposes the engine's native chunk stream directly, via the same mechanism the C API offers as `duckdb_fetch_chunk`. A query result becomes a lazily fetched sequence of data chunks, and you read column vectors out of each chunk in batches, the way the engine produced them, avoiding the per-row overhead required by the JDBC specification.

Putting that together:

```
try (DuckDBConnection conn = DriverManager
        .getConnection("jdbc:duckdb:")
        .unwrap(DuckDBConnection.class);
     DuckDBPreparedStatement ps = conn.prepare(
         "SELECT l_orderkey, l_linenumber, l_shipmode "
             + "FROM 's3://my-bucket-name/lineitems.parquet'")) {

    try (DuckDBChunkedResult res = ps.query()) {

        // advance to the next chunk, returns true on success
        while (res.nextChunk()) {

            // get the current chunk from the result
            DuckDBDataChunkReader chunk = res.chunk();

            // vectors are addressed by 0-based column index
            DuckDBReadableVector orderKeys = chunk.vector(0);
            DuckDBReadableVector lineNumbers = chunk.vector(1);
            DuckDBReadableVector shipModes = chunk.vector(2);

            // read each column with the getter for its type
            for (long row = 0; row < chunk.rowCount(); row++) {
                long orderKey = orderKeys.getLong(row);
                int lineNumber = lineNumbers.getInt(row);
                String shipMode = shipModes.getString(row);
                System.out.println(orderKey + " " + lineNumber + " " + shipMode);
            }
        }
    }
}
```

This approach has a few notable properties:

- **Laziness.** `nextChunk()` pulls one chunk at a time from the engine. The full result is never materialized on the native side or the Java side, so you can stream through results far larger than your heap, the same way the engine itself produces them.
- **Columnar access.** Inside a chunk you work vector by vector. If your destination is columnar too, such as a `long[]`, an Arrow `VectorSchemaRoot` or a Parquet writer, you copy values in tight, monomorphic loops instead of switching columns on every row.
- **Enough metadata to dispatch.** Each result still reports its column count and column types, so you can pick the right `get*` call per vector without carrying a separate schema description alongside the data.
- **A familiar API if you have written a UDF.** Chunk contents are accessed through the same `DuckDBDataChunkReader` API that the driver's user-defined functions use to read their input vectors. So the same code that reads a function's arguments inside a UDF also reads query results outside one.
- **Zero-based indexing.** Chunk *columns and rows* are 0-based, matching the C API and the UDF interfaces.

Inside a vector you are free to iterate however suits your code, a Java `Stream` included. A *parallel* stream over a single chunk rarely helps, though: 2,048 values is small enough that fork/join coordination usually costs more than it saves. If you want parallelism, apply it across chunks rather than within a single one.

Current Limitations

This is the first iteration of the API, and there are a few limitations you should know about before adopting it:

- **Basic data types only, for now.** The scalar types are supported. Composite types (`LIST`, `STRUCT`) are not yet readable through the chunked interface. Support for them is planned for a future release.
- **Prepared statements only.** `query()` currently exists on `DuckDBPreparedStatement`, and there is no `query(String)` convenience overload yet. Preparing the statement first adds one line.
- **A small reader surface, for now.** The reader covers the essential types today, with more coverage planned for future releases. One addition already planned is reading VARCHAR values directly as UTF-8 `byte[]`, so text-heavy workloads can skip materializing a Java `String` per value.

If any of these blocks a use case you care about, please open an issue. Your feedback is what drives the priorities for the next iteration.

Conclusion

The JDBC `ResultSet` remains the right default for most applications. But the engine produces its results in columnar chunks, and `DuckDBChunkedResult` lets your Java code read them in that same form: directly and lazily, with little overhead at the JNI boundary.

The feature is provided by duckdb-java#682 and is available in the current `duckdb_jdbc` releases on Maven Central. Try it on your largest result set and let us know how it performs. Issues and pull requests are welcome, and the DuckDB team is always happy to talk Java in the `#java` channel on the DuckDB Discord.

How DuckDB Runs Recursive CTEs Faster

Publication date: 2026-08-25

Author: Denis Hirn

TL;DR: DuckDB's recursive CTE engine now treats recursion as one long-lived computation: it retains eligible epoch-invariant state, chooses execution modes from exact frontier cardinalities and physical work, probes keyed state directly and gives `USING KEY ... UNION` changed-key semantics.

When I implemented DuckDB's first recursive CTE operator in 2020, correctness determined the design: evaluate the non-recursive term once, then evaluate the recursive term until the next working table is empty. That established the right semantic contract, but reusable runtime state was scoped too narrowly. Across iterations, the recursive input changes while most of the machinery that evaluates it remains reusable. The implementation nevertheless treated every iteration almost like a new query and repeatedly paid for pipeline scheduling, operator setup, execution and teardown. I have wanted to remove that mismatch ever since.

In the upcoming DuckDB v2.0, we assign those scopes explicitly. The query plan owns the physical operator tree, precomputed pipeline schedules and a pool of reusable pipeline executors. Each recursive invocation borrows executors from that pool and owns the accumulated recursive state and retained materializations whose lifetime spans the complete fixed-point computation, while each iteration owns only frontier-dependent state. This separation enables retained hash builds, per-iteration selection between inline and scheduled execution and direct probes into keyed state. The physical execution changes are accompanied by one semantic change: for `USING KEY`, `UNION` now makes new keys and keys whose finalized payload changed visible to the next iteration.

To illustrate the performance impact, we compare the original and new engines on a small reachability query. The table contains one million edges over 100,000 nodes. Ten identical edges leave each source node, and following them from node 0 visits 20,000 nodes:

```
CREATE TABLE edges AS
    SELECT (range % 100_000)::INTEGER AS src,
           ((range * 13 + 7) % 100_000)::INTEGER AS dst
    FROM range(1_000_000);

WITH RECURSIVE reachable(node) AS (
    SELECT 0
    UNION
    SELECT dst
    FROM edges, reachable
    WHERE src = node
)
SELECT count(*)
FROM reachable;
```

With the optimizations described below, median runtime falls from 4.051 seconds with DuckDB v1.5.5 to 0.095 seconds with the DuckDB v2.0 preview, a 42.6× speedup without changing the SQL.

Reusable Execution State Outlives the Frontier

Recursive CTE execution sits inside the query-plan lifetime. The physical operator tree belongs to that plan, and immutable schedule projections are built during physical pipeline construction. Within the plan, an invocation spans one complete fixed-point computation, while an epoch spans one evaluation of the recursive term. The original query plan already retained the recursive physical operator, but its runtime treated each epoch as a fresh execution: it recreated events and executors, reconstructed operator state and repeated scheduling work.

During pipeline construction, DuckDB derives immutable views of the pipeline dependency schedule, including the form that omits invocation-retained build pipelines after their first execution. We refer to these views as **schedule projections**. The runtime can therefore select the required dependency schedule without deriving it again for every epoch.

The Frontier Bounds Recursive Input

Every recursive **epoch** replaces the working table, often called the **frontier**, while the query remains fixed. The anchor term produces the first frontier. The recursive term reads it and produces candidates for the next one. With regular `UNION`, DuckDB rejects rows already seen; the survivors form the next frontier and also contribute to the logical **union table**. Evaluation stops when no rows survive.

The union table belongs to the semantics. DuckDB need not materialize it: result chunks can stream downstream while the engine retains the current frontier and, for regular `UNION`, the hash state required for duplicate elimination. It accumulates a complete copy only when the recursive term accesses `recurring.<cte_name>`{:.language-sql .highlight}.

For a monotone, linear recursive term, this frontier discipline is the operational core of **semi-naive evaluation**: an epoch reads only the preceding delta instead of reevaluating the recursive term from every row seen so far. The original operator already preserved this semantic invariant; the remaining mismatch concerned the lifetime of the physical machinery evaluating that frontier.

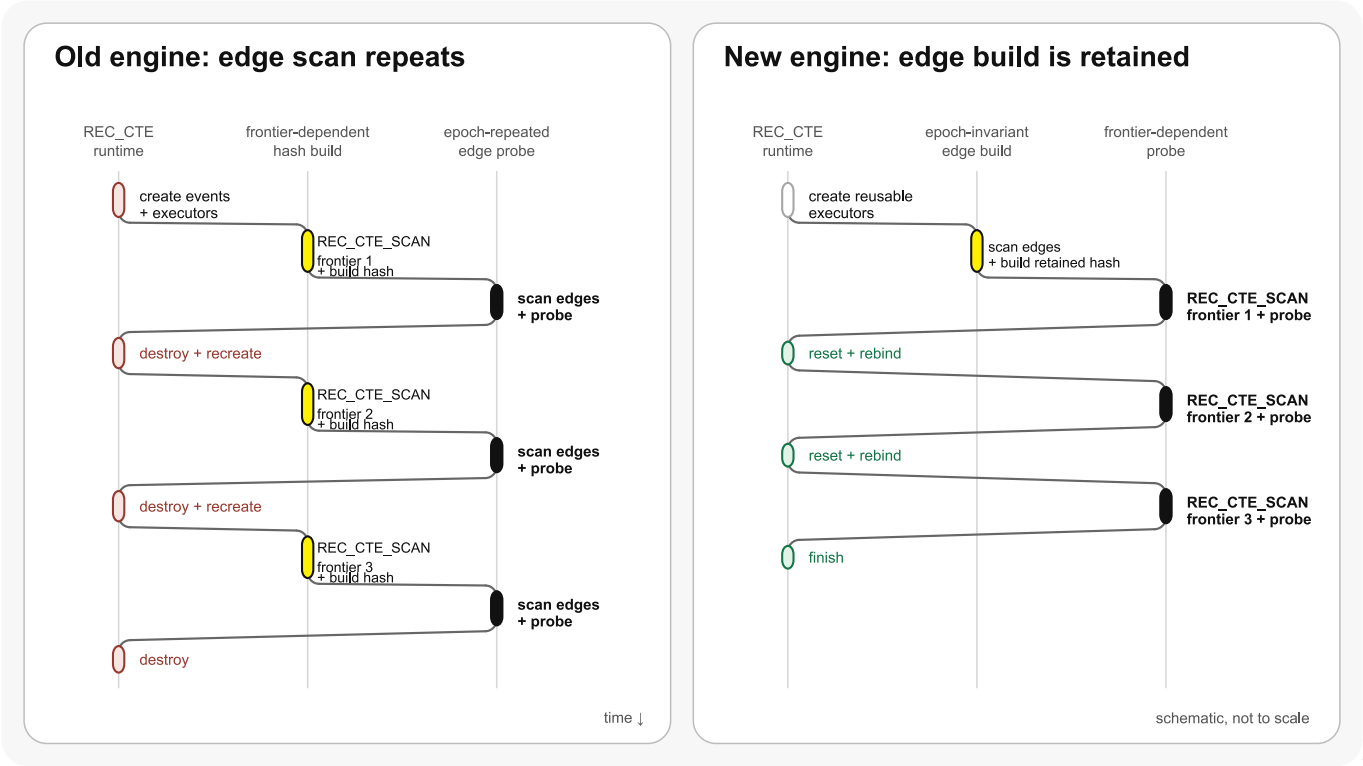
Epoch-Invariant State Belongs to the Invocation

The frontier scan, shown as `REC_CTE_SCAN` in an `EXPLAIN` plan, must read new input after an epoch boundary, and the runtime state of every operator that depends on that input must be reset. The query-plan-owned schedule projections remain valid. Executors checked out for the invocation and invocation-owned buffers can be reset for reuse, while a build exclusively over recursion-independent base tables can retain its materialized state if repeating the build would produce the same observable result. The old runtime failed to distinguish invocation-scoped materialized state from frontier-dependent state: materialized state whose validity spans the recursive invocation cannot be owned by one epoch.

An operation is **epoch-invariant** when its inputs do not depend on the current frontier and reevaluating it would reproduce the same observable result. Scanning a stable base table and building a hash table from it is the canonical example. The recursive runtime follows the same ownership distinction as **loop-invariant code motion**: the invariant build belongs outside the epoch loop, while probing it with the changing frontier remains inside.

The ownership invariant first failed in the recursive runtime. DuckDB executes a query as a graph of pipelines. Although the old physical operator retained its recursive meta-pipeline, every epoch recreated events and executors, scheduled the pipelines and reconstructed their state. Epoch-invariant pipelines consequently reread the same static input and rebuilt the same state.

A large epoch can amortize fixed scheduling and setup costs across many frontier rows. Rebuilding invariant state remains proportional to the static input size and can cost far more than probing it with the changing frontier.



{: .lightmode-img }

DuckDB v1.5.5 rebuilds a hash table from each frontier and revisits `edges` as the probe input. The new engine reverses the join, builds the edge hash table once and probes it with each frontier.{: .caption }

The resulting ownership boundary is:

State	Owning scope	Epoch-boundary action
Physical operator tree and immutable schedule projections	Query plan	Retain
Reusable pipeline-executor pool	Query plan	Retain
Executor checkout, chunks and collection capacity	Recursive invocation	Reset contents for reuse
Accumulated duplicate-elimination or keyed state	Recursive invocation	Retain
Repeatable, recursion-independent builds	Recursive invocation	Retain materialized state
Volatile, side-effecting or unknown builds	Epoch	Rebuild
Recursive scans	Epoch	Rebind to current recursive state
Candidate output	Epoch	Clear or combine

The hash join in the opening query requires both a join reorientation and retained state. DuckDB v1.5.5 builds a hash table from the current `reachable` frontier and scans `edges` as the probe input in every epoch. The new planner places the recursion-independent `edges` relation on the build side where valid. The first epoch scans `edges` and builds the hash table; later epochs rebind `reachable` as the probe and reuse that table.

`EXPLAIN (ANALYZE, FORMAT JSON)` makes the avoided input work concrete. Both versions return the expected 20,000 nodes. DuckDB v1.5.5 reports `operator_rows_scanned: 19,718,328,320` on the `edges` scan, equivalent in row volume to about 19,718 complete scans, while the DuckDB v2.0 preview reports the table's one million rows once. Dynamic filtering skips parts of some v1.5.5 edge scans, but it cannot prevent the base relation from being revisited in every epoch.

Retention Requires a Repeatability Proof

We retain a build only when its observable result is repeatable. Recursion independence establishes one requirement. Repeatability adds another: a seeded sample may be retained; an unseeded sample, a volatile expression such as `nextval()`, DML, a side-effecting operator or an unknown extension operator must be rebuilt. The repeatability classifier owns that decision and rejects retention when it cannot prove safety. This conservative choice may forgo reuse while preserving query semantics.

Fully materialized CTE producers follow the same ownership rule: a recursion-independent result may survive while each consumer scan is reset. We do not retain streaming or hybrid producers because their consumption state does not have the same lifetime. Early source termination is another boundary case. A retained pipeline must not withhold partial output merely to fill a vector, because doing so could prevent a downstream `LIMIT` from stopping recursive execution early. Reuse therefore preserves that termination contract.

Exact Cardinalities Make Execution Adaptive

The epoch boundary provides information unavailable during ordinary query planning. By the time an epoch ends, DuckDB has produced the next frontier in full and therefore knows its row and chunk counts exactly. These observed cardinalities govern how the recursive runtime executes the next epoch; optimizer estimates no longer have to stand in for them.

Each Epoch Chooses Its Own Mode

Once the opening query has built and retained its edge hash table, an epoch carries one reachable node into one probe. Sending that work through a parallel scheduler would create more coordination than useful work. A traversal whose frontier expands to many chunks has the opposite shape: executing it on one thread would leave independent pipeline work idle. A static mode chosen for the complete query cannot serve both cases, and a single query may move between them as its frontier grows and contracts.

When the exact frontier and physical work shape do not justify task scheduling, an epoch runs **inline**. One thread walks the selected immutable schedule projection and drives its operators directly, without creating tasks or entering the general scheduler. When the work classifier finds enough independent work, **scheduled execution** instantiates the corresponding recursive event graph and distributes work across a bounded number of workers. Both modes use the same query-plan-owned schedule projections. Executors checked out for the invocation and invocation-owned buffers are reset for reuse; only the epoch-level execution policy changes.

Rows alone do not determine useful parallelism. The policy also considers frontier chunks, the number of recursive references, independent source tasks, configured threads and the pipelines that would actually execute. The same frontier can feed several pipelines, while an independent source can expose work not represented by frontier rows. Conversely, worker-local output and boundary combination introduce costs of their own. We therefore bound workers by both recursive input and physical work units, using private output only for broad regular `UNION ALL` and duplicate-eliminating `UNION` epochs where its combine cost can amortize.

Frozen Keyed State Enables Direct Probes

Torsten Grust and Björn Bamberg explain the original `USING KEY` feature and its applications in their "`USING KEY` in Recursive CTEs" blog post. Semantically, `USING KEY` operates the union table as keyed state: the declared key columns identify a row, while the remaining payload columns are maintained by declared aggregates or the default `last` aggregate. DuckDB physically represents this state in an aggregate hash table. The recursive term can therefore probe selected keys directly instead of scanning the complete recurring state.

Each Epoch Reads a Frozen Keyed State

Direct access must preserve one semantic invariant: every access through `recurring.<cte_name>`{:.language-sql .highlight} during epoch `i` observes the same keyed state S⊠. The epoch has three ordered phases: read S⊠, buffer its candidate bag and commit those candidates to obtain S⊠⊠$_1$. Making a candidate visible during the read phase would let another candidate from the same epoch observe it and make the result depend on worker arrival order. The three phases preserve an epoch-at-a-time state transition while probes execute concurrently.

The original implementation preserved the invariant by materializing keyed state into a collection for recurring-state reads. That representation gave every access a broad physical path: an epoch touching a few keys could still copy or scan a state containing millions of keys.

The keyed-state owner now enforces those phases. While recursive pipelines execute, `recurring.<cte_name>{:.language-sql .highlight}` reads the frozen aggregate hash table and candidates accumulate separately. The owner commits them only at the epoch boundary. A probe cannot observe a half-applied update, and hash-table growth cannot invalidate an address held by a concurrent reader. After recursion ends, the source drains the final keyed state once.

Probe Eligibility Follows Join Shape

The frozen representation supports specialized physical lookups while preserving the semantic view. An inner join that compares every declared key with = or `IS NOT DISTINCT FROM` can use a **recursive key join**, shown as `RECURSIVE_KEY_JOIN` in `EXPLAIN`, and probe the aggregate hash table directly. Such comparisons on a proper subset of a composite key can use an epoch-stable secondary hash index through `RECURSIVE_PARTIAL_KEY_JOIN`. New complete keys extend an index only after their addresses are stable; payload-only updates do not require index maintenance.

We select the complete-key specialization only for an inner join against a direct recurring-state scan (`REC_REC_CTE_SCAN`) with a direct, exactly typed scalar = or `IS NOT DISTINCT FROM` comparison for every declared key. Residual predicates, wrapped scans, mismatched types, nested keys and other join types use the general path. An ordinary recursive reference is also ineligible because it denotes the frontier; the accumulated keyed state is available through the recurring reference. Treating those two identities as interchangeable would change the query.

Sparse Work Should Avoid Full-State Scans

The workload below retains one million keys and advances 1,000 of them through 20 epochs:

```
WITH RECURSIVE state(key, value) USING KEY (key) AS (
    SELECT key, CASE WHEN key < 1_000 THEN 0 ELSE 100 END
    FROM range(1_000_000) keys(key)
    UNION ALL
    SELECT frontier.key, recurring_state.value + 1
    FROM state AS frontier
    JOIN recurring.state AS recurring_state USING (key)
    WHERE frontier.value < 20
)
SELECT count(*) AS keys, sum(value)::BIGINT AS value_sum
FROM state;
```

Before direct probes, the join repeatedly scanned `recurring.state`. The runtime metrics counted roughly 20 million full-state scan rows for 20,000 useful matches. The specialized plan replaces them with 20,000 direct probes, a 1,000× reduction in recurring-state rows examined. In a comparison of builds immediately before and after the direct-probe work, median runtime fell from 0.401 to 0.040 seconds. The new path also removes the per-epoch collection previously used to materialize the complete keyed state.

Candidates Can Leave Keyed State Unchanged

The candidate bag can be nonempty even when keyed state does not change. If candidates automatically become the next frontier, recursion may continue after the state visible to the query has converged. Retaining execution state and probing it efficiently cannot repair that semantic mismatch.

`USING KEY ... UNION` now gives this distinction SQL-level semantics. Both our "A Fix for the Fixation on Fixpoints" CIDR paper and "How DuckDB is USING KEY to Unlock Recursive Query Performance" SIGMOD paper describe the original candidate-frontier design, in which the candidate bag becomes the next working table regardless of whether keyed state changes. Neither defines a changed-key delta. To our knowledge, DuckDB is the first and currently the only database system to distinguish changed-key recursion under `UNION` from candidate-frontier recursion under `UNION ALL`.

Finalized State Belongs to the Commit Layer

Before this change, DuckDB followed the original design and treated `USING KEY ... UNION` and `USING KEY ... UNION ALL` alike. Every candidate produced by the recursive term became input to the next epoch, even if applying it left the recurring table unchanged. Because plain `UNION` had the same candidate-frontier behavior as `UNION ALL`, DuckDB v1.5 deprecated it for `USING KEY`. The changed-key semantics introduced here give the two keywords distinct meanings, so the upcoming v2.0 retains both. The observable aggregate result for a key is determined only after every candidate for the epoch has been applied. Consequently, the epoch-commit layer owns the decision whether a key changed.

A shortest-path epoch demonstrates why candidates cannot decide this themselves. Several routes may reach the same node, and every route must participate in `min(distance)`. Only the finalized minimum is observable in keyed state. Forwarding every inferior route multiplies the work of the next epoch; forwarding an unchanged minimum can keep candidate production alive after state convergence.

Consider keyed state `{A:8, B:7}` with a `min` payload and candidates `[A:9, A:5, B:7]`. Applying the complete candidate bag produces `{A:5, B:7}`. The `A:9` candidate does not affect the finalized minimum, and `B:7` reproduces an existing value. `UNION ALL` nevertheless forwards all three candidates. `UNION` forwards only the finalized row `A:5` because it is the only observable state change.

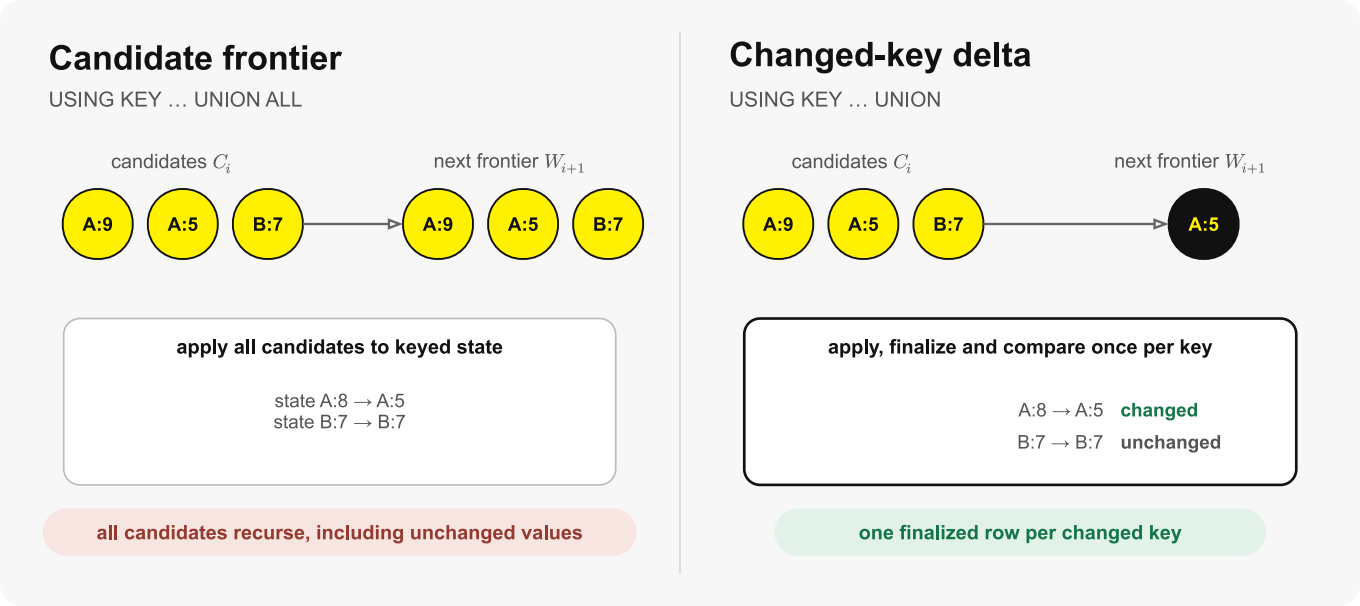
`UNION` and `UNION ALL` Expose Different Frontiers

Let C_i be the candidate bag produced in epoch i, S_i the keyed state visible during that epoch and W_{i+1} the next working table. Let `update(S_i, C_i)` apply all candidates in the bag and finalize every affected payload aggregate, and let `keys(S_i)` denote the keys present before the update. We define the two forms as follows:

```
USING KEY ... UNION ALL
S_{i+1} = update(S_i, C_i)
W_{i+1} = C_i

USING KEY ... UNION
S_{i+1} = update(S_i, C_i)
W_{i+1} = { S_{i+1}[k] | k ∈ keys(S_i) OR S_{i+1}[k] IS DISTINCT FROM S_i[k] }
```

For the same input state and candidate bag, both forms compute the same next keyed state. They differ in what becomes recursively visible: `UNION ALL` forwards the bag unchanged, while `UNION` forwards one finalized row for each new or observably changed key. Recursion under `USING KEY ... UNION` therefore stops when keyed state stops changing.

The epoch-commit layer implements the rule by recording prior key existence and snapshotting the pre-epoch finalized payload of an existing key when that key is first touched. It then applies every candidate before finalizing and comparing each touched key once. Duplicate candidates and losing `min` or `max` inputs still participate in aggregate evaluation, but they do not become recursive work on their own.



{: .lightmode-img }

For the same keyed state and candidate bag, both forms compute the same state update; the next frontier is either that bag or the changed-key delta.{: .caption }

We compare finalized values with SQL semantics, including `NULL` and collations. Key identity uses the same normalization as `GROUP BY`, and collated scalar keys remain eligible for direct and partial-key probes. The implementation also covers multi-column and nested payloads, `NaN` and signed zero.

The two frontiers also have different types. A `UNION ALL` frontier contains raw candidate rows, so its payload columns use aggregate input types. A `UNION` frontier contains finalized keyed rows, so its payload columns use aggregate result types. This distinction is visible through the SQL contract and constrains the internal representation. Because the keyword selects the frontier's contents, types and termination condition, we never infer one form from the other as an optimization.

Changed-Key Deltas Still Pay for Candidate Work

A changed-key delta reduces the next frontier only after all candidates have been applied. Millions of duplicate candidates can therefore still make the recurring hash-table update expensive even when few keys ultimately change. For eligible duplicate-heavy epochs, we first combine candidates in a temporary keyed hash table, then combine those aggregate states into recurring state.

Preaggregation Requires Combinable Aggregate State

Early combination must be observationally equivalent to applying candidates directly. We enable preaggregation only when every payload aggregate provides a state-combine operation and none is order-dependent. The default `last` aggregate and extension aggregates without a combine callback therefore remain on the direct update path. Without a combine operation, preaggregation lacks a valid state transition. For an order-dependent aggregate, it could change the observed input order and therefore change the result.

Cardinality Evidence Governs Preaggregation

Eligibility establishes correctness; observed cardinality determines whether the valid transformation is worthwhile. Epochs smaller than one standard vector bypass classification, as do non-expanding epochs whose candidate count does not exceed the preceding frontier. Larger eligible epochs build a HyperLogLog sketch over candidate keys. We preaggregate only when the sketch's error-inflated cardinality estimate is below one quarter of the candidate count, and stop sketching early once distinct-key evidence is sufficient to reject the additional hash table. The decision therefore adapts to the duplication observed in each epoch.

The cost of proving an unchanged value constrains this policy. For a wide workload with 102,400 unique existing-key updates, median runtime increases by 0.913 milliseconds, or 6%, under the new `UNION` semantics. The executor must compare finalized values where the previous implementation forwarded candidates without proving a change. Aggregate-specific shortcuts for `min` and `max` would place aggregate semantics in the recursive executor, so we do not use them. An aggregate-level change-reporting contract could put that responsibility in the aggregate interface. The current interface provides no such contract.

The large motivating workload was an LDBC SF100 pathfinding query. It generated about 21 million aggregate candidates, but epoch-level aggregation produced only 3.7 million observable keyed results. The adaptive path preaggregated 17 million candidates. On matched Release builds immediately before and after the changed-key work, query time fell from a median of 19.319 to 2.948 seconds, a 6.55× speedup, while peak resident memory fell from 3.918 to 2.663 GB.

The wider regression suite bounds that result. In the same build comparison, the geometric mean across 63 recursive benchmarks improved by 5.5%, and 60 were within ±2%. Duplicate fan-in and duplicate-heavy preaggregation improved substantially; the wide unique-key case above is the one stable loss. For this reason, we classify work per epoch instead of applying the strategy that wins the motivating query to every workload.

Recursive CTEs Now Execute as One Adaptive Computation

The new engine resolves the lifetime mismatch I wanted to remove after implementing DuckDB's original recursive operator. The query plan owns the physical operator tree, schedule projections and reusable executor pool. Within each invocation, the runtime checks

out executors and retains proven repeatable, recursion-independent state across epochs, while each epoch rebinds and resets frontier-dependent state. This separation makes retained invariant work and per-epoch execution policy compatible. Keyed recursion uses the epoch boundary to freeze recurring state for direct probes, then commits the candidates and exposes only observable changes as the next frontier under `UNION`. The implementation landed across #22211, #24031, #24565 and #24647.

That matters because recursion multiplies every physical cost placed inside the epoch loop. A scan, hash build or trip through the general scheduler that seems modest in isolation may run tens of thousands of times. The new engine pays eligible invariant costs once per invocation, avoids scheduler overhead when the observed work cannot amortize it and distributes epochs that expose sufficient independent work. Eligible keyed joins can access the required state directly, while `USING KEY ... UNION` recursion terminates when that state converges. This makes recursive SQL a more practical execution model for graph traversals, pathfinding and state machines, especially when the frontier remains small while the static input is large or when many candidates update the same keys.

The next step is to extend this foundation to more recursive plans. Future work can extend invocation-scoped reuse to more eligible operator states and use cardinalities observed at epoch boundaries for further execution decisions. Each extension must retain the same frontier and state-transition semantics. I intend to keep working along that boundary: reduce the physical cost of iteration while keeping recursive SQL predictable.

DuckDB Table Functions in Java

Publication date: 2026-08-25

Authors: Geertjan Wielenga, Alex Kasko

TL;DR: The DuckDB Java client can register table functions written in pure Java, exposing any Java-accessible data source as a SQL table. That turns DuckDB into a single-node query engine for heterogeneous joins across remote systems and local files, with no export step.

In a large organization, data is spread across many systems, including relational databases, document stores, message queues, data lakes, and cloud data warehouses. Some of these systems can be reached only through a vendor SDK, usually provided in Java, or a SOAP endpoint, and many sit behind custom authentication or single sign-on that is awkward to satisfy from anything but a JVM client. The layer that ties these together is often a JVM-based distributed query engine, such as Trino, that can run a single query joining across multiple data sources. The Java clients for those systems are mature and fast, with streaming APIs and, increasingly, virtual threads for asynchronous work. They are also already in production and cleared by corporate security.

Teams increasingly add DuckDB to these environments for fast single-node analytics. Because the surrounding infrastructure is Java, they use the DuckDB Java client (the JDBC driver). But a query engine is only as useful as the data it can reach, and in these environments much of that data is accessible only through Java client libraries.

Bringing External Data into DuckDB

Before writing a table function to reach a new source, it is worth asking what DuckDB can already read on its own, and the answer is almost everything. It reads Parquet and other files directly from data lakes. It connects to relational databases such as PostgreSQL, MySQL, and SQLite directly. The ODBC extension reaches proprietary databases such as Oracle, SQL Server, and DB2. JSON functions can query REST-like services by treating them as remote JSON files, and community extensions cover many more systems.

For a source that none of those cover, the data typically ends up in application code, processed by hand on top of a DuckDB result set. It is often more convenient, and frequently more performant, to bring that data into DuckDB's SQL directly, where it can be filtered and joined like any other table. *Table functions* are meant for exactly that. DuckDB has always let you plug in a custom source this way, but until now that meant writing the function in C++, which in practice was the only realistic choice.

Extending DuckDB in Pure Java

The DuckDB Java client can now register table functions written in pure Java. Any data source you can reach from Java can be exposed as a SQL table function, reusing the same streaming, asynchronous client libraries your infrastructure already runs and trusts. You can then query it alongside local Parquet and CSV files, wildcard globs included, joining and filtering across all of them in a single SQL statement.

Until now, adding a custom source in native code meant building and shipping a DuckDB extension. That involved a C++ toolchain, notoriously complex build systems, and the operational risk that a crash in native code is a segfault that takes down the whole JVM. A pure-Java table function is an ordinary Maven dependency that runs on the JVM alongside the rest of the application, with far less risk. And because it is just a Java API, any JVM language works, so the same function could be written in Kotlin, Scala, or Clojure.

This post walks through an example project that adds a `mongo_query()` function to DuckDB, and uses it to explain the mechanism.

> MongoDB is a document store with a robust Java API. The `mongo_query()` function below is a deliberately minimal, illustrative example, not a production connector. There are better ways to reach MongoDB in particular, including the ODBC extension and the MongoDB community extension. We use MongoDB as an example because it is a popular system with a good Java client. The same

> pattern applies to any source reachable from the JVM, whether through a JDBC driver, one of the most common Java clients, a SOAP service, or a proprietary vendor SDK.

The Goal

The function should be callable from ordinary SQL:

```
FROM mongo_query(
    'tab1',
    '{ "col1": "foo", "col2": { "$lt": 42 } }',
    columns = '["col1", "col2", "col3"]',
    hostname = 'localhost',
    port = 27017,
    database = 'db1'
);
```

The first argument is a MongoDB collection name. The second is a MongoDB filter document, in native MongoDB query syntax, passed through unchanged. The named `columns` parameter specifies which fields to project and the shape of the result. Because the filter is a MongoDB BSON document rather than a full SQL query, the collection name and the desired result columns cannot be read out of the query itself and are passed as separate parameters instead. Run against a seeded collection, the function returns the documents that match the filter:

col1 varchar	col2 int32	col3 varchar
foo	10	match-a
foo	41	match-b

There is no export to Parquet step and no intermediate table. The rows are read from a MongoDB cursor during query execution. And because the result is an ordinary relation, it can be joined with anything else DuckDB can read. Here it joins a remote collection against a directory of local CSV files in one statement:

```
SELECT
    c.region,
    count(*)                     AS orders,
    sum(o.amount::DECIMAL(10,2)) AS revenue
FROM mongo_query(
         'orders',
         '{ "status": "shipped" }',
         columns = '["customer_id", "amount"]',
         database = 'app'
     ) AS o
JOIN 'customers/*.csv' AS c
  ON c.customer_id = o.customer_id
GROUP BY c.region;
```

That is the heterogeneous join. A remote system reached through its Java driver is queried together with a glob of local files in one SQL statement on a single node.

Table Functions in Java

Adding a new data source no longer requires a native extension. Through the DuckDB Java client, a table function is registered entirely in Java using the `DuckDBFunctions.tableFunction()` builder. The builder declares the function name, its positional and named parameters and their types, and the implementation class. Registration in the example is done in `MongoExample.java`:

```
DuckDBFunctions.tableFunction()
    .withName("mongo_query")
    .withParameter(String.class)   // collection name
```

```
    .withParameter(String.class)   // Mongo filter (JSON)
    .withNamedParameter("columns", String.class)
    .withNamedParameter("hostname", String.class)
    .withNamedParameter("port", Integer.class)
    .withNamedParameter("database", String.class)
    .withNamedParameter("username", String.class)
    .withNamedParameter("password", String.class)
    .withFunction(new MongoQueryFunction())
    .register(connection);
```

This completes registration. `mongo_query(...)` is then available as a table function on that DuckDB connection and can be used in `FROM` clauses, joins, CTEs, and subqueries like any built-in function.

> This example takes `username` and `password` parameters and opens a fresh connection on every call. In a real, non-toy implementation it is often preferable to open a connection to the remote server once and run several queries on it. That can be done with a set of scalar functions. A `mongo_connect()` opens a connection and returns its handle as a DuckDB value, a `mongo_query()` reads through that handle, and a `mongo_close()` closes it. That approach is outside the scope of this post and could be the subject of a follow-up. For a similar example, see `odbc_connect` in the ODBC extension.

The implementation is a class that implements `DuckDBTableFunction`, which follows DuckDB's native table-function lifecycle with three callbacks:

- **`bind`** runs at prepare time. It reads the parameters, determines the output schema, and declares each result column with `addResultColumn()`.
- **`init`** runs once before execution and sets up shared state, such as opening a cursor or issuing the remote request.
- **`apply`** is called repeatedly, each time filling a chunk of the output, until it returns `0`.

> The API also exposes an `initLocal` callback for per-thread local state used in multithreaded execution. This post keeps to single-threaded execution and does not use it, though it may be covered in a future post.

The example implements these in `MongoQueryFunction.java`, with named parameters defined in `MongoQueryParameters.java`.

Bind: Declare the Output Schema

`bind` reads the call parameters and declares the output columns. Because MongoDB documents have no fixed schema, this example asks the caller to list the columns rather than inferring them from a prepared statement, and declares each one to DuckDB:

```
public MongoQueryBindData bind(DuckDBTableFunctionBindInfo info)
    throws Exception {
    String collectionName = info.getParameter(0).getString();
    String queryJson      = info.getParameter(1).getString();
    String columnsJson    = info.getNamedParameter("columns").getString();
    // ... read hostname / port / database / credentials ...

    MongoClient client = MongoClients.create(settings.build());
    MongoCollection<Document> collection =
        client.getDatabase(database).getCollection(collectionName);

    List<String> columns = new ArrayList<>();
    for (BsonValue bv : BsonArray.parse(columnsJson)) {
        String name = bv.asString().getValue();
        columns.add(name);
        info.addResultColumn(name, String.class); // declare it to DuckDB
    }

    Document query = Document.parse(queryJson);
    return new MongoQueryBindData(client, collection, columns, query);
}
```

Two aspects here generalize beyond MongoDB. DuckDB never parses or interprets the source's query language. The filter is handed to the source's own driver essentially unchanged, apart from a JSON-to-BSON conversion, so the full source syntax stays available. The object returned from `bind`, here a `MongoQueryBindData`, then carries that state forward to the next callback. Note that `bind` also runs when a query is `EXPLAIN`ed, since DuckDB needs the output schema to plan the query.

Init: Open the Cursor

`init` runs once and prepares the execution state that `apply` will consume. In the example it issues the query and stores the resulting cursor in a `MongoQueryInitData`. It also pins execution to a single thread with `setMaxThreads(1)`, because this post does not cover multithreaded execution:

```
public MongoQueryInitData init(DuckDBTableFunctionInitInfo info)
    throws Exception {
    info.setMaxThreads(1);
    MongoQueryBindData bindData = info.getBindData();
    FindIterable<Document> iter = bindData.collection.find(bindData.query);
    return new MongoQueryInitData(iter.cursor());
}
```

Apply: Stream Rows into Vectors

`apply` reflects DuckDB's vectorized execution model. Rather than returning rows, it writes them into the output data chunk column by column, up to the chunk's capacity of 2048 rows, and returns the number of rows produced:

```
public long apply(DuckDBTableFunctionCallInfo info,
    DuckDBDataChunkWriter output) throws Exception {
    MongoCursor<Document> cursor = info.getInitData().getResultCursor();

    long row = 0;
    for (; row < output.capacity() && cursor.hasNext(); row++) {
        Document doc = cursor.next();
        for (long col = 0; col < output.columnCount(); col++) {
            copyValueFromResultSetToVector(
                doc,
                output.vector(col),
                row,
                columns.get((int) col)
            );
        }
    }
    return row; // 0 signals "no more data"
}
```

Values are written into typed vectors through the vector API, using `setString`, `setInt`, `setDouble`, `setNull`, and related methods. DuckDB calls `apply` repeatedly, draining the source one vector-sized batch of 2048 rows at a time, so the full result does not have to be materialized in memory on either side. In the example, the BSON-to-vector conversion lives in `MongoTypes.java`.

> Writing rows into a `DuckDBDataChunkWriter` here is the mirror image of reading them out. The previous post covered consuming query results through the `DuckDBDataChunkReader` API, where the driver hands you chunks to read. As shown in the code snippet above, a table function uses the writer side of the same columnar model to produce them.

Implications

This example is a MongoDB connector, but the broader point is that DuckDB can now be extended in pure Java, putting the rich ecosystem of Java libraries within reach as a data source:

- **No native build.** The project contains no C++ code. It is a Maven project with two dependencies, the DuckDB Java client and the MongoDB driver. Adding a new source is ordinary Java work, done with the tools a team already uses and without touching a native toolchain.

- **Reuse of the official client.** The example does not reimplement a wire protocol or query language. It uses the vendor's Java driver and passes filters through unchanged. The same applies to any system with a JDBC driver or a Java SDK, such as a REST API, a message queue, or a SOAP service.
- **In-process access.** The data is not exported and reloaded. It is read from a cursor during query execution. Because the result is an ordinary table function, it can be joined with Parquet files, CSV globs, or attached databases in a single SQL statement, so DuckDB acts as the query layer over the source rather than a copy of it.
- **Predicates run at the source.** The filter is written in the source's own query language and handed to its driver unchanged, so it executes at the source, against its indexes, and only matching rows cross the wire. The selective work happens remotely, and DuckDB streams back only what it needs.

Current Limitations

These are limitations of the current Java table-function API:

- **A Java table function cannot yet be packaged as a DuckDB extension.** A DuckDB extension is a native shared library loaded into the DuckDB process, whereas a Java table function requires a JVM to run. As a result, the function can only be used from the DuckDB Java client. Packaging JVM code as a loadable extension is technically possible, for example through a JNI or FFM shim that embeds the JVM, or by compiling to a GraalVM native image, but neither is supported today.
- **Bind and init objects are not managed automatically.** In the current release, the caller is responsible for the lifecycle of the objects returned from `bind` and `init`, for example closing the MongoDB client and cursor after the query completes. A `DuckDBTableFunctionState` mechanism (contributed by a community member) manages that lifecycle for you. The init object then becomes:

```
public class MongoQueryInitData
    implements AutoCloseable /* DuckDBTableFunctionState */ {

    final MongoCursor<Document> cursor;

    // ...
    @Override
    public void close() {
        cursor.close();
    }
}
```

- **Composite DuckDB types are not yet supported.** The vector API currently covers scalar types, but `STRUCT`, `LIST`, and other nested types are planned for the future. Until then, nested source data has to be flattened or serialized to a scalar column.

Conclusion

For many analytical workloads, the data does not have to move to a cluster, and you do not need a distributed query engine to join across systems. With pure-Java table functions, the client libraries already running in your infrastructure expose remote data sources as SQL tables, and DuckDB joins them against each other and against local files in a single query on one node, using the Java client libraries you already have.

The table-function API is documented under Defining Functions. The complete example is available in the `duckdb_mongo_example` repository, and the DuckDB team is always happy to talk Java in the `#java` channel on the DuckDB Discord.

DuckLabs to Join AWS, Projects to Remain Open Source

Publication date: 2026-08-26

Authors: Mark Raasveldt, Hannes Mühleisen

TL;DR: DuckLabs will join Amazon Web Services (AWS), which is expected to be effective in early September. The projects will remain open-source under the MIT license.

DuckLabs today has announced that they will be joining Amazon Web Services as a new subsidiary. The big news for DuckDB and related projects like DuckLake is that there are no changes for our projects' roadmap, licensing, and governance model!

DuckDB, DuckLake, Quack, and all the other extensions will remain free and open source software under the MIT license under the stewardship of the non-profit DuckDB Foundation. DuckDB and related projects will continue to be developed at a very high pace. The Foundation will set up a Technical Advisory Board, which can influence the direction of the projects. We are lifting the limitations of community support. Please consult the blog post on ducklabs.com and download the media kit for more information.

Try DuckDB v2.0-dev

Publication date: 2026-09-02

Author: The DuckDB team

TL;DR: DuckDB's development team in Amsterdam has started getting DuckDB v2.0 ready for release in October. If you like shiny new things, try out the alpha releases now and report anything that might not be working as expected!

> **Update.** DuckDB v2.0.0 is out now! For details, see the announcement blog post.

The State of DuckDB v2.0

A few weeks ago, we published a preview of DuckDB v2.0.

Today, we branched off `v2.0-cyanoptera` from the `main` development branch in the `duckdb/duckdb` repository. This marks a feature freeze – in the coming weeks, development work will focus on testing, bugfixing and iterating toward the best DuckDB release to date: v2.0 Cyanoptera, projected for the second half of October.

The DuckDB ecosystem has `duckdb/duckdb` at its center: it's a dependency for most other work, so stabilizing it unlocks work across the other repositories. There are two main directions: clients and extensions.

Clients

Several DuckDB clients are already available as alpha versions.

CLI

To install the command line client on Linux or macOS, run:

```
curl https://install.duckdb.org | DUCKDB_VERSION=alpha bash
```

```
~/.duckdb/cli/latest/duckdb -c "SELECT version() AS version;"
```

```
┌──────────────────┐
│     version      │
│     varchar      │
├──────────────────┤
│ v2.0.0-alpha39998 │
└──────────────────┘
```

On Windows, run the following PowerShell script:

```
$env:DUCKDB_VERSION = "alpha"; irm https://install.duckdb.org/install.ps1 | iex
```

Alternatively, download the tarball, extract it and run `duckdb.exe`:

- DuckDB v2.0-dev for Windows AMD64 (x86_64)
- DuckDB v2.0-dev for Windows AArch64 (arm64)

Python

To install the Python client, run:

```
pip install duckdb --pre --upgrade
```

```
python3 -c "import duckdb; print(duckdb.version())"
```

This prints the versions of both the Python client (currently 1.6-dev) and the underlying DuckDB library (2.0.0-dev):

```
1.6.0.dev379 (with duckdb 2.0.0-alpha39998)
```

More Clients

We'll gradually add other clients to this list. Each new client that moves to 2.0 improves coverage for the DuckDB project, given that different clients will often have very different test environments and setups.

Extensions

In DuckDB v2.0, the `quack` extension will move from version 0.x to 1.0. The new version will unlock higher throughput for queries (both server to client *and* client to server) and better compatibility.

Other core extensions, such as `httpfs`, `ducklake`, `iceberg` or `spatial`, are also available in the alpha stage – please report any issues that you might encounter.

Community extensions can already be tested by providing a `ref_next` SHA, which builds them against the `v2.0-cyanoptera` branch before the release date. If no `ref_next` is provided, the `ref` SHA will be used once v2.0.0 is finalized.

Can I Help?

Yes, you can! DuckDB alpha clients are explicitly **not** production-ready, but you can already install them and throw some SQL queries at them to see whether your existing workloads work. Most are expected to be fine, some are expected to complete visibly faster, and some might result in an error. For the next few weeks, we are particularly interested in the latter group: are there environments, situations or queries where the DuckDB v2.0 alpha releases fall short?

If you find an issue, please open a bug report with a reproducible example, either in `duckdb/duckdb` or in the repository of the relevant client or extension.

DuckDB's main strength is its community: given enough eyeballs, all bugs are shallow, and your bug reports translate to stability gains for every user.

DuckDB Skills for Claude Code

Publication date: 2026-09-16

Author: The DuckDB team

TL;DR: The duckdb-skills plugin gives Claude Code a growing number of skills that use the DuckDB CLI to read data files, run queries, convert formats, explore object storage, work with spatial data, search the documentation and recall earlier sessions.

More likely than not, you've been using AI tools such as Claude Code for day-to-day work. You may have noticed that when AI needs to look at a data file, it makes use of Python, writes a small script, runs it, and then reads the output. This works, although it is slow, and the agent guesses column names and types and doesn't really check them.

Since DuckDB can read pretty much any file directly, the DuckDB team has written a set of skills that can tell your AI to use the DuckDB CLI instead. Whether the data is a file on your machine, a file on the internet or an Iceberg table behind a login, to DuckDB it is still a table or a catalog that it can query and process efficiently.

This post describes what is in the plugin and how it works.

Installation

In Claude Code, install `duckdb-skills` from the official plugins:

```
/plugin install duckdb-skills@claude-plugins-official
```

The skills are then available as `/duckdb-skills:<skill-name>{:.language-sql .highlight}` in all subsequent sessions. You need the DuckDB CLI installed; if it is not found, the skills will offer to install it.

Alternatively, you can add our repository as a plugin marketplace and install the plugin from there:

```
/plugin marketplace add duckdb/duckdb-skills
/plugin install duckdb-skills@duckdb-skills
```

An Example Interaction

Here is what a short exchange looks like when you ask a question about a data file:

```
You: How many taxi trips in that Parquet file were longer than 10 miles?

Claude runs:
    SELECT count(*) FROM 'trips.parquet' WHERE distance > 10;

DuckDB:
    Binder Error: Referenced column "distance" not found
    Candidate bindings: "trip_distance"

Claude reads the error, checks the columns and retries:
    SELECT count(*) FROM 'trips.parquet' WHERE trip_distance > 10;

DuckDB:
```

count_star()
184362

```
Claude: 184,362 of the trips were longer than 10 miles.
```

What you see in the chat is a conversation in natural language between you and Claude. Underneath it runs a second conversation in SQL between Claude and DuckDB. People and Claude are both at home in natural language, and Claude and DuckDB are both at home in SQL, so each step uses the language that fits it. When a query fails, Claude reads the error, adjusts the SQL and tries again, as in the retry above.

The two tools cover different work. DuckDB gives exact answers to exact questions, and those exact answers give Claude a firm base to reason from and to turn back into a short, readable reply for you.

What's in the Plugin

Each skill wraps a common data task behind a `/duckdb-skills:<skill-name>`{:.language-sql .highlight} slash command, grouped here by what they do.

You do not have to type the slash command, though: each skill also has a description that tells Claude Code when it applies, so you can simply ask in plain language (for example, “convert this CSV to Parquet” or “how far is the nearest station?”) and the agent will pick the right skill on its own.

Reading, Querying and Converting Data

`attach-db` attaches a DuckDB database and records it in the session state, so the other skills can use it:

- “Attach my_analytics.duckdb and show me its tables.”

`query` runs SQL or a plain-English question against attached databases or a file:

- “Show me the first 10 rows of the sales table.”
- “What are the top 5 customers by revenue?”

`read-file` reads and profiles any data file (CSV, JSON, Parquet, Avro, Excel, spatial, SQLite, Jupyter) on local disk or in S3, GCS, Azure or HTTPS:

- “What columns does variants.parquet have?”
- “How many rows are in the CSV file at that URL?”

`convert-file` converts a file from one format to another:

- “Convert sales.csv to Parquet.”
- “Save data.json as an Excel file.”

Remote and Spatial Data

`s3-explore` lists and queries data on S3, R2, GCS, MinIO or any S3-compatible storage without downloading it:

- “What's in s3://my-bucket/?”
- “How many rows are in s3://my-bucket/data.parquet?”

`spatial` answers spatial questions: distances, nearest neighbors, spatial joins and geographic lookups, including free Overture Maps data:

- “What are the 5 closest cafes to this point?”
- “Which districts in districts.geojson overlap?”

Documentation and Session Context

`duckdb-docs` searches the DuckDB and DuckLake documentation and blog posts:

- “How do window functions work in DuckDB?”
- “How do I read a CSV with custom delimiters?”

`read-memories` searches past Claude Code session logs for earlier decisions, conventions and open TODOs:

- “What did we decide about the DuckDB schema in earlier sessions?”

Setup

`install-duckdb` installs or updates extensions, including community extensions:

- “Install the spatial and httpfs extensions.”
- “Update my DuckDB extensions.”

The skills share a single per-project `state.sql` file, a plain SQL script of ATTACH, USE and LOAD statements, secrets and macros, so a session can be restored with `duckdb -init state.sql`.

For session-state details and local development instructions, see the README.

Conclusion

The duckdb-skills plugin is available at github.com/duckdb/duckdb-skills. If you run into problems or have suggestions, please open an issue. For DuckDB-related errors, it helps to include the output of `duckdb --version` and the full error message.

Persistent Databases in the Browser with DuckDB-Wasm and OPFS

Publication date: 2026-09-18

Authors: Carlo Piovesan, Geertjan Wielenga

TL;DR: DuckDB-Wasm can open a persistent database file in the browser's Origin Private File System (OPFS). This post shows how, and when data reaches disk.

When DuckDB-Wasm was launched in 2021, databases could not be persisted: everything lived in the Wasm heap and vanished when the tab closed. Keeping data meant serializing tables to Parquet, storing the bytes in IndexedDB, and re-registering them on the next page load. This was doable, but had to be handled at the application layer and was not offered out of the box by DuckDB-Wasm.

Modern browsers (since March 2023) now ship the Origin Private File System (OPFS), a per-origin, sandboxed file system with random-access reads and writes. DuckDB-Wasm (tested with versions 1.32.0 and 1.33.1-dev64.0) can use it as a storage backend, as described in the DuckDB documentation: a database opened at an `opfs://` path survives reloads and browser restarts.

The following call opens a database file in OPFS:

```
await db.open({
    path: 'opfs://analytics.duckdb',
    accessMode: duckdb.DuckDBAccessMode.READ_WRITE,
});
```

The result is a regular `.duckdb` file with a write-ahead log and checkpoints that survives page reloads and browser restarts.

> At the time of writing, the build that npm serves as `latest` (1.33.1-dev57.0) creates the OPFS files but never writes to them, so nothing persists. It canonicalizes the path to `opfs:/analytics.duckdb` with a single slash, which no longer matches the OPFS handle. Pin 1.32.0 or use the `next` tag (1.33.1-dev64.0 or later).

Opening a Database

The setup is the same as for any DuckDB-Wasm application: pick a bundle, start a worker, instantiate the database. The only new part is the open call, marked below. The import resolves to whichever version is installed, and `getJsDelivrBundles()` fetches the matching worker and `.wasm` files, so install a version that persists correctly: `npm install @duckdb/duckdb-wasm@1.32.0` or `@next`.

```
import * as duckdb from '@duckdb/duckdb-wasm';

const bundles = duckdb.getJsDelivrBundles();
const bundle = await duckdb.selectBundle(bundles);

// Worker scripts must be same-origin, so wrap the CDN worker URL in a Blob
const workerUrl = URL.createObjectURL(
    new Blob([`importScripts("${bundle.mainWorker}");`], {
        type: 'text/javascript'
    })
);
const worker = new Worker(workerUrl);
const db = new duckdb.AsyncDuckDB(new duckdb.ConsoleLogger(), worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
URL.revokeObjectURL(workerUrl);

// NEW: open a persistent database in OPFS instead of the default :memory:
await db.open({
    path: 'opfs://analytics.duckdb',
```

```
    accessMode: duckdb.DuckDBAccessMode.READ_WRITE,
});

const conn = await db.connect();
await conn.query(`
    CREATE TABLE IF NOT EXISTS transactions (
        id BIGINT,
        ts TIMESTAMP,
        merchant VARCHAR,
        category VARCHAR,
        amount DECIMAL(10, 2)
    );
`);

await conn.query(` INSERT INTO transactions VALUES (1, now(), 'Coolblue', 'electronics', 49.95)`);
await conn.query('CHECKPOINT');

const result = await conn.query('SELECT count(*) AS n FROM transactions');
console.log(result.toArray()[0].n);
```

Reload the page and run the same code. The `CREATE TABLE IF NOT EXISTS` statement finds the existing table and does nothing, the insert adds a second row, and the count prints 2. There is no sync step, no export, no `localStorage` key to remember. The `opfs://` prefix tells DuckDB-Wasm's file system layer to resolve the path against the origin's private file system instead of the in-memory Emscripten file system.

Opening the database creates the database file and its `.wal` in OPFS. Builds from 1.33.1-dev64.0 onward also create two empty helper files, `.wal.checkpoint` and `.wal.recovery`, that DuckDB uses during checkpointing. The `.duckdb` file is a regular DuckDB database file. If you pull it out of OPFS (shown below) and open it with the CLI or the Python client, it works.

Data Files

The same prefix works for data files. A common pattern is to load a remote dataset once, keep it in the persistent database, and cache derived results as Parquet files in OPFS. The example below uses the TPC-H `orders` table (scale factor 0.01, about 1,500 rows) that the DuckDB web shell serves:

```
await conn.query(`
    CREATE TABLE IF NOT EXISTS orders AS
    SELECT * FROM 'https://shell.duckdb.org/data/tpch/0_01/parquet/orders.parquet';
`);
await conn.query('CHECKPOINT');
```

DuckDB-Wasm reads the remote file with HTTP range requests. Because the table is created with `IF NOT EXISTS`, the file is fetched only on the first page load; on later loads the table comes from OPFS and no request goes to `shell.duckdb.org`. You can see this in the browser's Network tab, which lists the range requests on the first load and stays quiet afterwards, or in DuckDB-Wasm's own logs: the `ConsoleLogger` passed to `AsyncDuckDB` records each HTTP read, so the absence of those log lines on a reload confirms the data is served entirely from OPFS.

With the data local, an aggregation can be written to a Parquet file in OPFS and read back later:

```
COPY (
    SELECT o_orderpriority AS priority,
           date_trunc('month', o_orderdate) AS month,
           sum(o_totalprice) AS total
    FROM orders
    GROUP BY ALL
) TO 'opfs://cache/monthly_totals.parquet';

SELECT * FROM 'opfs://cache/monthly_totals.parquet';
```

Nested directories such as `cache/` are created on demand. OPFS files are ordinary DuckDB file paths, so globbing, `read_csv` and the other readers work as usual. Reading and writing `opfs://` paths from SQL needs one extra option on `open()`, described next.

File Handling Modes

With `opfs: { fileHandling: 'auto' }`, DuckDB-Wasm scans each statement for single-quoted `'opfs://...'` literals, registers those files before execution (creating them and any missing directories if needed) and drops the handles afterwards. The option only takes effect when the database itself was opened from an `opfs://` path. Without it, every file other than the database has to be registered by hand:

```
// Option 1: automatic registration of opfs:// paths found in SQL
await db.open({
    path: 'opfs://analytics.duckdb',
    accessMode: duckdb.DuckDBAccessMode.READ_WRITE,
    opfs: { fileHandling: 'auto' },
});

// Option 2: manual registration (the default)
await db.open({
    path: 'opfs://analytics.duckdb',
    accessMode: duckdb.DuckDBAccessMode.READ_WRITE,
});
await db.registerOPFSFileName('opfs://cache/monthly_totals.parquet');
// ... run queries against it ...
await db.dropFile('opfs://cache/monthly_totals.parquet');
```

Automatic mode is convenient for one-off reads. Manual mode requires more code but avoids re-acquiring an OPFS access handle on every statement, which adds up for applications that run many small queries. A file can be held by only one handle at a time, so the DuckDB documentation recommends dropping registered files with `db.dropFile()` before another connection or database instance opens them.

Durability

DuckDB-Wasm writes to OPFS the same way native DuckDB writes to a local disk: through a write-ahead log and periodic checkpoints. What differs is that a browser tab is rarely closed cleanly, so the defaults that work on a desktop can leave you with a slow reopen.

DuckDB uses a write-ahead log. Committed transactions are appended to `analytics.duckdb.wal` first. The main file is updated at *checkpoint* time. A checkpoint happens automatically when the WAL grows past `checkpoint_threshold` (16 MB by default), when the database is closed cleanly, or when you run `CHECKPOINT` yourself.

In a desktop process, "closed cleanly" is the common case. In a browser tab, it is not: the user closes the tab, the phone kills the background page, the laptop lid goes down. None of these run your shutdown code reliably. Two rules follow from that.

Call `CHECKPOINT` after writes you cannot afford to lose. The DuckDB documentation is explicit about this: writes are flushed to OPFS by `CHECKPOINT`. Committed transactions are appended to the WAL, and DuckDB replays the WAL on the next open, but a browser tab can be terminated at any point, so a checkpoint is the only way to be certain that the data is in the main file.

Checkpoint per batch, not per statement. A large WAL also makes the *next* open slower, because replay has to happen before the first query. For an interactive app, checkpointing after each batch of user edits keeps both the data safe and the reopen fast:

```
await conn.query('INSERT INTO transactions VALUES (...)');
await conn.query('CHECKPOINT');
```

If you would rather not track batches, set the checkpoint threshold to zero once after connecting. DuckDB then checkpoints after every statement, which costs some write throughput but removes the question entirely:

```
await conn.query(` SET checkpoint_threshold = '0KB'`{:.language-sql .highlight});
```

A clean shutdown looks like this:

```
await conn.query('CHECKPOINT');
await conn.close();
await db.terminate();
```

What happens when a tab is killed mid-transaction, and how to share one database between tabs, are covered in a follow-up post.

There is a second kind of durability to keep in mind, one that sits below DuckDB. OPFS is browser storage, not a hard guarantee. The browser can evict it when disk space runs low or when the origin has not been visited for a long time, and the user can clear it from the site's settings. Treat OPFS as a fast local cache for accelerating startup and persisting working state, not as your only copy of data you cannot lose. For durable storage, keep the source of truth somewhere stable and sync back to it: a DuckLake catalog, or plain files on object storage through `s3://` paths.

Export

Users will want to move their data to another device, back it up, or open it with a different tool. DuckDB-Wasm itself cannot move files into or out of OPFS yet, but the database is a plain DuckDB file and the browser's OPFS API lets you read it back as bytes:

```
await conn.query('CHECKPOINT');
const root = await navigator.storage.getDirectory();
const handle = await root.getFileHandle('analytics.duckdb');
const file = await handle.getFile();
// Offer as a download, upload to your backend, etc.
const url = URL.createObjectURL(file);
```

Or export from SQL to Parquet:

```
COPY transactions
TO 'opfs://export/transactions.parquet'
(FORMAT parquet, COMPRESSION zstd);
```

Combined with DuckDB's Parquet support, this allows preparing and cleaning data in the browser before uploading it to a server. And because the on-disk format is standard, the reverse works too: ship a pre-built `.duckdb` file with your app, copy it into OPFS on first launch, and open it. Users get a local dataset without an import step.

Conclusion

Lack of persistence was the main limitation of DuckDB-Wasm for a long time. With OPFS, DuckDB-Wasm can open a database file in the browser, commit transactions to a WAL, checkpoint, and reopen the same database after a reload. Three things make it work well: run `CHECKPOINT` after each batch of writes rather than after every statement, give users a way to download the database file, and read the limitations listed in the DuckDB documentation before shipping: one handle per file, and renames from SQL only work between two already-registered OPFS files.

With this, a local-first application no longer needs a server, IndexedDB wrapper, or custom serialization to keep analytical data between sessions. Try it in your own application, and share what you build on GitHub or Discord.

DuckDB Now Ships inside dbt v2

Publication date: 2026-09-22

Author: Geertjan Wielenga

TL;DR: dbt v2, which runs on the new Rust-based Fusion engine, is the first dbt release that ships with a built-in DuckDB adapter. This post covers setup, DuckLake and Iceberg catalogs, querying dbt's Parquet metadata with DuckDB, plus other v2 features that matter to DuckDB users, including migrating to dbt v2.

dbt is the tool many data teams use to manage their SQL transformations: you write each model as a `SELECT` statement, and dbt works out the order to run them in from the references between models, builds the resulting tables and views in your database, and can test them along the way.

dbt-duckdb, the dbt adapter for DuckDB, received its first pull request on August 27, 2021, and in the meantime has 1.4k stars on GitHub. Since then, dbt users have been able to install one Python package (dbt-duckdb, via `pip`), point it at a file (a local DuckDB database), and have a working project (models building into tables and views), without having to sign up to (and pay for) servers or warehouses.

When dbt Labs announced the new Rust-based Fusion engine in May 2025, DuckDB initially wasn't supported out of the box. That has changed with dbt v2, which ships with a DuckDB adapter built in. Here is how to set it up and what else is new.

Background

dbt Labs announced the new Rust-based Fusion engine on May 28, 2025. Two days later, a user, ran-codes, opened a GitHub issue asking for a DuckDB adapter:

> Quote There is a huge community utilizing the DuckDB adaptor to run DBT. For me personally, I was able to learn and start using DBT just because of the light-weight setup for the dbt-duckdb workflow and it has allowed me to get over the learning curve to start using DBT.
>
> -- ran-codes, on GitHub

At the time of this writing, the issue resulted in 146❤and 21 reactions. The adapter is now built into dbt v2.

On June 1, 2026, dbt Labs released the first alpha of dbt Core 2.0, built on the same foundations as Fusion, and open-sourced a large part of the Fusion code. That code moved into the dbt-core repository under Apache 2.0, and the dbt-fusion repository was archived. There are two distributions of v2, both free to install locally and both running on the same engine.

dbt 2.0.0 was released on September 14, 2026. That release also renamed the CLI branding from Fusion and dbt-core to dbt (proprietary) and dbt-oss (open source). So “Fusion” is now mostly the name of the engine, and the thing you install is just called dbt.

Setup

In dbt v1, an adapter was a standalone Python package. In v2, adapters live inside a Rust monorepo and connect through ADBC drivers.

With v2, dbt automatically downloads and caches the DuckDB driver the first time you run it, so after you install dbt there is nothing else to add. dbt also publishes a DuckDB quickstart guide for getting a project running locally.

A basic profile looks the same as before:

```
my_project:
  target: dev
  outputs:
```

```
    dev:
      type: duckdb
      path: ./warehouse.duckdb
```

DuckLake and Iceberg Catalogs

v2 adds catalog support that the Python adapter doesn't have. dbt's DuckDB docs flag it as "dbt v2 only"; the legacy Python adapter instead attached DuckLake through the profile's `attach` block.

With `catalogs.yml` you can configure DuckLake and Iceberg REST catalogs, with catalog-aware materializations. This requires the v2 engine with the `use_catalogs_v2` flag enabled and isn't available in the Python adapter. dbt generates and runs the ATTACH statements for you.

A DuckLake catalog is defined in `catalogs.yml`:

```
catalogs:
  - name: local_lake
    type: ducklake
    table_format: default
    config:
      duckdb:
        metadata_path: metadata.ducklake
        data_path: s3://my-bucket/lake
```

Enable the flag in `dbt_project.yml`, then reference the catalog from a model:

```
flags:
  use_catalogs_v2: true
```

```
{{ config(materialized = 'table', catalog_name = 'local_lake') }}
select * from {{ ref('customers') }}
```

The example above follows the DuckDB catalog support documentation.

dbt Metadata as Parquet

v2 also writes its metadata as Parquet as an alternative to the large JSON files, and these (as well as the large JSON files) can be queried directly with DuckDB.

dbt calls this the Information Schema, a v2 feature that stores the manifest as Parquet instead of JSON. Running `dbt parse --generate-info-schema` writes a set of Parquet files to `target/info_schema/v1/`, so you can list your models without parsing `manifest.json`.

These are the same artifacts dbt ships as test fixtures, so you can query one straight from the dbt repository using DuckDB without running dbt first:

```
SELECT name, materialized, schema_name
FROM 'https://raw.githubusercontent.com/dbt-labs/dbt/main/crates/dbt-docs-
server/web/src/test/fixtures/parquet/dbt.models.parquet';
```

For the above, this lists the three models in the fixture, along with how each is materialized and the schema it lands in:

name varchar	materialized varchar	schema_name varchar
my_second_model	view	main
my_third_model	view	main
my_first_model	view	main

Why would you do this? On a large project, the JSON `manifest.json` can grow to hundreds of megabytes, and reading it means loading and parsing the whole file just to answer a simple question. (Although, DuckDB can do this too.) The Parquet files are columnar, so DuckDB reads only the columns you select and can filter them without materializing everything in memory. That makes it practical to ask questions about the project itself: which models are materialized as tables rather than views, which schema each one lands in, or which models are missing tests.

This is useful in a CI check or an audit script, where you want to enforce conventions across a project without standing up dbt or the warehouse. Because the files are located on disk after a `dbt parse`, you can point DuckDB at them directly and treat your project's metadata as just another dataset to query.

SQL Comprehension and Column-Level Lineage

dbt models combine SQL with Jinja templating, which earlier versions compiled into a query string without inspecting the SQL itself. The v2 engine instead has a native understanding of SQL across multiple engine dialects. That means it can catch invalid column references and type mismatches before a query reaches the warehouse, rather than surfacing them only when the model runs against DuckDB.

That same analysis produces column-level lineage locally, without a dbt platform account. Running `dbt compile` with `--generate-info-schema --static-analysis strict` writes a `dbt.column_lineage` file into the Information Schema Parquet directory covered above, so you can trace which upstream columns feed each model with a plain DuckDB query.

Faster Local Development

v2 is distributed as a compiled Rust binary rather than a set of Python packages, so there is no Python dependency tree to resolve before a run. dbt describes the engine as the foundation for fast builds on large projects, where parsing and compiling happen inside that single native executable.

The dbt VS Code extension builds on the same SQL comprehension. As you edit models, it gives you autocomplete, hover information, and inline errors, so mistakes show up in the editor instead of after a round trip to the warehouse. The extension is published on the VS Code Marketplace.

Pinned DuckDB and Native Functions

v2 ships a pinned DuckDB version, rather than relying on whatever version pip resolves for the Python adapter. Pinning the version is what enables the read-write Iceberg REST catalog support described above, which depends on features from that specific DuckDB build.

Pinning a specific DuckDB version also lets dbt push work down into the database. Some adapter logic that used to be a SQL macro is now implemented as a native DuckDB extension function, such as `array_except`, which is exposed as `sf_array_except`.

Migrating

A low-risk first step is to test the v2 parser while still on dbt v1.12, which ships an opt-in v2 parser. dbt's docs describe this as a way to catch compatibility issues early before fully migrating. Run the following command to check whether your project parses:

```
dbt parse --use-v2-parser
```

If it does, follow the install guide to switch. The dbt-autofix package handles many of the required changes, and there is an upgrade guide for v2.

Conclusion

The DuckDB adapter is now part of dbt v2 and needs no separate install, and the Python versions of dbt Core remain available if you'd rather not move or not move yet. Either way, running dbt on DuckDB means you develop, test, and publish your models on your own machine.

Beyond removing the separate install, v2 is where DuckDB picks up several new capabilities: catalog support for DuckLake and Iceberg, metadata written as queryable Parquet, native SQL comprehension with column-level lineage, and a pinned DuckDB build.

If you've already been using dbt-duckdb, upgrading to v2 means one less package to install and all of the above to build on. And if you haven't, a single dbt install and a few lines of profile are enough to start building models directly on your laptop, without servers or warehouses.

DuckDB and Hugging Face: Querying Datasets Directly

Publication date: 2026-09-25

Author: The DuckDB team

TL;DR: Hugging Face hosts hundreds of thousands of datasets, and DuckDB can read them directly, exactly where they are and without downloading anything, over the DuckDB `hf://` protocol. This post looks at how the integration works and the scenarios where it works best.

Hugging Face is where much of the machine learning community publishes and finds its datasets, while DuckDB is the in-process analytical database that queries files like CSV and Parquet directly, with no server or warehouse to install or run.

Did you know that, since DuckDB v0.10.3 (released on May 22, 2024), you can point a `SELECT` at a dataset on the Hugging Face Hub, using the DuckDB `hf://` protocol, and query it, without downloading it first? This post covers how that integration works and the use cases it fits.

Background

On the Hugging Face Hub, each dataset is a git repository holding its data as plain files, usually CSV, JSONL, or Parquet. The `cais/mmlu` benchmark and the `datasets-examples/doc-formats-csv-1` repository used later in this post are two such examples: you can browse their files and commit history in the browser, the same way you would any git repository.

Before the Hugging Face integration, getting at that data from DuckDB meant downloading the files first, or loading them with the Hugging Face `datasets` library, before they could be read and analyzed. Either way, the data had to be copied out of the Hugging Face Hub before you could query it.

DuckDB was already able to read remote files over HTTP through its `httpfs` extension, so reading a URL directly was not new. Hugging Face datasets are also increasingly published as Parquet, the columnar format DuckDB reads natively and can scan without materializing everything in memory.

DuckDB and Hugging Face worked together to add the `hf://` path scheme on top of `httpfs`, announced in May 2024 with DuckDB v0.10.3. As a result, DuckDB can resolve a dataset repository to the files inside it, so that a query can read and analyze them exactly where they are located, instead of via a downloaded copy.

Reading Hugging Face Datasets Directly

The examples below cover the common cases. DuckDB's Hugging Face docs are the full reference. The scheme maps a Hugging Face dataset repository onto a path DuckDB can read:

```
hf://datasets/<my_username>/<my_dataset>/<path_to_file>
```

Reading a file is then just a query. This reads the CSV file from the `datasets-examples/doc-formats-csv-1` repository:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

kind	sound
dog	woof
cat	meow

kind	sound
pokemon	pika
human	hello

Here `datasets-examples` is the user or organization, `doc-formats-csv-1` is the dataset repository, and `data.csv` is the file inside it. The same example data is published in the `doc-formats-jsonl-1` and `doc-formats-parquet-1` repositories, so these queries return the same four rows:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-jsonl-1/data.jsonl';
```

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-parquet-1/data/train-00000-of-00001.parquet';
```

DuckDB infers the format from the file and reads only the columns that the query actually needs. And, nothing is downloaded to a local copy first.

Querying Many Files at Once

Datasets are often split across many files. A glob pattern lets you treat a whole directory as one table. The `cais/mmlu` benchmark stores its `astronomy` task across three Parquet files (`dev`, `test`, and `validation`), and this counts the rows across all of them:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet';
```

count
173

Because DuckDB reads Parquet column by column, you can filter across all those files without pulling every row into memory:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet'
WHERE question LIKE '%planet%';
```

count
21

Versions and the `~parquet` Branch

Each Hugging Face dataset is a git repository, so it has branches and revisions. You can pin a query to a specific one with an `@` suffix:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1@~parquet/**/*.parquet';
```

kind	sound
dog	woof
cat	meow
pokemon	pika

kind	sound
human	hello

Hugging Face automatically converts every dataset into Parquet on this special `~parquet` branch to make it efficient to scan. So even a dataset published as CSV or JSONL usually has a columnar version ready, which is what DuckDB reads fastest.

Saving a Local Copy

If you are going to query the same data repeatedly, materialize it once so you are not hitting the remote endpoint each time:

```
CREATE TABLE data AS
    SELECT *
    FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

After that, the data lives in the local table and queries no longer touch the Hugging Face Hub:

```
SELECT *
FROM data;
```

kind	sound
dog	woof
cat	meow
pokemon	pika
human	hello

Private and Gated Datasets

Public datasets need no setup. For private or gated ones, store a Hugging Face token in DuckDB's Secrets Manager. You can pass the token directly:

```
CREATE SECRET hf_token (
    TYPE huggingface,
    TOKEN 'your_hf_token'
);
```

Or let DuckDB pick it up from `~/.cache/huggingface/token`, where the Hugging Face tooling stores it:

```
CREATE SECRET hf_token (
    TYPE huggingface,
    PROVIDER credential_chain
);
```

Typical Use Cases

The integration is a good fit whenever you want to look at data on the Hugging Face Hub without committing to a download or a pipeline.

- **Exploring a dataset before you use it.** Before training or finetuning a dataset, you usually want to know what is in it: the row count, how many rows are unique, and what the columns look like. A single query against an `hf://` path answers that, reading only the columns you ask for:

```
SELECT
    count(*) AS questions,
    count(DISTINCT question) AS distinct_questions,
    avg(len(choices)) AS avg_choices
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet';
```

questions	distinct_questions	avg_choices
173	166	4.0

- **Filtering and sampling for training.** Large datasets often need to be narrowed to a subset, say a single language or the rows above some quality threshold, before they are useful. Express that as a `WHERE` clause and write the result straight to a local Parquet file with COPY:

  ```
  COPY (
      SELECT question, choices, answer
      FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet'
      WHERE question LIKE '%planet%'
  ) TO 'astronomy_planets.parquet';
  ```

  This turns a remote dataset into a focused local file, here the 21 astronomy questions that mention a planet.

- **Working with benchmarks and evaluation sets.** Benchmarks like MMLU ship as many small files grouped by task. You can read several tasks as one table and compute per-task statistics:

  ```
  SELECT subject, count(*) AS questions
  FROM read_parquet([
      'hf://datasets/cais/mmlu/astronomy/test-00000-of-00001.parquet',
      'hf://datasets/cais/mmlu/anatomy/test-00000-of-00001.parquet'
  ])
  GROUP BY subject
  ORDER BY subject;
  ```

subject	questions
anatomy	135
astronomy	152

- **Joining Hugging Face Hub data with your own.** Because an `hf://` path behaves like any other table source, you can join a public dataset against your own tables. Here a small lookup table maps each numeric answer to a choice letter:

  ```
  SELECT l.letter AS correct_choice, count(*) AS n
  FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet' AS m
  JOIN (VALUES (0, 'A'), (1, 'B'), (2, 'C'), (3, 'D')) AS l(idx, letter)
    ON m.answer = l.idx
  GROUP BY l.letter
  ORDER BY l.letter;
  ```

correct_choice	n
A	35
B	32
C	48
D	58

- **Reproducible analysis.** Pinning a query to a specific commit means it reads the same data every time it runs, which matters for anything you need to reproduce later. Add the revision with an `@` suffix:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu@c30699e8356da336a370243923dbaf21066bb9fe/astronomy/*.parquet';
```

count
173

Conclusion

The `hf://` protocol lets you query a dataset on the Hugging Face Hub by putting its path in a `SELECT`, with no download step and no server to run.

If you work with datasets on the Hugging Face Hub, that covers a lot of day-to-day tasks, from inspecting a new dataset to creating a training subset out of a larger one.

For further reading, see the original announcement post, DuckDB's Hugging Face docs, and Hugging Face's own DuckDB guide.

Announcing DuckDB 1.5.6

Publication date: 2026-09-28

Author: The DuckDB team

TL;DR: Today we are releasing DuckDB 1.5.6 with bugfixes and performance improvements.

In this blog post, we highlight a few important fixes in DuckDB v1.5.6, the sixth patch release in DuckDB's 1.5 (Variegata) line. The release ships bugfixes, performance improvements and security patches. You can find the full release notes on GitHub.

To install the new version, please visit the installation page.

Fixes

Here are the most important fixes from the DuckDB v1.5.6 release, organized by category:

Correctness

- #24240 – Fix `LIMIT` pushdown through a volatile projection with an `OFFSET`
- #24239 – Don't push filters on volatile groups through aggregates
- #24119 – Fix `UNNEST` pushdown
- #24399 – Preserve `NULL`s in Top-N window elimination when the `ORDER BY` expression has no column references
- #24551 – Fix Top-N window elimination for nullable ordering expressions
- #25831 – Fix wrong results from common subplan elimination with `UNION ALL` arms sharing a join subtree
- #25714 – Fix silent truncation of very long integer literals into `HUGEINT`
- #25766 – Fix `max` on Hive partition column after file pruning
- #24438 – Fix ICU `strptime` leaking time zone state between rows
- #24845 – Fix `GEOMETRY` row group pruning with `NULL`s and empty geometries
- #25728 – Fix reading and writing `TIME_NS` values in Parquet
- #26027 – Fix Parquet v2 value count mismatch for `NULL`s in a list that fills a page
- #26162 – Fix Parquet `VARIANT` shredding for `REQUIRED` fields and element groups

Crashes and Internal Errors

- #25103 – Fix crash in Top-N with `LIMIT 0`
- #24427 – Fix segfault in `url_decode` with `TRY()` on dictionary-encoded columns
- #24447 – Fix failed checkpoint marker recovery
- #25490 – Close the main WAL handle before renaming over it during WAL recovery

Generic Bugfixes

- #24065 – Automatically roll back failed implicitly wrapped multi-statements on all paths
- #25693 – Fix dead node counting in ART indexes
- #25573 – Report the real storage version when opening a DuckDB v2.0+ database file
- #25808 – Reject invalid UTF-8 produced by `printf`'s `%c` conversion

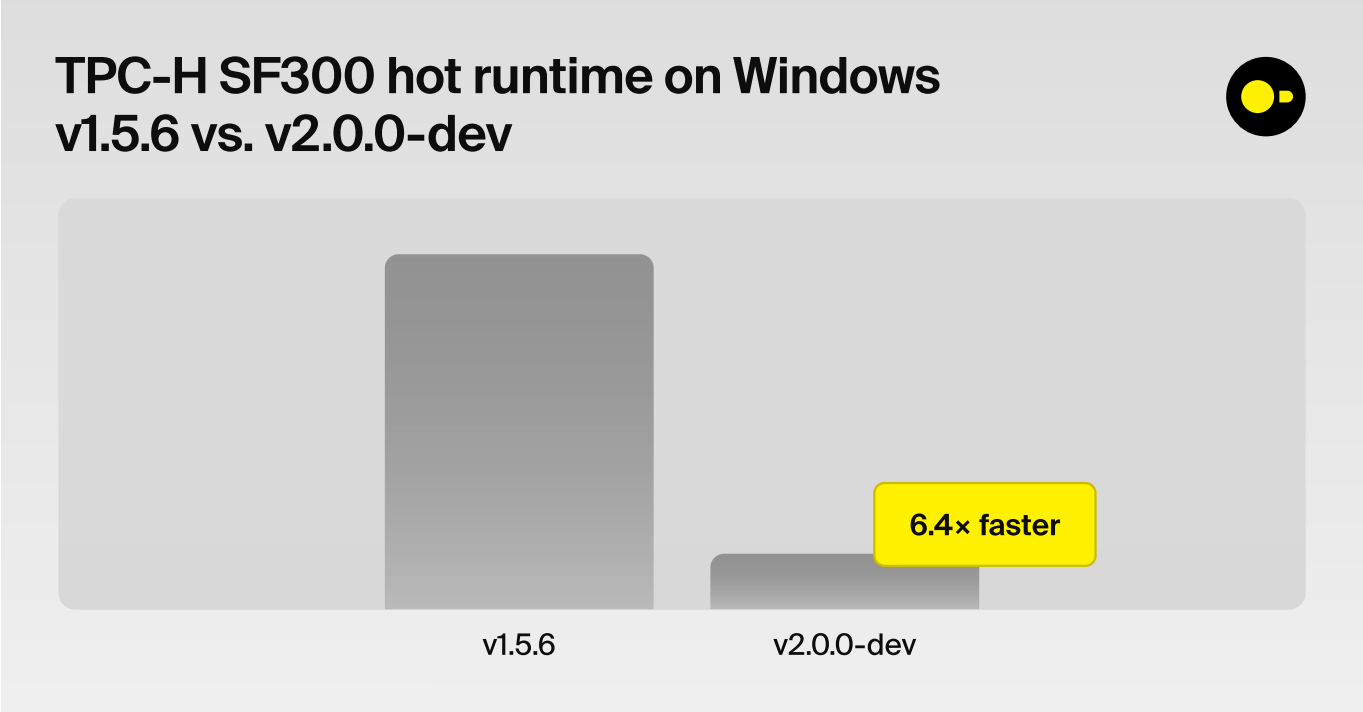
Miscellaneous

- #26102 – Add `enable_optimistic_write` setting
- #25283 – Harden temporary file reads
- #24362 – Unify C API symbol versioning for clients and extensions, stabilize all v1 APIs
- #25214 – Always quote identifiers in error messages

Looking Ahead: DuckDB v2.0 on Windows

If you have read this far, don't miss out on a sneak peek at DuckDB v2.0.0-dev for Windows. First, extensions are now available for these clients. Second, we recently ran a benchmark to measure the performance improvement that the new clients bring, and the results blew our minds!

We used Windows 11 25H2 on a laptop with 128 GB RAM and 12 AMD Ryzen AI 300 CPU cores with simultaneous multithreading (yielding 24 threads). We used the TPC-H SF300 dataset and ran each query twice on both DuckDB v1.5.6 and v2.0.0-dev (alpha43586). For each query, we took the runtime of the second (hot) run. The total runtime for DuckDB v1.5.6 was 822 seconds, while for v2.0.0-dev it was 129 seconds. That's more than 6× faster!



The detailed results are as follows:

Query	v1.5.6 (s)	v2.0.0-dev (s)
1	17.590	19.954
2	1.065	1.972
3	4.282	8.787
4	5.310	8.386
5	4.562	6.097
6	1.167	1.057
7	5.091	3.405
8	4.986	5.359

Query	v1.5.6 (s)	v2.0.0-dev (s)
9	16.662	17.505
10	60.919	7.619
11	6.797	0.674
12	3.350	2.579
13	174.847	13.256
14	20.044	2.001
15	22.836	1.807
16	43.019	1.953
17	4.958	2.805
18	230.175	11.453
19	23.013	2.416
20	24.303	2.391
21	67.344	5.766
22	79.726	1.533

This improvement is thanks to several optimizations, including a switch to the `clang-cl` compiler and a new allocator. That said, you can observe that there are some regressions for the time being (we are working on them!) and the 6× speedup may not generalize to all workloads – but rest assured that you should see significant improvements for most workloads on Windows once you upgrade to DuckDB v2.0.

Conclusion

This post was a short summary of the changes in v1.5.6. As usual, you can find the full release notes on GitHub. We would like to thank our contributors for providing detailed issue reports and patches. Stay tuned for future DuckDB releases, including v2.0.0 in October!

Jev and DuckDB: Plain-English Conditions in SQL

Publication date: 2026-09-29

Authors: Geertjan Wielenga, Gábor Szárnyas

TL;DR: TypeSafe AI released Jev, a model that returns typed answers instead of text, on September 15, 2026. Within ten days, several community extensions let you filter, classify and score DuckDB rows with plain-English conditions. This post covers what Jev is, how the extensions work and the posts and videos about them.

Say you have a Parquet file of 50,000 support tickets and want to know which customers are angry. A `LIKE` pattern won't find that. The usual options are to label data and train a classifier, or to send each row to a chat LLM and parse the text it returns. The first takes work up front. The second is slow and costs more per row.

On September 15, 2026, TypeSafe AI introduced Jev, a model that answers this kind of question with a typed value and a probability instead of a paragraph. Within a week, Hamilton Ulmer posted on X that he had built a DuckDB extension that classifies about 1,000 rows from any CSV, Parquet file or table in roughly ten seconds. Other DuckDB extensions appeared over the next few days.

This post describes what Jev is, how the extensions work and where to read more.

What Jev Is

Jev is what TypeSafe calls a *System One model*: you send it a state and a set of questions, and it returns typed answers with calibrated probabilities. It never generates a sentence. In the launch post, founder Diogo Almeida (who worked on the research behind ChatGPT at OpenAI) describes it as a function call backed by frontier intelligence. The name refers to Kahneman's fast System 1 thinking and to the economist William Stanley Jevons.

There are three question types, and each corresponds to something SQL already has:

Jev type	What it returns	SQL analogue
Noul	A yes/no judgment with a probability	A `BOOLEAN` predicate in `WHERE`
Choice	One option from a fixed list (up to 255)	An `ENUM` column you can `GROUP BY`
Score	A position on an ordered rubric	An ordinal you can `ORDER BY`

TypeSafe quotes 70–500 ms end-to-end latency and $0.042 per million input tokens, with output tokens free. All questions in a request are evaluated in parallel, so asking five questions about a row takes little more time than asking one. The launch post lists "map-reducing over big data" as a target use case, which is the kind of work people use DuckDB for.

For independent context, The New Stack's launch coverage covers the $40 million seed round. LangChain's post on building a harness with Jev shows where it fits inside an agent loop. Hyperstack's deep dive reproduces the scoring mechanism on an open model and checks TypeSafe's speed claims against LangChain's benchmark. Note that the headline 190× speedup figures come from TypeSafe's own workflow evals.

pg-jev and the DuckDB Ports

The first SQL integration was for Postgres. On September 17, Zachi posted pg-jev on X: a `jev()` function that turns a plain-English condition into a `WHERE` clause, with no index and no embeddings. His demo judged 129 rows in about a second for under a tenth of a cent, and you can try it on mock tables in the pg-jev live demo. The source is on GitHub. Four days later, Actian hired him.

DuckDB ports followed within days. The closest to the original is judoaseeta/duckdb-jev, which keeps pg-jev's function names.

Installation

With the exception of the `jev` community extension, most of the extensions are not yet distributed through the community extensions repository, so you build them yourself and load them as unsigned extensions. For the rest of the post, we'll use the `jev` community extension. To install and load it, run:

```
INSTALL jev FROM community;
LOAD jev;
```

> The `jev` community extension is supported in DuckDB v1.5.5 and v1.5.6.

Then set your TypeSafe API key, which you can get from the TypeSafe console after registering and topping up your account:

```
SET jev_api_key = '...';
```

The extension must be built for the exact DuckDB version and platform you run.

An Example

> **Warning.** All of the extensions in this post send row contents to a third-party API. Do not use them on data you are not allowed to share.

Let's start with filtering using the `person` table of the LDBC SF0.1 dataset.

```
-- Filter: a boolean predicate
CREATE TABLE person AS
FROM 'https://blobs.duckdb.org/data/ldbc-sf0.1-person.parquet';

SELECT id, firstName, lastName
FROM person
WHERE jev(person, 'the name is European')
LIMIT 5;
```

id int64	firstName varchar	lastName varchar
1129	Carmen	Lepland
10995116278700	Joseph	Anderson
28587302322727	Steve	Moore
30786325578904	Giuseppe	Donati
6597069766983	A. C.	Bos

In a ticketing system, you could rank, classify or score the tickets as follows:

```
-- Rank: a probability you can sort on
SELECT subject, jev_prob(tickets, 'the customer is angry') AS p
FROM tickets
ORDER BY p DESC
LIMIT 20;

-- Classify: one of a fixed set of labels
SELECT
    jev_choice(tickets, 'which team should handle this?',
               ['billing', 'technical', 'security', 'sales']) AS team,
    count(*)
FROM tickets
GROUP BY 1;

-- Score: a position on an ordered rubric
SELECT
    name,
```

```
    jev_score(products, 'how luxurious is this product?',
              ['budget', 'mid-range', 'premium', 'luxury']) AS luxury
FROM products
ORDER BY luxury DESC;
```

Because `jev()` returns a boolean, it works with the rest of SQL. If you add a cheaper condition such as `age > 40` in the same `WHERE` clause, it runs first, and rows it rejects are never sent to the API. A `LIMIT` stops the scan early, and `jev_max_rows_per_statement` caps the spend.

The video "I Tested Jev with DuckDB. It's 1.8x Faster Than Haiku." demonstrates plain-English SQL queries in DuckDB and compares Jev's speed with Claude Haiku. The 1.8× figure is from one test.

The DuckDB Extensions

Several people built these extensions independently, and they made different choices about batching, caching and output types.

Extension	Angle	Notable detail
judoaseeta/duckdb-jev	Port of pg-jev	Same `jev`, `jev_prob`, `jev_choice`, `jev_score` API, plus a process-wide cache keyed by row content
recodelabs/duck-jev	Natural-language `WHERE` clauses	Groups rows by question, judges duplicates once, runs batches across threads and shares a database-scoped cache
prasanthj/duckdb-jev	Throughput and production controls	Packs up to 1,000 judgments per request, streams across DuckDB chunks, reports usage through `jev_usage()`
Colliber's duckdb-jev	Typed results	Adds `jev_choice`, `jev_score`, `jev_noul` and `jev_ask`. A choice's options become an `ENUM` column type

DuckDB passes rows to an extension in chunks of up to 2,048, which suits batched API calls. The recodelabs extension groups each chunk by question and options, skips identical rows and sends the rest in batches across `jev_concurrency` threads. It retries 429s and 5xx errors with exponential backoff.

Batching makes a large difference to speed. The prasanthj extension reports a 0.211 s median for 100 nested JSON rows sent in batches of 25, compared with 16.8 s one row at a time. A 2,049-row run took under a second. The author notes these timings come from one local run on synthetic data. Larger batches may cost accuracy, though: pg-jev's docs report that accuracy drops above roughly 20–25 rows per request, so test batch sizes on your own data.

Colliber's version uses Jev's typed output directly. Choice options are defined in advance, so the result can be a DuckDB `ENUM` rather than text that needs parsing. The prasanthj build can evaluate nested JSON, `STRUCT`, `LIST` and `ARRAY` values without exporting them to Python.

Some replies under Hamilton Ulmer's post point out that a fine-tuned ModernBERT is still cheaper once training is amortized, and that treating classification as embedding retrieval can reach about 20,000 rows per second on CPU. Jev makes most sense when you have no labeled data and a new question to ask. For a fixed classification task at high volume, a trained model may be cheaper and faster.

Conclusion

Jev returns a `BOOLEAN`, an `ENUM` or an ordinal score, and DuckDB can filter, group and sort on all three. With these extensions, you can write a condition in English and use it alongside joins, `GROUP BY` and window functions.

Jev is in early access behind a waitlist, the extensions are days old and unsigned, and most performance numbers come from their own authors. Start on a small, non-sensitive sample, watch your usage counters and compare against a baseline before you trust the labels. If you build something with Jev and DuckDB, the extension authors welcome issues and pull requests on their repositories.

Further Reading

- Introducing System One Models & Jev (TypeSafe AI's launch post)
- TypeSafe launched Jev because sequential LLMs are “totally useless for computers” (The New Stack)
- What Is Jev? A Guide to TypeSafe AI's System One Model (LangChain blog)
- Jev: TypeSafe AI's First System One Decision Model (Hyperstack)
- I Tested Jev with DuckDB. It's 1.8x Faster Than Haiku. (YouTube)
- Calling Jev from SQL (YouTube)
- pg-jev launch post and live demo (Zachi)
- Hamilton Ulmer's DuckDB extension post (X)

Faster String Aggregations with Dimension Tables

Publication date: 2026-10-02

Author: The DuckDB team

TL;DR: When a query groups on long, repeated strings, move the strings into a small dimension table with sorted, narrow integer keys. Aggregate on the keys, then join the strings back in at the very end. The query works as before, though on small fixed-width integers instead of variable-length text.

Analytical workloads are full of repeated strings: product names, country names, station names, user agents, category labels. Take DuckDB's public train services dataset, which has one row for every stop a Dutch railway train makes. The data comes from the open datasets published by the Rijden de Treinen *(Are the trains running?)* application. You can query it straight from its URL:

```
SELECT departure_time, station_name, type
FROM 'https://blobs.duckdb.org/train_services.parquet'
LIMIT 5;
```

departure_time	station_name	type
2023-05-15 00:00:00	Rotterdam Centraal	Intercity
2023-05-15 00:13:00	Delft	Intercity
2023-05-15 00:29:00	Den Haag HS	Intercity
2023-05-15 00:45:00	Leiden Centraal	Intercity
2023-05-15 01:03:00	Schiphol Airport	Intercity

The table has 380,959 rows but only 537 distinct station names, so each name is stored again in thousands of rows: `Amsterdam Centraal` (18 bytes) alone appears in 7,591 of them. Load it into a table to follow along, in the web shell or the DuckDB CLI:

```
CREATE TABLE train_services AS
    FROM 'https://blobs.duckdb.org/train_services.parquet';
```

When you GROUP BY the station name, DuckDB has to process the full string for every row. For example, this query counts how many trains call at each station:

```
SELECT station_name, count(*) AS calls
FROM train_services
GROUP BY station_name;
```

Consider what happens for a handful of rows that call at just two stations:

Row	station_name	Work when grouping on `station_name`
1	Amsterdam Centraal	hash 18 bytes; new group, so copy 18 bytes into the hash table
2	Rotterdam Centraal	hash 18 bytes; new group, so copy 18 bytes into the hash table
3	Amsterdam Centraal	hash 18 bytes; existing group, so compare 18 bytes

Row	station_name	Work when grouping on `station_name`
4	Amsterdam Centraal	hash 18 bytes; existing group, so compare 18 bytes
5	Rotterdam Centraal	hash 18 bytes; existing group, so compare 18 bytes

The string work repeats for every row, even though there are only two distinct stations here, and only 537 across the whole table.

This post shows how to do that work on small integers instead, by giving each distinct string a number and looking up the strings only at the end.

Background

The approach is the star schema from data warehousing, used here for query performance. It came up when we looked at a user report of heavy memory use in a high-cardinality grouping.

That report turned out not to involve strings, but Richard Wesley pointed out that in his own work, building dimension tables and joining the strings back at the end made a large difference to string-heavy aggregations.

We have since added the pattern to the schema section of the Performance Guide.

Why Strings Are Expensive to Group On

DuckDB computes a GROUP BY with a hash aggregation, keeping one hash table entry per group, as described in the Parallel Grouped Aggregation in DuckDB blog post (2022). The aggregation uses the hash of each group key to find the key's slot in a hash table, then compares the key with the one stored in that slot. For integers, this takes a few CPU instructions. For strings, the cost grows with the length of the string.

In DuckDB, a string value is a 16-byte structure. Strings of up to 12 bytes are stored inline. Longer strings store a 4-byte prefix plus a pointer to the actual characters. That design keeps short strings cheap, but many real-world labels are longer than 12 bytes.

For these longer strings, hashing reads every byte of every string in every row. A match can't be confirmed from the prefix alone, so DuckDB follows the pointer and compares the full string. When a new group appears, its string is copied into the hash table's own memory, which makes the table larger than it would be with fixed-width keys.

Copying strings also affects memory use. A wider hash table fits less well into CPU caches, and in larger-than-memory aggregations it reaches the memory limit sooner and has to spill more data to disk. DuckDB does apply dictionary encoding to string columns on disk, as described in the Lightweight Compression in DuckDB blog post (2022), but that is a storage optimization: once a column is read into an aggregation, each group key is a full string again.

Integer keys avoid these costs, as their fixed width makes hashing and comparing them cheap. When the key range is small, DuckDB can avoid hashing altogether: if the statistics show that the keys fit in a small enough domain, the optimizer picks a perfect hash aggregate, controlled by the `perfect_ht_threshold` setting, which uses the key value directly as an index into an array.

Building a Dimension Table with Sorted, Narrow Keys

The examples build on the `train_services` table loaded above. Each row records one stop: a `service_id`, the `date`, the service `type`, the `train_number`, the `station_code` and `station_name`, and the `departure_time` and `arrival_time`. The repeated string we want to encode is `station_name`.

Step 1: Measure the Cardinality

First, find out how many distinct values the column has, since that count sets how narrow the key can be.

```
SELECT count(DISTINCT station_name) AS num_stations
FROM train_services;
```

This returns 537, and that count decides how wide the key needs to be. You want the narrowest integer type whose range still covers every distinct value, because a narrower key means fewer bytes per row in the fact table and a smaller entry in the hash table you group on.

The keys come from `row_number()`, which starts at 1 and only counts upward, so an unsigned type is the right fit, spending none of its range on negative values. `UTINYINT` (1 byte) holds up to 255 distinct values, `USMALLINT` (2 bytes) up to 65,535, and `UINTEGER` (4 bytes) up to about 4.3 billion.

The 537 station names do not fit in a `UTINYINT`, so `USMALLINT` is the narrowest that works, and that is the type the next step casts to. If the number of distinct values will grow, choose a larger type so you don't run out of keys.

Step 2: Build the Dimension Table

Next, assign each distinct string an integer key. The important detail is the `ORDER BY station_name` inside the window: the keys are assigned in string order.

```
CREATE OR REPLACE TABLE stations AS
    SELECT
        station_name,
        (row_number() OVER (ORDER BY station_name))::USMALLINT AS station_id
    FROM (SELECT DISTINCT station_name FROM train_services WHERE station_name IS NOT NULL);
```

Sorted keys have two advantages. First, `ORDER BY station_id` produces the same order as `ORDER BY station_name`, so you can sort on the cheap integer. Second, the key assignment is deterministic: rebuilding the table from the same data yields the same keys.

Step 3: Store the Key in the Fact Table

Finally, replace the string column in the fact table with its key, a rewrite of the table that you pay for once. The dimension table leaves out `NULL`, so the `LEFT JOIN` keeps the rows without a station and gives them a `NULL` key.

```
CREATE OR REPLACE TABLE train_services_encoded AS
    SELECT ts.* EXCLUDE (station_name), s.station_id
    FROM train_services ts
    LEFT JOIN stations s USING (station_name);
```

The `stations` dimension table holds each name once, keyed in alphabetical order:

station_id	station_name
1	's-Hertogenbosch
2	's-Hertogenbosch Oost
3	't Harde
4	Aachen Hbf

The fact table now stores the 2-byte `USMALLINT` key, so grouping on it is cheap:

Row	station_id	Work when grouping on `station_id`
1	28	hash 2 bytes; new group, so store 2 bytes

Row	station_id	Work when grouping on `station_id`
2	403	hash 2 bytes; new group, so store 2 bytes
3	28	hash 2 bytes; existing group, so compare 2 bytes
4	28	hash 2 bytes; existing group, so compare 2 bytes
5	403	hash 2 bytes; existing group, so compare 2 bytes

The keys follow string order: `Amsterdam Centraal` sorts before `Rotterdam Centraal`, so it gets the smaller key (28 versus 403). With a key range this small and dense, DuckDB can use a perfect hash aggregate, indexing directly by the key instead of hashing.

You can also leave out this step and join the dimension table on the fly inside each query. That still keeps the aggregation's hash table narrow, but every query then pays for hashing the strings once in the join. Storing the key in the fact table removes string processing from the query.

Querying the Encoded Table

With the key in place, the query aggregates on integers and only looks up the strings once the result is small. The `LEFT JOIN` keeps the group of rows without a station.

```
WITH rollup AS (
    SELECT
        station_id,
        date,
        count(*) AS calls
    FROM train_services_encoded
    GROUP BY ALL
)
SELECT s.station_name, rollup.* EXCLUDE (station_id)
FROM rollup
LEFT JOIN stations s USING (station_id)
ORDER BY station_id, date;
```

`GROUP BY ALL` groups by every selected column that is not aggregated, here `station_id` and `date`, so you don't repeat the list. The final join runs against the aggregated result, which has one row per group rather than one row per event. If the aggregation reduces a billion rows to a few hundred thousand, the join only looks up a few hundred thousand strings. Because the keys are sorted, `ORDER BY station_id` also sorts alphabetically by station name.

For top-N queries, apply the `LIMIT` before the join, so that only ten strings are looked up:

```
WITH top_stations AS (
    SELECT station_id, count(*) AS calls
    FROM train_services_encoded
    GROUP BY station_id
    ORDER BY calls DESC
    LIMIT 10
)
SELECT s.station_name, top_stations.calls
FROM top_stations
LEFT JOIN stations s USING (station_id)
ORDER BY calls DESC;
```

station_name	calls
Utrecht Centraal	7663
Amsterdam Centraal	7591
Zwolle	5013

station_name	calls
Schiphol Airport	4961
Amsterdam Sloterdijk	4854
…	…

To filter on the string, look up its key in the dimension table and filter the fact table on the integer.

```
SELECT count(*)
FROM train_services_encoded
WHERE station_id IN (SELECT station_id FROM stations WHERE station_name LIKE '%Centraal%');
```

Measuring the Effect

How much you gain depends on your data, mainly on how long the strings are and how many distinct values they have. On this 380,000-row sample the difference is small, and it grows as row counts and string lengths increase. To find out for your own data, run the same aggregation both ways and compare:

```
.timer on

-- Group on the string
SELECT station_name, count(*) AS calls
FROM train_services
GROUP BY station_name;

-- Group on the key, then join the strings back
WITH rollup AS (
    SELECT station_id, count(*) AS calls
    FROM train_services_encoded
    GROUP BY station_id
)
SELECT s.station_name, rollup.calls
FROM rollup
LEFT JOIN stations s USING (station_id);
```

To see where the time goes, prefix each query with `EXPLAIN ANALYZE` and compare the timings of the aggregation operators. To compare memory use, set a lower `memory_limit` with `SET` and check which query starts spilling to disk first.

Variations

So far the examples use a single string column with a fixed set of values. The variations below cover a built-in alternative to the manual dimension table, encoding more than one string column, and keeping the keys current as new data arrives.

Comparison with ENUM

DuckDB's ENUM type is dictionary encoding built into the type system: values are stored as small integers, and DuckDB picks the integer width for you. The Lord of the Enums blog post (2021) benchmarks this, with a GROUP BY on an ENUM column running faster than the same grouping on the raw strings. You can create one from a query:

```
CREATE TYPE station_enum AS ENUM (
    SELECT DISTINCT station_name FROM train_services WHERE station_name IS NOT NULL ORDER BY station_name
);
```

If the set of values is known up front and rarely changes, an ENUM gives you most of the benefit with none of the extra joins. The dimension table is the better choice when:

- **New values keep arriving.** An ENUM's values are fixed when the type is created, so inserting an unknown value fails. A dimension table can grow.
- **You need attributes.** A dimension table can carry extra columns, such as the station's city or the line it sits on, that you can group or filter on without parsing strings.
- **The data leaves DuckDB.** Integer keys and a lookup table can be exported to Parquet or CSV and used in other tools.

Several String Columns

The pattern applies per column: build one dimension table for each high-repetition string column. In this dataset both `station_name` and the service `type` qualify. Each key then gets its own integer type, sized to the cardinality of its column, and a query only joins back the dimensions it needs. The `type` column has only 15 distinct values, so its key fits in a `UTINYINT`, while `station_name` still needs a `USMALLINT`:

```
CREATE OR REPLACE TABLE service_types AS
    SELECT
        type,
        (row_number() OVER (ORDER BY type))::UTINYINT AS type_id
    FROM (SELECT DISTINCT type FROM train_services WHERE type IS NOT NULL);

CREATE OR REPLACE TABLE train_services_encoded AS
    SELECT ts.* EXCLUDE (station_name, type), s.station_id, t.type_id
    FROM train_services ts
    LEFT JOIN stations s USING (station_name)
    LEFT JOIN service_types t USING (type);
```

The fact table now carries both keys, and a query joins back only the dimensions it reads. A count of calls per station needs `stations`, while a breakdown by service type needs `service_types`.

If two columns always appear together, such as `station_code` and `station_name`, a single dimension table keyed on the combination is often simpler. The fact table then stores one key instead of two, which makes it narrower still.

Keeping the Dimension Up to Date

When new data arrives, add unseen strings with keys that continue after the current maximum:

```
INSERT INTO stations
    SELECT
        n.station_name,
        ((SELECT max(station_id) FROM stations)
            + row_number() OVER (ORDER BY n.station_name))::USMALLINT AS station_id
    FROM (SELECT DISTINCT station_name FROM new_train_services WHERE station_name IS NOT NULL) n
    ANTI JOIN stations USING (station_name);
```

The `ANTI JOIN` keeps only the names that are not already in `stations`, so the existing keys stay untouched and each genuinely new name gets a key that continues past the current maximum.

Then encode the new rows the same way as in Step 3 and append them:

```
INSERT INTO train_services_encoded
    SELECT n.* EXCLUDE (station_name), s.station_id
    FROM new_train_services n
    LEFT JOIN stations s USING (station_name);
```

Appended keys no longer follow alphabetical order. If your queries rely on `ORDER BY station_id` matching `ORDER BY station_name`, rebuild the dimension table and re-key the fact table periodically, or sort on the string after the final join.

Narrow Keys Do Not Reduce the Number of Groups

Dictionary encoding makes each group smaller. It does not reduce how many groups there are, and that matters for very high-cardinality aggregations.

The report that prompted this post, duckdb/duckdb#14584, shows this. It grouped 9.2 billion rows into 320 million distinct groups and used far more memory than expected. The group key was already a `UBIGINT`, so there were no strings to encode.

The cause is how DuckDB parallelizes aggregation. Each thread first aggregates its share of the rows in its own thread-local hash table, and the partial results are combined at the end. This works well when each thread sees many repeats of the same groups, which is typical for real-world data.

In the report, with 8 threads, each thread saw about 1.1 billion rows: only about 3 times the number of distinct values, spread without any useful pattern. Nearly every group ended up in nearly every thread's table, so memory use approached the number of threads times the number of groups.

Narrow keys make each of those entries smaller, but they cannot prevent the duplication. When the number of groups is close to the number of rows each thread processes, reducing the number of `threads` helps more, because each thread holds its own copy of each group.

```
SET threads = 1;
```

This trades speed for memory, so reserve it for aggregations that would otherwise run out of memory or spill large amounts of data to disk. Data that is clustered by the group key also helps, as each thread then sees a smaller, more distinct set of groups.

When the Pattern Does Not Help

The pattern makes the schema and the queries more complex, so it is not always worth it.

Don't use it when:

- **The strings are short.** Values of up to 12 bytes are already stored inline, so the gap to an integer key is much smaller.
- **The column is nearly unique.** If most values are distinct, such as IDs or free text, the dimension table has almost as many rows as the fact table and little is saved.
- **You query the data once.** Building the dimension table and rewriting the fact table costs a full pass over the data. That cost is only worth paying if you query the data repeatedly.
- **You don't aggregate on the column.** If you only ever filter on or display the string, the benefit is limited.

In these cases, keep the string column as it is. If you are unsure, compare both versions as described in Measuring the Effect.

Conclusion

Grouping on repeated strings is expensive. By moving them into a small dimension table with sorted, narrow integer keys, DuckDB can aggregate on fixed-width integers and keep its hash tables compact. The strings come back in a final join against a result that is already small.

This approach helps most when you repeatedly aggregate on long strings that have few distinct values. You can find a condensed version of this tip in the Performance Guide.

Importing Data using Java Table Functions

Publication date: 2026-10-05

Authors: Guest Author, Geertjan Wielenga, Alex Kasko

TL;DR: When the analytics screen running on our main operational database got too slow, we moved a year of data into DuckDB on the same server. Getting the data in was the hard part. This is the story of every method we tried, and why a table function written in pure Java is the one we shipped.

> Guest blog post by a DuckDB user.

The Analytics Screen Problem

At the company where I work, we store vulnerability data for our clients as a collection of documents in MongoDB. Initially, our analytics screen performed well, but as our data volume increased, response times began to slow. For many months we did a lot of query optimization, restructuring, index additions, and saw significant improvement. However, execution times for certain scenarios remained in the tens of seconds, for example, when returning counts of closed/open vulnerabilities or when getting the top 10 most vulnerable business units for last six months. The main issue was in our MS Excel export feature which was taking 2-3 minutes for 50,000 records, and performance was often inconsistent. So we needed to find an alternative.

The plan was to import one year of records into an alternative database and run analytical queries on it. The requirements were to run all the queries in one second or less and export 50,000 records in Excel format under ten seconds. Anything more than that would be handled in background.

Due to internal requirements, the analytical database had to run on the same server as MongoDB. It needed to be actively developed, free, open-source and capable of working well with limited resources - and DuckDB checked all the boxes.

Sample Data Set and Code Snippets

To share my performance measurement results, I have prepared an anonymized dataset that roughly corresponds to the data used in our production environment. The size of this dataset has been scaled down significantly from the original to allow for faster experimentation.

The dataset is available within this link: vulnerability_sample.csv.zst, it contains 485168 records with 24 fields. Two notable fields are the VARCHAR “description” and “references”, which contain long strings with median lengths of 5,000 and 2,500 characters, respectively. Some records contain strings as large as 32,000 characters, making many operations on this dataset computationally expensive.

Additionally, along with the DuckLabs team, I have prepared several code snippets to illustrate data import in Java. While these snippets are significantly simplified compared to the code we run, their performance characteristics more or less matched the real ones. The code snippets are available in this GitHub repository staticlibs/duckdb_java_data_import.

MongoDB Community Extension

While exploring options to copy the data from MongoDB, I discovered the mongo community extension. My first proof-of-concept was created using this extension, and it was very simple to setup and use, the implementation looked like this:

```
INSTALL mongo FROM community;
ATTACH 'host=localhost port=27017 database=db1' AS m (TYPE MONGO);
CREATE OR REPLACE TABLE m.vuln1 AS FROM 'vulnerability_sample.csv.zst';
```

Using the sample dataset, this query took 55 seconds.

I was quickly able to get it up and running and import the data into DuckDB. I ran many of our analytical queries and it was very fast, under 100ms for GROUP BY and COUNT queries and I was able to export 50,000 records in less than 5 seconds. The only exception was when projecting large text columns, which caused a performance degradation; I have raised an issue about that.

Sorting the data provided further query performance improvements, and using `SET storage_compatibility_version = 'latest'`{:.language-sql .highlight} resulted in a significant reduction in the database file size.

Ultimately, the `mongo` community extension appears to be the easiest, most painless, and most performant way to set up a MongoDB-to-DuckDB pipeline.

However, we decided against using it due to the lack of "future-proof" guarantees associated with community extensions. DuckDB is a fast-moving project which is constantly improving, and we did not want to be held back from updating to new DuckDB version simply because an extension was not updated and can no longer be built.

The only real option for using the community extension would have been to build it locally - effectively maintaining an internal fork of it. While possible, we decided against this for two reasons. First, the burden of maintaining a C++ codebase was a major concern. Second, we prefer our data ingestion pipelines to be written in Java, that is much easier for us to maintain, and to be able to reuse parts of this pipeline across other internal data sources that we use from Java and that do not have ready-to-use DuckDB extensions.

Below, I describe the three ingestion approaches I tested, using the `mongo` extension's performance numbers as a baseline.

Besides these three, there were other attempts that were abandoned early. The notable one being the intermediate export from MongoDB into NDJSON/CSV/Parquet files and then importing those files directly from DuckDB. While promising at first, this approach was aborted due to the lack of disk space for the intermediate data copies.

Standard JDBC Batch Inserts

The initial attempt at importing data with Java utilized `INSERT` queries via the standard JDBC `executeBatch` API.

For simplicity, in this and subsequent examples, the data import period is divided into day-sized batches. With each batch fetched from MongoDB separately. This approach is imperfect (the number of records per day may be quite different, a single query from Mongo may be more performant), but is very easy to implement in parallel form and was found to not skew the results too much. Consequently, the following worker can be used to run `executeBatch`:

```
public class InsertWorker implements Runnable {
    Queue<LocalDate> queue; // concurrent queue distributed between worker threads
    MongoCollection<Document> collection; // Mongo query interface
    Connection connection; // DuckDB connection
    PreparedStatement ps; // DuckDB prepared statement

    @Override
    public void run() {
        this.ps = connection.prepareStatement("INSERT INTO staging VALUES(?, ?, ...)");
        for (;;) {
            LocalDate day = queue.poll();
            if (day == null) {// input queue exhausted
                break;
            }
            insertDayRecords(day, collection, ps);
        }
        // error handling omitted
    }

    static void insertDayRecords(LocalDate day, MongoCollection<Document> collection, PreparedStatement ps)
    {
        MongoCursor<Document> docs = collection.find(Filters.and( // other filters omittted
                Filters.gte("updated_date", day.atStartOfDay().toInstant(ZoneOffset.UTC)),
                Filters.lt("updated_date", day.plusDays(1).atStartOfDay().toInstant(ZoneOffset.UTC))
        )).iterator();
        int count = 0;
```

```
        while (docs.hasNext()) {
            Document doc = docs.next();
            ps.setString(1, doc.getObjectId("_id").toString());
            ps.setString(2, doc.getString("vendor"));
            // other 22 fields omitted
            ps.addBatch();
            count++;
        }
        if (count > 0) {
            ps.executeBatch();
        }
    }
}
```

Note, that it is not necessary to change anything about transactions handling, like disabling auto-commit, because calling `executeBatch` on all records for a given day automatically handles the start and commit/rollback of a single transaction.

By running eight of these workers in parallel, we import data into a staging DuckDB database file. We then copy the staging table into the final database file using an `ORDER BY updated_date DESC` clause, because sorting was found to drastically improve the resulting database file size.

> Sorting the data in a post-processing step allowed us to easily run the import in parallel. However this requires an intermediate copy of the data, and the sorting step itself is relatively expensive due to large VARCHAR fields.

With this approach, the total execution time on the sample data was 3 minutes and 37 seconds, with the final `CREATE TABLE... AS... ORDER BY` operation taking approximately 30 seconds.

The performance being 3.5x slower compared to the `mongo` extension is largely due to the lack of native "batch" insert support in DuckDB. The `executeBatch` is implemented on Java level. And while the statement is prepared only once per batch, the each entry in the batch is run in a separate database-level `INSERT`, thereby incurring a per-row overhead.

Java Appender Interface

The next attempt was to use the Appender interface, that is exposed in Java with the `Connection#createAppender` method. The worker for Appender can be structured very similar to the batch insert worker above:

```
public class AppendWorker implements Runnable {
    Queue<LocalDate> queue; // concurrent queue distributed between worker threads
    MongoCollection<Document> collection; // Mongo query interface
    Connection connection; // DuckDB connection
    DuckDBAppender appender; // Appender instance

    @Override
    public void run() {
        this.appender = connection.createAppender("staging_db", "main", "vuln1");
        for (;;) {
            LocalDate day = queue.poll();
            if (day == null) {// input queue exhausted
                break;
            }
            appendDayRecords(day, collection, appender);
        }
        // error handling omitted
    }

    static void appendDayRecords(LocalDate day, MongoCollection<Document> collection, DuckDBAppender
    appender) throws Exception {
        MongoCursor<Document> docs = collection.find(...).iterator();
        int count = 0;
        while (docs.hasNext()) {
            Document doc = docs.next();
            appender.beginRow();
```

```
            appender.append(doc.getObjectId("_id").toString());
            appender.append(doc.getString("vendor"));
            // other 22 fields omitted
            appender.endRow();
            count++;
        }
        if (count > 0) {
            appender.flush();
        }
    }
}
```

Unlike `executeBatch`, the DuckDB Appender performs true batching: a Data Chunk of 2,048 rows is prepared in Java and then flushed to DuckDB in a single operation.

With this approach, the total execution time on the sample data was 2 minutes and 2 seconds. This remains 2x slower than the `mongo` extension (or 1.5x if the final CTAS with sorting is excluded). On real data this performance was not acceptable so I continued the experiments.

COPY over a Java Table Function

After accepting defeat with the `Appender` and on the verge of giving up, thinking I cannot beat the `mongo` extension, I did what I should have done much earlier. I looked inside the `mongo` community extension code to find out what it is doing internally so I can do the same in Java. It appeared that the extension was using user-defined table functions. So I searched properly regarding table functions for the first time and found two following resources that appeared to be very helpful:

- Benchmarking DuckDB From Java: Fast INSERT, UPDATE, and DELETE
- DuckDB Table Functions in Java

Based on them the following parallel table function can be used with the sample dataset:

```
DuckDBFunctions.tableFunction()
            .withName("vuln_import")
            .withParameters(LocalDate.class, LocalDate.class)
            .withFunction(new ImportFunction())
            .register(conn);
stmt.execute("SET preserve_insertion_order = FALSE"); // necessary to allow parallel import
stmt.execute("CREATE TABLE staging.vuln1 AS FROM vuln_import('2026-02-13'::DATE, '2026-08-13'::DATE)");
```

Where the key parts of the `ImportFunction` are (see the full example in duckdb_java_data_import repo):

```
public class ImportFunction implements DuckDBTableFunction<BindData, GlobalData, LocalData> {

    @Override
    public BindData bind(DuckDBTableFunctionBindInfo info) {
        LocalDate from = info.getParameter(0).getLocalDate();
        LocalDate to = info.getParameter(1).getLocalDate();

        info.addResultColumn("_id", String.class);
        info.addResultColumn("vendor", String.class);
        // other 22 columns omitted
        return new BindData(from, to);
    }

    @Override
    public GlobalData init(DuckDBTableFunctionInitInfo info) {
        BindData bdata = info.getBindData();
        info.setMaxThreads(8);
        // open connection to Mongo, prepare days queue
        return new GlobalData(client, collection, queue);
    }

    @Override
```

```
    public long apply(DuckDBTableFunctionCallInfo info, DuckDBDataChunkWriter output) {
        GlobalData gdata = info.getInitData();
        LocalData ldata = info.getLocalInitData();

        while (ldata.cursor == null || !ldata.cursor.hasNext()) {
            LocalDate day = gdata.queue.poll();
            if (day == null) { // input queue exhausted
                return 0;
            }
            // fetch the day data from Mongo
            ldata.cursor = gdata.collection.find(...).iterator();
        }

        long row = 0;
        for (; ldata.cursor.hasNext() && row < output.capacity(); row++) {
            Document doc = ldata.cursor.next();
            output.vector(0).setString(row, doc.getObjectId("_id").toString());
            output.vector(1).setString(row, doc.getString("vendor"));
            // other 22 fields omitted
        }
        return row;
    }
}
```

In this implementation, we use our table function as the data source for the `CREATE TABLE` query. The `apply` method is invoked by DuckDB repeateadly, each time it can write up to 2048 (`output.capacity()`) rows to the output. The source is considered exhausted once the input queue is empty and the `apply` method returns a record count of zero.

To make the query over this function to run in parallel we need two bits:

- `info.setMaxThreads(8)` in `globalInit`: tells DuckDB that function supports multiple concurrent `apply` calls
- `SET preserve_insertion_order = FALSE`{:.language-sql .highlight}: tells DuckDB that it is not necessary to maintain the original order of records as they are read from the table function during insertion. If this flag is not set to FALSE, DuckDB will restrict apply calls to a single thread to preserve order. Since we perform sorting as a post-processing step after the import, preserving the insertion order is unnecessary.

With this approach, the total execution time on the sample data was 1 minute and 27 seconds. While it does not achieve the `mongo` extension time (it is not an apples-to-apples comparison due to day-sized queries and post-processing sorting) - it incurs the least overhead among all Java approaches I tried.

On real data, due to storage space restrictions, I ended up with a non-parallel version of the table function, where `ORDER BY` was run in the same `CREATE TABLE` query without intermediate copies. It allowed me to achieve the import times on-par with the `mongo` extension. And as a result we got a robust ingestion pipeline written in pure Java that can be adapted for other data sources.

A DuckDB Database with No Data in It

Publication date: 2026-10-07

Author: The DuckDB team

TL;DR: A DuckDB database file does not have to hold any data. It can store nothing but view definitions that point at Parquet files on object storage, so a few hundred kilobytes behave like a shared catalog over a whole data lake. This post shows how to build one, host it, and attach it read-only.

A DuckDB database file does not have to contain any data. It can store nothing but view definitions that point at files stored somewhere else, such as Parquet on object storage. The file stays a few hundred kilobytes no matter how large the data it describes, and anyone who attaches it sees a set of named relations that are ready to query.

Nikolas Goebel described this idea in DuckDB doesn't need data, which argues that modern databases have separated where data is stored from how it is read, to the point that most databases no longer need to hold any data of their own. From that perspective, DuckDB becomes a browser for the data cloud, where you access a dataset through a URL rather than a copy on your own disk.

In this blog post, we turn that idea into a small, shareable catalog.

The Idea

A DuckDB table stores its rows inside the database file. A view stores only the text of a query. When you read the view, DuckDB runs that query and returns the current result. If the query reads a remote Parquet file, the data lives in object storage and the view holds only the instruction for how to read it.

A database file made entirely of such views contains no rows of its own. It is a catalog: a set of named, ready-to-query relations created over data that stays in its original location and format. You can publish that catalog as a curated semantic layer over a data lake, where the view definitions encode the joins, filters, and column names that consumers should use, without copying or moving the underlying data.

Building a Catalog of Views

The examples use DuckDB's public train services dataset, which records one row for every stop a Dutch railway train makes. The data comes from the open datasets published by the Rijden de Treinen *(Are the trains running?)* application, and you can read it straight from its URL.

First, install the `httpfs` extension so that DuckDB can read over the network. This only needs to be run once.

```
INSTALL httpfs;
```

Now create a persistent database file and define a view over the remote data. Use a fully qualified location, a remote URL or an absolute path, so that the view resolves regardless of the client's working directory.

```
ATTACH 'rail_catalog.duckdb' AS rail;
USE rail;

CREATE VIEW services AS
    SELECT *
    FROM 'https://blobs.duckdb.org/train_services.parquet';
```

The catalog now exposes a `services` relation that reads from the remote file whenever it is queried.

```
SELECT departure_time, station_name, type
FROM rail.services
LIMIT 3;
```

departure_time	station_name	type
2023-05-15 00:00:00	Rotterdam Centraal	Intercity
2023-05-15 00:13:00	Delft	Intercity
2023-05-15 00:29:00	Den Haag HS	Intercity

Because the file stores only the view definition, it stays small no matter how large the referenced dataset grows. The `train_services.parquet` file holds 380,959 rows, yet the catalog that reads it is a few hundred kilobytes.

Hosting the Catalog

Upload `rail_catalog.duckdb` to any location that DuckDB can read over the network, for example an `s3://` bucket or an `https://` endpoint. The same bucket that already holds the Parquet files is a convenient place to put it.

Sharing the catalog then means sharing one URL instead of a list of individual file paths. The recipient does not need to know how many files back each view or where they are stored, only the address of the catalog.

Attaching the Catalog Read-Only

Consumers attach the catalog read-only and query the views as if they were local tables. Each query reads the current data from the referenced files, so the result reflects whatever is in object storage at that moment.

```
ATTACH 'https://example.com/rail_catalog.duckdb' AS rail (READ_ONLY);

SELECT station_name, count(*) AS calls
FROM rail.services
GROUP BY station_name
ORDER BY calls DESC
LIMIT 3;
```

station_name	calls
Utrecht Centraal	7663
Amsterdam Centraal	7591
Zwolle	5013

Consumers need the `httpfs` extension and, for `s3://` sources, credentials that grant access to the underlying data. DuckDB reads only the columns and row groups a query needs, so a filtered or aggregated query over a large remote dataset does not download the whole thing.

Surviving Changes to the Data Layout

Because a query reads the view rather than the files directly, the publisher can change the files without changing the query. The consumer runs the same query before and after.

Suppose the single Parquet file becomes too large and the publisher splits it into one file per year. Updating the view to read a directory of files is a one-line change in the catalog, and consumers notice nothing.

```
CREATE OR REPLACE VIEW services AS
    SELECT *
    FROM read_parquet('s3://my-bucket/rail/services/year=*/*.parquet');
```

A view can also keep the schema stable when the files change. If a later batch of files renames `station_name` to `station`, the view can map the new column back to the name consumers already use.

```
CREATE OR REPLACE VIEW services AS
    SELECT
        * EXCLUDE (station),
        station AS station_name
    FROM read_parquet('s3://my-bucket/rail/services/*.parquet');
```

If you repartition the files or move them to a new bucket, you only have to update the catalog. Anyone querying it keeps running the same queries as before.

When to Use It

Use this pattern when you want to share a dataset without sending the recipient a long list of individual file paths, and when the data is already in a format DuckDB reads well, such as Parquet. Readers get a set of named relations to query, and the publisher can change the file layout without breaking those queries.

Note the following limitations:

- Consumers must be able to access both the catalog file and every data source that its views reference.
- Connections over HTTPS and the S3 API are read-only, so the catalog cannot be modified in place. To change a view, edit a local copy and upload it again.
- Query performance depends on the network and the layout of the referenced files. The tips for querying remote files apply.

Conclusion

A DuckDB file that holds only views is a small catalog you can share as a single URL. The data stays in its original location, the publisher can change how it is stored, and the reader gets a stable set of relations to query.

For the full reference, see the Share a DuckDB Database That Only Stores Views guide.

Agent Mode in the DuckDB CLI

Publication date: 2026-10-09

Author: The DuckDB team

TL;DR: The DuckDB v2.0 CLI has an agent mode that gets AI coding agents to a correct result faster and with fewer tokens. When the CLI detects an agent, it prints compact Markdown tables instead of padded boxes, says clearly when a result was cut, stops runaway queries early, and reports errors as JSON. Long queries announce their expected cost before they start, so the agent can decide whether to wait.

Let's Ask Claude

We asked Claude to measure the effect of agent mode. In the experiment, Claude Code answered the 22 TPC-H questions on a scale factor 100 dataset. The questions were phrased in plain English, without any SQL. Each question was answered three times with agent mode and three times with `-no-agent`. All 132 runs produced the correct answer. With agent mode, the DuckDB output the model read went from 123.6k to 50.8k tokens, a 59% reduction. Claude summarized the result as follows:

> Quote Agent mode fixes the things about a terminal shell that quietly mislead a model. When I answered the 22 TPC-H questions through the DuckDB CLI, agent mode cut the tokens I read back by 60 percent, and every answer was still right.
>
> -- Claude Fable 5.1

The savings did not show up in the total cost or runtime. Each turn re-reads the agent's system prompt, which is much larger than the query output, so the saved tokens amount to about 0.5% of the total input. The agent mode runs also took slightly more turns (237 vs. 224). Claude wrote up the setup, the full results and a follow-up experiment in a report.

Background

The DuckDB CLI was built for a person sitting at a terminal. Its default renderer pads columns so they line up, draws borders around the result and prints the column types under the names:

name varchar	n int64	total int128	avg_amount decimal(10,2)
customer_0	3	777	259.00
customer_1	3	888	296.00

For a large result, it shows the first and last 20 rows with a row of dots in the middle:

name varchar	n int64	total int128	avg_amount decimal(10,2)
customer_0	3	777	259.00
customer_1	3	888	296.00
·	·	·	·
·	·	·	·
customer_4998	4	1203	300.75
customer_4999	2	510	255.00
5000 rows (40 shown)			4 columns

A person sees the dots and the `(40 shown)` footer and knows that rows are missing.

Today, many CLI invocations do not come from a person at all. Coding agents such as Claude Code, Codex, Cursor, Gemini CLI and GitHub Copilot run `duckdb -c "..."` once per shell call, capture stdout through a pipe and pass the text to a language model. This has quickly become a common way to use DuckDB.

This reader differs from a person in a few ways:

- It has no terminal, so there is no scrolling and no pager.
- It cannot ask for more. Whatever was printed is all it sees.
- It pays for every token it reads, including padding and border characters.
- It handles errors by pattern matching on text.

For this reader, the box renderer is a poor fit. The alignment padding costs tokens and carries no information for a model. A truncated result also looks a lot like a complete one. The footer says `(40 shown)`, but a model can skim past that line and then draw conclusions from data it never saw.

Agent mode changes the CLI's defaults for this reader. This post describes how the CLI detects an agent, what changes in the output and how to try it.

> Agent mode ships with DuckDB v2.0.

Detecting an Agent

Agents mark the commands they run with environment variables. There is no single convention for this yet, so the shell checks a list of them:

Variable	Set by
`AI_AGENT`, `AGENT`	Claude Code, Goose, Amp
`CLAUDECODE`	Claude Code
`CODEX_CI`, `CODEX_SANDBOX`, `CODEX_THREAD_ID`	Codex
`CURSOR_AGENT`	Cursor
`GEMINI_CLI`	Gemini CLI
`COPILOT_AGENT`, `COPILOT_CLI`, `COPILOT_AGENT_SESSION_ID`	GitHub Copilot

Agent mode turns on when three things are true: one of these variables is set, stdout is not a terminal, and no output format was given on the command line. If you pass `-csv`, `-json` or `-markdown`, you asked for a specific format, and agent mode stays out of the way.

Two flags override detection. `-agent` forces the mode on and `-no-agent` forces it off. The two combine with format flags, so `duckdb -agent -csv` keeps the agent behavior and only changes how the rows are printed. The `DUCKDB_AGENT_MODE` environment variable does the same from the environment, for when the command line cannot be changed, such as a script run from an agent's shell. For example, to manually enable agent mode, run:

```
DUCKDB_AGENT_MODE=1 duckdb
```

The `.show` command reports whether the mode is active and which agent was detected:

```
agent: claude-code
```

Not every agent sets a variable. For those, a run that fails while writing to a pipe prints one hint line on `stderr`, as long as no agent was identified and neither flag was given:

```
hint: -agent renders errors as JSON and results compactly for AI coding agents (duckdb -help lists all options)
```

An agent that nobody told about the mode can find it this way the first time something goes wrong.

What Changes in Agent Mode

Everything below is a default. Explicit settings such as `.mode`, `.maxrows`, `EXPLAIN (FORMAT ...)` or anything in your `.duckdbrc` still take precedence.

Compact Tables

Results are printed as Markdown tables without alignment padding, with the column type in the header cell:

```
| name:VARCHAR | n:BIGINT | total:HUGEINT | avg_amount:DECIMAL(10,2) |
|---|---|---|---|
| customer_0 | 3 | 777 | 259.00 |
| customer_1 | 3 | 888 | 296.00 |
```

A model reads the `|` separators and does not need the columns lined up. Since this is still valid Markdown, a person reading the agent's transcript gets a rendered table. On typical results this format is 25 to 65% smaller than the box output, with the largest savings on wide results.

Large Results

Printing every row would avoid silent truncation, but a million rows is 17 MB on the pipe. Agent mode prints a result of up to 1,000 rows and 10,000 bytes in full. Anything larger becomes a sample of the first and last 20 rows around an explicit marker row, with a footer that says what happened:

```
| 19 | customer_19 | 13.3 |
| … 4960 rows omitted … |
| 4980 | customer_30 | 86.0 |
...
first 20 and last 20 of 5000 rows (.maxbytes 0 for all), hash 7389c2f5ff3cefb6
```

The footer also tells the reader how to get the rest. The limits can be changed with `.maxrows N` and `.maxbytes N`, where `-1` or `0` means no limit. Long cell values are cut at 500 characters with a visible marker such as `…(+4500 chars)`, adjustable with `.maxcellwidth N`.

Stopping Early

The compact table needs no column widths, so the result can be streamed. Once the cap is reached, the remaining rows are only counted and hashed. After 100,000 more rows, the query is stopped and the count is reported as a lower bound. For example:

```
FROM range(1_000_000_000);
```

This returns in 30 milliseconds with the footer `first 20 of at least 102400 rows (query stopped early; .maxrows -1 for all)`.

A Hash of the Result

The footer includes a hash of the entire result, covering the rows past the cap. The hash does not depend on row order and distinguishes `NULL` from the string `'NULL'`. An agent that wants to know whether a result changed after an edit can compare two footers instead of printing two results.

The footer is printed for empty results, for results of 10 rows or more, and whenever the output was capped. It is left out when the query was stopped early, because the hash would not cover the whole result.

Errors as JSON

Errors go to `stderr` as JSON, using the existing `errors_as_json` setting. Errors raised by the shell itself, such as an unknown dot command, are wrapped in the same structure:

```
{"exception_type":"Binder","exception_message":"Referenced column \"foo\" not found in FROM
clause!\nCandidate bindings: \"bar\"","error_subtype":"COLUMN_NOT_
FOUND","location":"[7,3]","position":"7","name":"foo"}
```

The shell drops the `candidates` field, because the message already lists the candidates. Without that change, a single "no matching function" error weighs 2.7 kB.

Compact Query Plans

`EXPLAIN` defaults to a new `compact` format with one operator per line, indented by depth. Estimates, actual row counts and timings go in parentheses, followed by the operator's properties:

```
UNGROUPED_AGGREGATE (est=1) Aggregates: count_star()
  HASH_JOIN (est=0) Join Type: INNER; Conditions: range = range
    FILTER (est=2) Expression: (r > 2)
      RANGE (est=10) Function: RANGE
    FILTER (est=1) Expression: (s > 2)
      RANGE (est=5) Function: RANGE
```

`EXPLAIN ANALYZE` starts with a summary line such as `QUERY (time=0.0012s, read=1.2 MB)`. This is a regular format, so you can use `EXPLAIN (FORMAT compact)` outside agent mode too.

Cost Estimates and Progress

When the planner expects a statement to read a million rows or more, its estimate goes to `stderr` before execution starts:

```
estimate: ~7501215 rows read (lineitem ~6001215, orders ~1500000), ~4 rows returned
```

The agent can then decide whether to wait or to interrupt and rewrite the query. Smaller queries print nothing, since the line would only be noise. While a long query runs, the terminal progress bar is replaced by a plain line on `stderr` every 5 seconds:

```
progress: 42% (elapsed 12.0s, remaining ~16.5s)
progress: done (elapsed 28.7s)
```

Listing Tables

`.tables` prints one line per table, with an approximate row count and the columns:

```
memory.main.t1 (table, ~15 rows): a INTEGER PK, b VARCHAR
```

The usual box layout is hard to read through a pipe, and this gives an agent the schema in a single call.

The Startup Line

When agent mode turns on by detection, the shell prints one line on `stderr` before running anything:

```
duckdb agent mode on: CLAUDECODE is set and stdout is not a terminal; -no-agent turns it off, .help agent
explains the output
```

The line states why the mode is on and how to turn it off.

The first version printed three lines of explanation (634 bytes) on every run. That was larger than most query results, and a model keeps its context between runs anyway, so it only needs the explanation once.

The longer text is available via `.help agent`. It describes the output rules and settings, and it points to a few features an agent might not find by itself:

- `SET max_execution_time = <ms>`{:.language-sql .highlight} to put an upper bound on a query's runtime
- `DESCRIBE <query>`{:.language-sql .highlight} to get the result columns without running the query
- `SUMMARIZE` for a quick profile of a table or query
- `.tables` for the schema
- `duckdb_functions()` for function documentation

If you do not want the startup line, add `.startup_text none` to your `~/.duckdbrc`.

Trying It Out

If you use a coding agent, there is nothing to configure. Agents such as Claude Code, Codex, Cursor, Gemini CLI and GitHub Copilot already set one of the environment variables from the table above, and they capture a command's output through a pipe rather than a terminal. Those are two of the three conditions for agent mode.

So as long as the `duckdb` binary on the agent's path is v2.0 or newer and the agent does not request a specific output format with a flag such as `-csv`, `-json` or `-markdown`, the mode turns on by itself. A format flag is treated as a deliberate choice, so it switches agent mode off: the CLI assumes you want exactly that format and does not override it.

To see what the agent sees, run a query through a pipe with the flag set:

```
duckdb -agent -c "SUMMARIZE FROM 'my_data.parquet'" | cat
```

Conclusion

With agent mode, an agent spends fewer tokens per call and does not act on partial results as if they were complete. That means fewer steps to a correct result.

The current behavior is a first attempt, and the conventions around agents are a work in progress. If your agent is not detected, or the output confuses it, please open an issue or comment on the pull request. Transcripts of an agent being confused are especially useful.

Further Reading

- Shell: agent mode, render for AI coding agents when one is detected (pull request #26167)
- Shell: `DUCKDB_AGENT_MODE=1/0` forces agent mode on or off (pull request #26416)
- Shell: proposal to add `.mode llm` (Carlo Piovesan's proposal, the basis for the byte budget and early stop)

What Is Inside an Iceberg Table?

Publication date: 2026-10-13

Author: The DuckDB team

TL;DR: An Iceberg table is a tree of JSON, Avro and Parquet files. DuckDB can read every one of them, so you can walk the tree yourself with SQL: from the metadata file, through the manifest list and manifests, down to the data files and delete files.

An Apache Iceberg table looks like a regular table when you query it, but underneath it is a small tree of files: one JSON file at the top, a few layers of Avro files in the middle, and Parquet files at the bottom. When something is slow or looks wrong, it helps to know what is in that tree.

DuckDB can read all three formats, so you can inspect every layer with plain SQL. In this post, we create a small Iceberg table, change it a few times, and then walk through the files from the top down.

Getting Started

You need DuckDB with the `iceberg` extension and an Iceberg REST catalog to attach to. The catalogs page has instructions for catalogs such as Apache Polaris, Lakekeeper and Amazon S3 Tables.

A catalog is needed because the files on their own cannot tell you which version of the table is current. An Iceberg table writes a new `metadata.json` on every change and never overwrites the old one. The catalog holds the one piece of mutable state: a pointer to the current metadata file. When you query the table, DuckDB asks the catalog for that pointer first, then follows the tree of files down from there.

For this post, we run a catalog locally with the Iceberg REST catalog fixture, a small reference catalog that the Iceberg project publishes as a Docker image. It is the quickest catalog to set up: it runs from a single image, needs no Cloud account and no authentication, and keeps everything on local disk where you can open the files yourself.

We start it like this:

```
docker run -p 8181:8181 \
    -v /data/lake:/data/lake \
    -e CATALOG_WAREHOUSE=/data/lake \
    apache/iceberg-rest-fixture
```

Each part of that command maps to something DuckDB relies on later:

- `-p 8181:8181` publishes the catalog's REST API on `localhost:8181`, the address DuckDB connects to.
- `-v /data/lake:/data/lake` mounts a host directory into the container, so the metadata, manifest and Parquet files survive after the container stops and you can read them directly.
- `-e CATALOG_WAREHOUSE=/data/lake` tells the catalog where to write those files, which is why every path in the output below begins with `/data/lake`.

> **Note.** Run the examples below in the local DuckDB CLI, not in the browser-based DuckDB Shell, because a page served from the web cannot reach a catalog on your own `localhost`.

With the fixture running, attach it like this:

```
ATTACH 'warehouse' AS lake (
    TYPE iceberg,
    ENDPOINT 'http://localhost:8181',
    AUTHORIZATION_TYPE 'none'
);
```

This opens a catalog connection named `lake` that you query like any attached database.

- `TYPE iceberg` tells DuckDB to speak the Iceberg REST protocol.
- `ENDPOINT` is the address published by the `-p` flag above.
- `AUTHORIZATION_TYPE 'none'` matches the fixture, which expects no credentials.
- The first argument is the warehouse name. The fixture serves a single warehouse, so any name works.
- With object storage, the paths in the output below start with `s3://` instead of `/data/lake`, but everything else is the same.

A Table with History

We create a table of train services, partitioned by date, and change it three times: an insert of three days of data, an insert of a fourth day, and a delete.

```
CREATE SCHEMA lake.demo;
CREATE TABLE lake.demo.services (
    service_id BIGINT,
    service_date DATE,
    station VARCHAR,
    delay_min INTEGER
)
PARTITIONED BY (service_date);

INSERT INTO lake.demo.services
    SELECT
        i,
        DATE '2026-10-01' + (i % 3)::INTEGER,
        ['Utrecht Centraal', 'Delft', 'Leiden Centraal'][1 + (i % 3)::INTEGER],
        (i % 9)::INTEGER
    FROM range(1, 9_001) t(i);

INSERT INTO lake.demo.services
    SELECT i, DATE '2026-10-04', 'Den Haag HS', (i % 5)::INTEGER
    FROM range(9_001, 12_001) t(i);

DELETE FROM lake.demo.services
WHERE service_date = DATE '2026-10-02' AND delay_min = 4;
```

Each of the two inserts and the delete commits on its own. A write does not change the table in place: DuckDB reads the current snapshot, writes new data and metadata files alongside the old ones, and then asks the catalog to move the table pointer from that snapshot to the new one. That move is the commit.

The commit names the snapshot it expects to replace, so two writers that start from the same snapshot cannot both succeed. The first commit moves the pointer, and the second is rejected, because the snapshot it meant to replace is no longer current. The rejected write then retries on top of the new snapshot. When run in sequence from a single session, the three statements here never conflict, because each one starts from the snapshot the previous one committed.

The table now has 11,000 rows: the first insert added 9,000, the second added 3,000, and the delete removed the 1,000 rows of October 2 that had a delay of 4 minutes. Each of the three statements created a new snapshot, which `iceberg_snapshots` lists as follows:

```
SELECT sequence_number, snapshot_id, operation
FROM iceberg_snapshots(lake.demo.services)
ORDER BY sequence_number;
```

sequence_number uint64	snapshot_id uint64	operation varchar
1	7523747767380427861	append
2	3062515218093523051	append
3	6456433563311493009	delete

The Tree

Before going through the layers one by one, here is the whole structure. Each layer points to the files in the layer below it:

```
metadata.json                    one per table version
└── manifest list (Avro)         one per snapshot
    └── manifests (Avro)         one per write (for simplicity)
        ├── data files (Parquet)
        └── delete files (Parquet)
```

All of these files are immutable. Every change writes new files and a new `metadata.json`, then commits by moving the catalog pointer to it.

Layer 1: The Metadata File

The catalog stores a single pointer per table: the location of the current metadata file. `iceberg_load_table_response` returns what the catalog sent back when DuckDB loaded the table, including that location:

```
SELECT metadata_location
FROM iceberg_load_table_response(lake.demo.services);
```

metadata_location varchar
/data/lake/demo/services/metadata/00003-b79b5ffd-f118-44fd-9d8e-3551c147ef3b.metadata.json

The file name starts with `00003` because this is the fourth version of the metadata, counting from `00000` when the table was created.

It is a JSON document, so `read_json` can read it. We store the location in a variable to keep the queries short:

```
SET VARIABLE metadata_file = (
    SELECT metadata_location FROM iceberg_load_table_response(lake.demo.services)
);

SELECT
    "format-version",
    "current-snapshot-id",
    len(snapshots) AS snapshots,
    len("metadata-log") AS previous_metadata_files
FROM read_json(getvariable('metadata_file'));
```

format-version int64	current-snapshot-id int64	snapshots int64	previous_metadata_files int64
2	6456433563311493009	3	3

Besides the schema and partition spec, the metadata file holds the list of all snapshots that are still around, and each snapshot has a `summary` with counters. These counters are a cheap way to see what each change did, without reading any data:

```
SELECT
    s."sequence-number" AS seq,
    s.summary.operation,
    s.summary."added-records",
    s.summary."added-position-deletes",
    s.summary."total-records",
    s.summary."total-data-files",
    s.summary."total-delete-files"
FROM (SELECT unnest(snapshots) AS s FROM read_json(getvariable('metadata_file')))
ORDER BY seq;
```

```
┌───────┬───────────┬───────────────┬──────────────────────────┬───────────────┬──────────────────┬──────────────────────┐
│  seq  │ operation │ added-records │ added-position-deletes   │ total-records │ total-data-files │ total-delete-files   │
│ int64 │  varchar  │    varchar    │         varchar          │    varchar    │     varchar      │       varchar        │
├───────┼───────────┼───────────────┼──────────────────────────┼───────────────┼──────────────────┼──────────────────────┤
│     1 │ append    │ 9000          │ NULL                     │ 9000          │ 3                │ 0                    │
│     2 │ append    │ 3000          │ NULL                     │ 12000         │ 4                │ 0                    │
│     3 │ delete    │ NULL          │ 1000                     │ 12000         │ 4                │ 1                    │
└───────┴───────────┴───────────────┴──────────────────────────┴───────────────┴──────────────────┴──────────────────────┘
```

In the last row, after the delete, `total-records` is still 12,000 and there are still four data files. The delete did not touch the data files. It added one *delete file* with 1,000 entries. We will get to that file in a moment.

> **Note.** The Iceberg spec treats these `summary` fields as optional, so an engine is free to leave any of them out. The counters above are populated by DuckDB, but a table written by another engine may be missing some of them, in which case the field reads as `NULL`. Treat them as a convenient hint rather than a guarantee.

> **Tip.** The metadata file grows with every commit, because it keeps the snapshot list and a log of previous metadata files. On a table that receives thousands of small commits, it can become large enough to slow down loading the table. Expiring old snapshots, which most catalogs and engines can do, keeps it in check.

Layer 2: The Manifest List

Each snapshot points to a *manifest list*, an Avro file that lists the manifests making up that snapshot. The `avro` extension reads it with `read_avro`. We take the manifest list of the latest snapshot:

```
SET VARIABLE manifest_list = (
    SELECT manifest_list
    FROM iceberg_snapshots(lake.demo.services)
    ORDER BY sequence_number DESC
    LIMIT 1
);

SELECT
    regexp_replace(manifest_path, '.*/metadata/', '') AS manifest,
    content,
    sequence_number,
    added_files_count,
    added_rows_count
FROM read_avro(getvariable('manifest_list'));
```

```
┌──────────────────────────────────────┬─────────┬─────────────────┬──────────────────┬──────────────┐
│               manifest               │ content │ sequence_number │ added_files_count │ added_rows_count │
│               varchar                │  int32  │      int64      │      int32       │      int64       │
├──────────────────────────────────────┼─────────┼─────────────────┼──────────────────┼──────────────────┤
│ ffbf7076-8d36-46f0-aaa2-76e7d95442f0-m0.avro │       0 │               1 │                3 │             9000 │
│ 74fa5b1f-0908-4e34-964a-504d946671de-m0.avro │       0 │               2 │                1 │             3000 │
│ 5282235d-42b4-4b5d-a190-147a867994b7-m0.avro │       1 │               3 │                1 │             1000 │
└──────────────────────────────────────┴─────────┴─────────────────┴──────────────────┴──────────────┘
```

We created one manifest with every write. `content` is 0 for manifests that track data files and 1 for manifests that track delete files. The manifest list also stores the range of partition values in each manifest (not shown here), so a reader can skip entire manifests when a query

filters on the partition column.

The latest snapshot still includes the manifests written by the two earlier snapshots. To make a commit cheap to write, an engine can add a new manifest for every write, as happened here. This comes at the cost of read performance, because of the rapidly accumulating small manifests. An alternative method merges new additions into an existing manifest, which is more expensive for the writer but more optimized for the reader.

Layer 3: The Manifests

The manifests list the actual files, one entry per file, with the partition values, the row count and per-column statistics. They are Avro too:

```
SELECT
    status,
    data_file.content,
    regexp_replace(data_file.file_path, '.*/data/', '') AS file,
    data_file.record_count,
    data_file.partition
FROM read_avro('/data/lake/demo/services/metadata/*-m*.avro')
ORDER BY file;
```

```
┌────────┬─────────┬──────────────────────────────────────────────────────────────────────────────┬──────────┬──
│ status │ content │                                   file                                       │ record_
count │                partition                │
│ int32  │  int32  │                                  varchar                                     │    int64
│  struct(identity_service_date_2 date)   │
├────────┼─────────┼──────────────────────────────────────────────────────────────────────────────┼──────────┼──
│      1 │       0 │ service_date=2026-10-01/01a12bb2-ad80-7cf4-8f63-beee9619f7c0.parquet         │
3000 │ {'identity_service_date_2': 2026-10-01} │
│      1 │       0 │ service_date=2026-10-02/01a12bb2-ad7f-7690-bda8-690b622b119a.parquet         │
3000 │ {'identity_service_date_2': 2026-10-02} │
│      1 │       1 │ service_date=2026-10-02/58a752c7-9925-4add-99ca-785505e17589-deletes.parquet │
1000 │ {'identity_service_date_2': 2026-10-02} │
│      1 │       0 │ service_date=2026-10-03/01a12bb2-ad80-71f7-b4cd-58684f05b129.parquet         │
3000 │ {'identity_service_date_2': 2026-10-03} │
│      1 │       0 │ service_date=2026-10-04/01a12bb2-adbe-790f-bde9-50db3acba4b2.parquet         │
3000 │ {'identity_service_date_2': 2026-10-04} │
└────────┴─────────┴──────────────────────────────────────────────────────────────────────────────┴──────────┴──
```

`status = 1` means the file was added in the snapshot that wrote the manifest. `content` is 0 for a data file and 1 for a position delete file. The partition value is stored per file, which is what lets a reader skip files without opening them.

The extension has a function that does this for you. `iceberg_metadata` follows the current snapshot through the manifest list and the manifests and returns one row per file:

```
SELECT
    manifest_content,
    regexp_replace(file_path, '.*/data/', '') AS file,
    record_count
FROM iceberg_metadata(lake.demo.services)
ORDER BY file;
```

```
┌────────────────────┬──────────────────────────────────────────────────────────────────────────────┬──────────┐
│ manifest_content │                                   file                                       │ record_count │
│     varchar      │                                  varchar                                     │    int64     │
├────────────────────┼──────────────────────────────────────────────────────────────────────────────┼──────────┤
│ DATA               │ service_date=2026-10-01/01a12bb2-ad80-7cf4-8f63-beee9619f7c0.parquet         │
3000 │
│ DATA               │ service_date=2026-10-02/01a12bb2-ad7f-7690-bda8-690b622b119a.parquet         │
3000 │
│ DELETE             │ service_date=2026-10-02/58a752c7-9925-4add-99ca-785505e17589-deletes.parquet │
1000 │
```

```
│ DATA              │ service_date=2026-10-03/01a12bb2-ad80-71f7-b4cd-58684f05b129.parquet          │
3000 │
│ DATA              │ service_date=2026-10-04/01a12bb2-adbe-790f-bde9-50db3acba4b2.parquet          │
3000 │
└──────────────────┴──────────────────────────────────────────────────────────────────────────────┴──────────────┘
```

Reading the Avro files directly is useful when you want to see fields that the function does not return, or when you are looking at an older snapshot or a table that is not in a catalog.

Column Statistics

The manifests also store, for every data file, the lower and upper bound of each column. `iceberg_column_stats` returns them in a readable form:

```
SELECT
    column_name,
    regexp_replace(file_path, '.*/data/', '') AS file,
    lower_bound,
    upper_bound
FROM iceberg_column_stats(lake.demo.services)
WHERE column_name = 'service_id'
ORDER BY file;
```

```
┌──────────────┬──────────────────────────────────────────────────────────────────────────────┬──────────────┬──────────────┐
│ column_name │                                  file                                        │ lower_bound │
upper_bound │
│   varchar   │                                 varchar                                      │   varchar   │
varchar   │
├──────────────┼──────────────────────────────────────────────────────────────────────────────┼──────────────┼──────────────┤
│ service_id  │ service_date=2026-10-01/01a12bb2-ad80-7cf4-8f63-beee9619f7c0.parquet          │ 3           │
9000        │
│ service_id  │ service_date=2026-10-02/01a12bb2-ad7f-7690-bda8-690b622b119a.parquet          │ 1           │
8998        │
│ service_id  │ service_date=2026-10-02/58a752c7-9925-4add-99ca-785505e17589-deletes.parquet │ NULL
│ NULL        │
│ service_id  │ service_date=2026-10-03/01a12bb2-ad80-71f7-b4cd-58684f05b129.parquet          │ 2           │
8999        │
│ service_id  │ service_date=2026-10-04/01a12bb2-adbe-790f-bde9-50db3acba4b2.parquet          │ 9001
│ 12000       │
└──────────────┴──────────────────────────────────────────────────────────────────────────────┴──────────────┴──────────────┘
```

These bounds are what DuckDB uses to skip files when you filter on a column that is not the partition column. A query with `WHERE service_id > 10000` only has to open the file for October 4, because the bounds of the other three files rule them out.

The ranges for the first three days overlap almost completely, because the rows were spread over the days round-robin. Data that is written in the order of a commonly filtered column gets narrower bounds and more skipping.

Partition Statistics

The bounds above are per column, per data file. For the partition column there is a coarser view, one row per manifest, that the manifest list keeps so a reader can skip an entire manifest before opening any of its files. `iceberg_partition_stats` returns it:

```
SELECT
    partition_field_name,
    partition_field_transform,
    lower_bound,
    upper_bound
FROM iceberg_partition_stats(lake.demo.services)
ORDER BY lower_bound;
```

partition_field_name varchar	partition_field_transform varchar	lower_bound varchar	upper_bound varchar
identity_service_date_2	identity	2026-10-01	2026-10-03
identity_service_date_2	identity	2026-10-02	2026-10-02
identity_service_date_2	identity	2026-10-04	2026-10-04

There is one row per manifest, so the bounds trace the three writes: the first insert spread three days across one manifest, the second insert added October 4, and the delete touched only October 2. The `partition_field_transform` is `identity` because we partitioned on the date column directly. A query with `WHERE service_date = DATE '2026-10-04'` can rule out the first and third manifests on these bounds alone, without reading a single manifest entry.

Layer 4: Data Files and Delete Files

At the bottom of the tree are Parquet files. The data files are regular Parquet that any reader can open. The delete file works differently. It is a *position delete* file, and it lists rows to ignore by file and row position:

```
SELECT regexp_replace(file_path, '.*/data/', '') AS data_file, pos
FROM read_parquet(
    '/data/lake/demo/services/data/*/*-deletes.parquet',
    hive_partitioning = false
)
ORDER BY pos
LIMIT 5;
```

data_file varchar	pos int64
service_date=2026-10-02/01a12bb2-ad7f-7690-bda8-690b622b119a.parquet	1
service_date=2026-10-02/01a12bb2-ad7f-7690-bda8-690b622b119a.parquet	4
service_date=2026-10-02/01a12bb2-ad7f-7690-bda8-690b622b119a.parquet	7
service_date=2026-10-02/01a12bb2-ad7f-7690-bda8-690b622b119a.parquet	10
service_date=2026-10-02/01a12bb2-ad7f-7690-bda8-690b622b119a.parquet	13

This is how the extension implements DELETE and UPDATE: it leaves the data file alone and records the positions of the deleted rows, a strategy called merge-on-read.

Every reader then has to combine the two files to get the correct rows. Writes are cheap, and reads pay a small cost, which grows with the number of delete files. We set `hive_partitioning = false` because the `service_date=...` directory would otherwise be added as an extra column.

For tables with `format-version` 3, the same information is stored as a compact binary deletion vector in a Puffin file instead of a Parquet file.

Why Look at This at All?

Most of the time, you do not need to. `FROM lake.demo.services` returns the right 11,000 rows, and the extension handles the tree. It becomes useful when something does not behave as you expect:

- **A query reads more files than you expected.** Compare the partition values and column bounds in the manifests with your filter.
- **A table gets slower over time.** Count the manifests and delete files. Many small commits and many deletes both add files that every query has to process.
- **Row counts look off.** The snapshot summaries show how many rows each commit added and deleted.
- **Another engine wrote the table.** The manifests show what it wrote, including properties that DuckDB does not use.

Since every layer is a file in a format DuckDB already reads, you can answer these questions with the same tool you use to query the data.

Further Reading

- Writes in DuckDB-Iceberg
- Iceberg in the Browser
- New DuckDB-Iceberg Features in v1.5.3

From JSON Strings to Typed Columns

Publication date: 2026-10-16

Author: The DuckDB team

TL;DR: JSON often arrives as a text column inside a table, not as a file. In this post, we infer a schema from such a column, adjust it, and turn the column into typed columns, without first writing the data to a file.

DuckDB's `read_json` turns JSON files into tables. It reads through part of the data to work out a schema, then gives you typed columns. However, a lot of JSON does not come as files. It comes as a text column: event payloads in a Postgres table, API responses stored by an ingestion job, a `metadata` column in a CSV export.

People regularly ask how to get the `read_json` treatment for such a column, and one workaround that comes up is to COPY the column to a file and read the file back.

You do not need the round trip. In this post, we get the same result directly in SQL.

The Data

Here is a small table of events, with the payload stored as VARCHAR:

```
CREATE TABLE events (id INTEGER, payload VARCHAR);
INSERT INTO events VALUES
    (1, '{"user": {"id": 17, "country": "NL"}, "action": "click", "ts": "2026-10-01 10:00:00"}'),
    (2, '{"user": {"id": 23, "country": "BE"}, "action": "view", "ts": "2026-10-01 10:00:05", "duration_
ms": 1200}'),
    (3, '{"user": {"id": 17}, "action": "purchase", "ts": "2026-10-01 10:02:11", "items": [{"sku": "A-1",
"qty": 2}, {"sku": "B-7", "qty": 1}]}');
```

Like most real payloads, the objects do not all have the same keys. Only the second event has a `duration_ms`, only the third has `items`, and the third has no `country`.

Picking out a Few Fields

If you only need a handful of fields, the JSON extraction operators are enough. `->` returns JSON and `->>` returns VARCHAR. Both accept a key name or a JSON path:

```
SELECT
    id,
    payload ->> '$.user.country' AS country,
    payload ->> 'action' AS action,
    (payload ->> 'duration_ms')::INTEGER AS duration_ms
FROM events;
```

id int32	country varchar	action varchar	duration_ms int32
1	NL	click	NULL
2	BE	view	1200
3	NULL	purchase	NULL

Missing keys come back as `NULL`, which is what you want for optional fields. However, this approach parses the JSON again for every extracted field, and it becomes verbose once you want more than a few columns.

Inferring a Structure from the Column

`json_structure` returns the structure of a single JSON value. Its aggregate variation, `json_group_structure`, merges the structures of all values in a column into one. It needs JSON input, so cast the VARCHAR column first:

```sql
SELECT json_group_structure(payload::JSON) AS structure
FROM events;
```

structure json
{"user":{"id":"UBIGINT","country":"VARCHAR"},"action":"VARCHAR","ts":"VARCHAR","duration_ms":"UBIGINT","items":[{"sku":"VARCHAR","qty":"UBIGINT"}]}

Every key that appears in any row is in there, including `duration_ms` and the array of `items`.

For a large table, you do not have to scan every row for this. Running it over a sample, for example `FROM events USING SAMPLE 10_000 ROWS`, is much cheaper, at the risk of missing keys that are very rare.

Adjusting the Types

The inferred structure needs a few fixes. `ts` came out as VARCHAR because JSON has no timestamp type, and the integers came out as `UBIGINT`, which is wider than needed.

Since the structure is itself a JSON document, you can override parts of it with `json_merge_patch`. Store the result in a variable so you can use it in the next query:

```sql
SET VARIABLE payload_structure = (
    SELECT json_merge_patch(
        json_group_structure(payload::JSON),
        '{
            "user": {"id": "INTEGER"},
            "ts": "TIMESTAMP",
            "duration_ms": "INTEGER",
            "items": [{"sku": "VARCHAR", "qty": "INTEGER"}]
        }'
    )
    FROM events
);
```

The patch only lists what you want to change. Keys you leave out, such as `user.country` and `action`, keep their inferred types.

Building the Table

`json_transform` (also available as `from_json`) converts a JSON value into a DuckDB `STRUCT` following the structure you give it. `unnest` then expands the top-level fields of that struct into columns:

```sql
CREATE TABLE events_typed AS
    SELECT id, unnest(json_transform(payload, getvariable('payload_structure')))
    FROM events;

DESCRIBE events_typed;
```

column_name varchar	column_type varchar	null varchar	key varchar	default varchar	extra varchar

id	INTEGER	YES	NULL	NULL	NULL
action	VARCHAR	YES	NULL	NULL	NULL
user	STRUCT(country VARCHAR, id INTEGER)	YES	NULL	NULL	NULL
ts	TIMESTAMP	YES	NULL	NULL	NULL
duration_ms	INTEGER	YES	NULL	NULL	NULL
items	STRUCT(sku VARCHAR, qty INTEGER)[]	YES	NULL	NULL	NULL

The JSON is parsed once, at load time. From here on, the columns are regular DuckDB columns, stored and compressed by type, and queries on them do not touch JSON at all:

```
SELECT id, "user".id AS user_id, "user".country, action, ts
FROM events_typed;
```

id int32	user_id int32	country varchar	action varchar	ts timestamp
1	17	NL	click	2026-10-01 10:00:00
2	23	BE	view	2026-10-01 10:00:05
3	17	NULL	purchase	2026-10-01 10:02:11

Nested Structs and Name Collisions

Recursive unnesting with `unnest(..., recursive := true)` flattens nested structs in a single pass. By default it names each column after its innermost key, so the `id` inside `user` ends up as a column called `id`, right next to the event's own `id`:

```
SELECT id, unnest("user", recursive := true)
FROM events_typed
LIMIT 1;
```

id int32	country varchar	id int32
1	NL	17

With two columns both named `id`, a later query cannot tell which one you mean. Pass `keep_parent_names := true` to prefix each column with the keys it came from, so names from different levels stay distinct. On the full payload, `user.id` and `user.country` no longer clash with the top-level columns, and you can select them by their prefixed names:

```
SELECT id, "user.id", "user.country"
FROM (
    SELECT id, unnest(json_transform(payload, getvariable('payload_structure')), recursive := true, keep_
    parent_names := true)
    FROM events
    LIMIT 1
);
```

id int32	user.id int32	user.country varchar
1	17	NL

Arrays of Objects

In `events_typed`, the `items` array is stored as a `STRUCT(sku VARCHAR, qty INTEGER)[]`, a list of structs, as the `DESCRIBE` output above shows. To get one row per item, `unnest` the list in the `FROM` clause:

```
SELECT id, u.item.sku, u.item.qty
FROM events_typed, unnest(items) AS u(item);
```

id int32	sku varchar	qty int32
3	A-1	2
3	B-7	1

Events without items do not appear here. If you need to keep them, write the `unnest` as a lateral join: `FROM events_typed LEFT JOIN unnest(items) AS u(item) ON true`.

Values That Do Not Fit

By default, `json_transform` does not raise an error on a value it cannot convert. The value becomes `NULL` instead:

```
SELECT json_transform('{"qty": "two"}', '{"qty": "INTEGER"}') AS result;
```

result struct(qty integer)
{'qty': NULL}

That is convenient for messy data, but it can hide problems. `json_transform_strict` raises an error instead:

```
SELECT json_transform_strict('{"qty": "two"}', '{"qty": "INTEGER"}') AS result;
```

```
Invalid Input Error: Failed to cast value to numerical: "two"
```

Use the strict version while you are working out the structure, and switch to the lenient version once you know which oddities you are willing to accept.

Should the Column Be VARCHAR or JSON?

This question comes up regularly in the DuckDB discussions. The JSON type is stored exactly like a VARCHAR, so there is no difference in size or scan speed. What it adds is validation on the way in, since an invalid document fails the insert, and the guarantee that the extraction functions do not have to deal with malformed input.

If you will query fields from the column many times, the type of the raw column matters much less than whether you extract the fields at all. Parsing JSON at query time is work that has to be repeated for every query. Turning it into typed columns once, as shown above, does that work a single time.

Further Reading

For more on working with JSON in DuckDB, see the following blog posts and documentation pages:

- Shredding Deeply Nested JSON, One Vector at a Time, on how DuckDB reads JSON files
- Reconciling JSON in DuckDB, One Patch at a Time, on combining JSON documents with `json_merge_patch`
- JSON Overview and JSON Functions
- The JSON Type
- `unnest` and the STRUCT data type

Benchmarking DuckDB v2.0-dev on NVIDIA Vera

Publication date: 2026-10-16

Author: Gábor Szárnyas

TL;DR: We put NVIDIA's new Vera CPU through its paces using the full TPC-H workload on scale factor 1,000.

We love running benchmarks on all sorts of new hardware. This often includes somewhat exotic hardware – where we sometimes need to compile a new DuckDB binary. This time, in collaboration with the NVIDIA team, we got our hands on their new Vera chip, aimed at agentic workloads.

NVIDIA Vera comes with 88 "Olympus" cores, Spatial Multithreading and up to 1.2 terabytes per second (TB/s) of memory bandwidth. What's more, Vera runs arm64 binaries – so it runs DuckDB's regular build without any modifications.

To see how this new hardware performs, we ran the full TPC-H benchmark (not just the read queries!) on scale factor 1,000. Our setup included two server platforms: an x86 baseline (Intel Xeon 6) and NVIDIA Vera.

Environment

System Setup

Both machines ran Ubuntu 26.04. Their hardware included 768 GB of memory and a single NVMe SSD with 3+ TB of storage, formatted as ext4. At the time of writing, DuckDB v2.0 was not released yet, so we used a preview build, `1.6.0.dev379`.

To run the experiments, we used the `duckdb-tpch` benchmark harness. To re-create our benchmark environment, see our script below.

> **Tip.** To generate the large-scale TPC-H dataset efficiently, the repository uses the wonderful `tpchgen-cli` tool, a parallel Rust rewrite of the TPC-H data generator. It also uses the old generator to produce update and query streams.

This workload is relatively large-scale (SF1,000) and highly concurrent during the throughput run. To get the best performance out of DuckDB, we set the following flags:

```
SET block_allocator_memory = '500G';
SET allocator_background_threads = true;
```

x86 Baseline: Intel Xeon 6

The x86 baseline setup used an Intel Xeon 6 CPU. The CPU had 96 cores with hyper-threading, yielding 192 threads. However, we found that DuckDB performs best when using 96 threads, so we configured it accordingly:

```
SET threads = 96;
```

NVIDIA Vera

The second machine had the NVIDIA Vera CPU, which packs 88 cores with Spatial Multithreading. We conducted the benchmark on a dual-socket system but restricted DuckDB to a single socket using the `numactl` command:

```
numactl --cpunodebind=0 --membind=0 python3 benchmark.py
```

We also pinned the number of threads:

```
SET threads = 88;
```

Benchmark Runs and Results

We performed an initial warmup run and discarded its result. Then, we conducted three complete runs in sequence and took the median value of the *QphH@Size* composite scores:

	x86 QphH@Size	Vera QphH@Size
Run 1	2,120,588.82	3,073,292.99
Run 2	2,022,795.01	3,131,306.27
Run 3	2,109,782.19	3,052,102.87
Median score	**2,109,782.19**	**3,073,292.99**

The results show that the Vera platform has a 46% performance advantage on TPC-H SF1,000 over the x86 baseline (Intel Xeon 6). In fact, if we were to audit these results through the official TPC-H channels, both would stand a chance to be in the top 3 among the TPC-H Top Performance Results on SF1,000.

We attribute the performance advantage of Vera to three key factors. First, the high memory bandwidth enables fast scans. Second, combined with low memory latency, it keeps the cores continuously occupied during processing. Finally, the Olympus cores' large caches allow efficient access to the hash tables built during aggregations and joins.

We are looking forward to continuing this benchmark by using DuckDB v2.0.0's stable build, running on the full dual-socket setup and incorporating different benchmarks.

Announcing DuckDB 2.0.0 LTS “Cinnamon Teal”

Publication date: 2026-10-21

Author: The DuckDB team

TL;DR: The DuckDB team is happy to announce that today we're releasing DuckDB version 2.0.0 LTS, codenamed “Cinnamon Teal” (Spatula cyanoptera). This is our biggest feature release so far, and it ships a polished version of the Quack protocol, asynchronous I/O, a first-class `VARIANT` type, agent mode, triggers, an extensible SQL parser and much more, built from close to 20,000 commits since DuckDB 1.5.

We are proud to release DuckDB 2.0.0, codenamed “Cinnamon Teal” after the *Spatula cyanoptera*, a strikingly reddish-brown duck found in the western Americas.

While DuckDB 1.0 was all about stability and backward compatibility, DuckDB 2.0 is above all a feature release. In this blog post, we summarize the highlights and then walk through the breaking changes and how to migrate. We showcased many of the new features in “A Preview of DuckDB v2.0”, which covers each one in more depth. For the complete release notes, see the release page on GitHub.

> **Tip.** To install the new version, please visit the installation page.

With this release, there are currently three DuckDB lines available. DuckDB 2.0 is a long-term support (LTS) release with support until at least October 2027. The previous LTS release, version 1.4, is still supported until November 17, 2026. To ensure a smooth transition and allow sufficient time for testing, we extended the support period of DuckDB 1.5 until March 2027, effectively also turning it into an LTS release. For details, see the release calendar.

To find your way around the documentation for all of the new features, we also published a “What's New in the Docs for 2.0” page that links each feature below to its documentation.

DuckDB as a Server

DuckDB has been an in-process database since day one. In DuckDB 1.5.3, released in May, we introduced the Quack protocol that lets you run DuckDB in a traditional client-server setup, along with the `quack` extension, which implements DuckDB's native protocol, so any DuckDB process can serve its databases.

Quack and `CONNECT`

DuckDB 2.0.0 ships a polished version of the Quack protocol with substantial performance and stability improvements. It also introduces the new `CONNECT` statement that allows attaching and switching to a remote database in a single command for better ergonomics:

For simplicity, this example runs the server and the client on the same machine. By default, the server only listens on `localhost`. To accept connections from other machines, keep the server on `localhost` and put it behind a TLS-terminating reverse proxy. The `allow_other_hostname` option, which exposes the server directly, is only meant for testing.

The `CONNECT` statement is not limited to Quack: it points your session at any remote database that supports it, including PostgreSQL and MySQL. For these, the new remote pushdown optimizer ships queries directly to the remote server instead of pulling tables over the wire.

Quack in the CLI

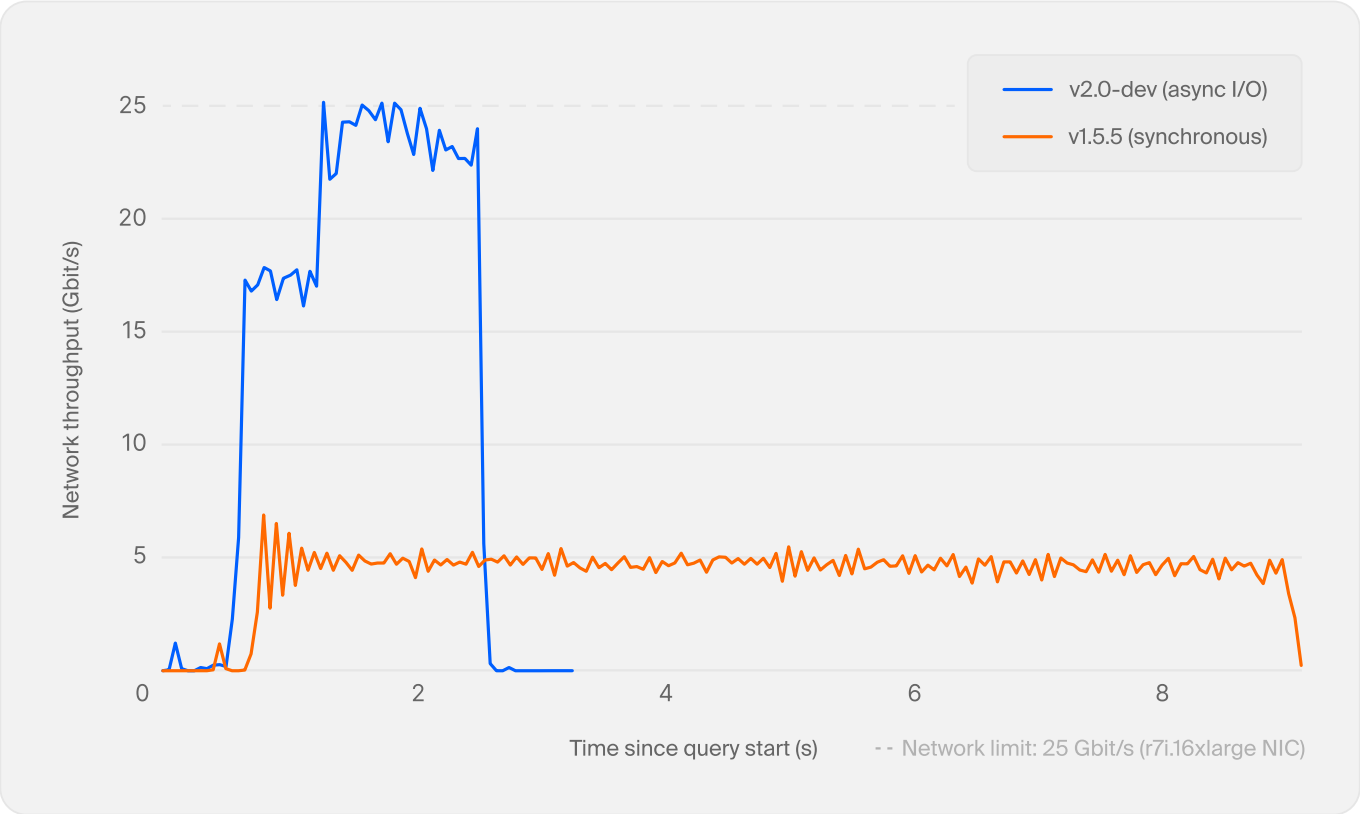
The CLI also has shortcuts for setting up a Quack client-server pair: the new `-serve` and `-connect` arguments start a Quack server or connect to one directly from the command line:

Performance

Asynchronous I/O

DuckDB 2.0 introduces asynchronous I/O for reading Parquet and CSV files, as well as DuckDB's native storage format. The I/O layer now scales independently from the query processing layer, issuing reads in the background and keeping multiple requests in flight. This means far more parallelism for remote reads and dramatically faster queries on object storage such as `s3://`.

We described the design in detail in a dedicated blog post. In that post, we showed the network throughput over the course of a query with and without asynchronous I/O:



{: .lightmode-img }

Network throughput. Higher utilization and shorter runtime are better.{: .caption }

The `VARIANT` Type Becomes a First-Class Citizen

The `VARIANT` type stores semi-structured, schema-less data such as JSON while keeping the performance of columnar execution. So you can think of it as JSON on steroids: a `VARIANT` column can store differently shaped data in every row, but unlike JSON it is not a text format.

DuckDB 1.5 already introduced `VARIANT`, but DuckDB 2.0 fully integrates it. DuckDB 2.0 detects the common structure in your data and "shreds" it, so it compresses well and executes fast. The pipeline works end to end, from shredded execution straight out of storage to shredded `VARIANT` reading and writing for Parquet, backed by a family of `variant_*` functions. For example:

```
CREATE TABLE events (payload VARIANT);
INSERT INTO events
VALUES
    ('{"user": {"id": 42, "tags": ["a", "b"]}}'::JSON::VARIANT),
    ('{"event": "click", "x": 10, "y": 20}'::JSON::VARIANT);

SELECT variant_type(payload), variant_keys(payload)
FROM events;
```

variant_type(payload) varchar	variant_keys(payload) varchar[]
OBJECT OBJECT	[user] [event, x, y]

Agent Mode in the CLI

Coding agents such as Claude Code, Codex and Cursor increasingly drive the DuckDB CLI on their users' behalf. The 2.0 CLI detects when it is being run by an agent and switches to agent mode: it prints compact Markdown tables instead of padded boxes, says clearly when a result was truncated, stops runaway queries early and reports errors as JSON.

Long-running SQL queries report their expected cost prior to running, so the agent can decide whether to wait. In our experiments on TPC-H, agent mode cut the amount of DuckDB output the model had to read by 59%. You can turn it *on* for manual testing using the `-agent` flag or instruct your agent to turn it *off* using the `-no-agent` flag.

Extensibility

A Brand New SQL Parser

DuckDB has always used a parser derived from PostgreSQL's. In 2.0, our own modern, extensible PEG-based parser, which debuted as an opt-in feature in DuckDB 1.5, becomes the default. Extensions can now hook into the grammar itself (an idea we first explored in our research on runtime-extensible parsers), so expect extensions that expose entirely new SQL syntax. Error messages now come with precise source locations. For details, see the PEG parser documentation. The new parser is designed to be compatible with the old one, apart from a few deliberate changes listed in the migration guide. If you notice any other differences, please file an issue. In the meantime, you can switch back to the old parser using the `legacy_parser` extension by running `SET enable_legacy_parser = true;`{:.language-sql .highlight}.

The old parser had served DuckDB since its first commit in 2018. Its grammar was generated with YACC/Bison, which produces an LALR(1) parser, so as DuckSQL grew, even small additions could collide with existing rules and raise `shift/reduce` or `reduce/reduce` conflicts. A PEG grammar avoids this by trying alternatives in the order they are written, so adding syntax no longer risks breaking unrelated rules. Python itself moved to a PEG-based parser in version 3.9 for the same flexibility.

Write Extensions Once and Host Them Yourself

DuckDB 2.0 ships a revamped C API v2, which is considerably more powerful than its predecessor and exposes components such as file systems and table functions to extensions. Before marking it as stable, we would like to test it in the field, so please give it a try and send us your feedback!

You can also register and host your own trusted extension repositories, so an organization can host and sign its own extensions and have them install and load just like the built-in ones:

```
SET allow_extension_repositories = 'allowed';

CREATE EXTENSION REPOSITORY my_repo
    WITH PREFIX 'https://extensions.example.org';

INSTALL AND LOAD my_ext FROM my_repo;
```

A repository is a name, a URL prefix and one or more RSA public keys that are trusted to sign the extensions served from it. The prefix can point at anything DuckDB can read: a local path, `https`, `s3`, you name it. At `CREATE` time, DuckDB fetches the repository's public keys and pins them into the repository definition, printing each key's SHA-256 fingerprint so you can compare it against one published out of band.

This feature is still a work in progress and its syntax may change in future releases. For more details, see the Extension Repositories page.

Core Extension Updates

DuckDB's core extensions are also shipping several new features, fixes and performance optimizations. These are too numerous to all list here, so we only highlight a few of them and will cover them in dedicated blog posts in the coming weeks:

- We released DuckLake 1.1, including a new specification and reference implementation.
- The Iceberg extension strives to become a feature-complete implementation. To this end, it adds REST catalog ATTACH options that derive a table's location from its namespace or load table metadata from the `metadata-location` file returned by the catalog. Further options control the URL prefix and the namespace separator.
- The Delta extension can create Delta tables with `CREATE TABLE` (including partitioning and Delta table properties), supports time travel and improves `VARIANT` support.

SQL Features

Triggers

DuckDB 2.0 delivers triggers: the `CREATE TRIGGER` statement with `BEFORE` and `AFTER` triggers, `FOR EACH ROW` and `FOR EACH STATEMENT`, transition tables via `REFERENCING OLD/NEW TABLE`, multiple triggers per event, `RETURNING` on triggered tables and `DROP TRIGGER`. The classic use case is audit tables, where a trigger records what changed in response to a table modification. Some combinations, such as `BEFORE ... FOR EACH ROW` and `UPDATE ... FOR EACH ROW` triggers, are not yet supported; see the limitations.

```
CREATE TRIGGER trg_audit
    AFTER UPDATE ON target
    REFERENCING
        OLD TABLE AS o
        NEW TABLE AS n
    FOR EACH STATEMENT
        INSERT INTO audit
        SELECT n.id, o.val, n.val
        FROM o JOIN n ON o.id = n.id;
```

Triggers can also maintain materialized views incrementally! For example, to keep track of the average age of people per nationality in a table `age_statistics`, you can use two triggers – one for insert and another for delete operations – and store the sum and the count alongside the average.

Triggers bring us one step closer to materialized views, a long-standing item in the "future work" section of our roadmap.

SQL Dialect Additions

As always, DuckDB's friendly SQL dialect keeps growing. A few favorites from this release cycle:

- The $<variable>{:.language-sql .highlight} shorthand lets you write `$x` in most places where an expression is allowed, instead of `getvariable('x')`{:.language-sql .highlight}.
- Data-modifying CTEs let you use `INSERT`, `UPDATE`, `DELETE` and `COPY` as pipeline steps.
- `NEAREST` joins turn top-k similarity search into a join clause, handy for vector and embedding workloads.
- Nested schemas allow schemas within schemas, e.g., `CREATE SCHEMA finance.reports`.
- JSON mutation functions `json_set`, `json_insert`, `json_replace` and `json_remove` return modified copies of JSON documents (see also our blog post on JSON patches).

There is more: DuckDB 2.0.0 supports the SQL-standard `FETCH FIRST` and allows `UNNEST` in the `GROUP BY` clause. It also ensures well-defined `MERGE` and `UPDATE ... FROM` semantics for cases when multiple rows are matched.

Migrating to 2.0

DuckDB 2.0 comes with a small set of breaking changes. For most users, upgrading is straightforward, but complex deployments running many DuckDB processes need more care. The storage format is the most likely source of problems: DuckDB versions before 2.0 cannot read database files created by DuckDB 2.0. DuckDB 2.0 can still read and write your existing database files, and these remain readable by older versions. We recommend testing 2.0.0 on a shadow system before switching production over. And, if your workload is mission-critical, you may want to wait for the first patch release, coming in November.

For the full details, see the migration guide.

Internals and Deprecations

Compiler Requirements

Following DuckDB's long stint on C++11, DuckDB 2.0 now requires a C++17 compiler.

We also switched to Clang as the default compiler for our release builds. On Linux and macOS, both DuckDB and its core extensions are built with Clang. On Windows, the CLI is built with Clang, while most core extensions are still built with MSVC. We may move more Windows builds to Clang in the future, but this requires further changes.

Timezones, Calendars and Collations Without ICU

Timezone-aware timestamps, calendars and collations have always been powered by the ICU library. In 2.0, the ICU library is gone entirely: the `icu` extension now implements timezones, calendars and collations itself, with the timezone data built directly from the IANA database and compressed down to around 45 kB.

Everything keeps working exactly as before, and the native implementation is also faster. If you find any issues, please let us know!

Deprecations

Lambda Syntax The old single-arrow lambda syntax (`x -> x + 1`) is now disabled by default. Use the Python-style syntax instead:

```
SELECT list_transform([1, 2, 3], lambda x: x + 1);
```

You can temporarily re-enable the old syntax with `SET lambda_syntax = 'enable_single_arrow';`{:.language-sql .highlight}. Both the flag and the single-arrow syntax will be removed entirely in v2.1.

Amalgamation Builds The amalgamation build, which bundles DuckDB's entire source code into a single header and a single source file, is now deprecated. The amalgamation has grown too large to be realistically useful in most cases.

Gradual Phase-Out of macOS x86_64 We are gradually phasing out `osx_amd64` from the list of fully supported platforms. For the time being, DuckDB and core extensions are still built and supported, but community extensions are no longer built for this platform. DuckDB 2.1 will stop supporting `osx_amd64`.

Summary

These were a few highlights, but there are many more features and improvements in this release. There have been almost 20,000 commits by many contributors since DuckDB 1.5, shipping smaller features as well as performance and usability improvements. The full release notes can be found on GitHub.

We would like to thank our community for the detailed issue reports and feedback that have driven this release, and our special thanks go to our contributors!

Acknowledgments

This document is built with Pandoc using the Eisvogel template. The scripts to build the document are available in the DuckDB-Web repository.

The emojis used in this document are provided by Twemoji under the CC-BY 4.0 license.

The syntax highlighter uses the Bluloco Light theme by Umut Topuzoğlu.