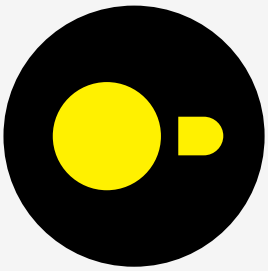
DuckDB Documentation

DuckDB version 0.9.1
Generated on 2023-11-13 at 12:42 UTC



Contents

Contents i

Summary 1

Documentation 3

Connect 5

Data Import 7

Importing Data . . . 7
CSV Files . . . 8
CSV Import . . . 8
CSV Auto Detection . . . 14
CSV Import Tips . . . 18
JSON Files . . . 19
JSON Loading . . . 19
Multiple Files . . . 26
Reading Multiple Files . . . 26
Combining Schemas . . . 29
Parquet Files . . . 31
Reading and Writing Parquet Files . . . 31
Querying Parquet Metadata . . . 35
Parquet Tips . . . 37
Partitioning . . . 38
Hive Partitioning . . . 38
Partitioned Writes . . . 40
Appender . . . 41
Insert Statements . . . 42

Client APIs 45

Client APIs Overview . . . 45

C . . . 45
C API - Overview . . . 45
C API - Startup & Shutdown . . . 46
C API - Configuration . . . 51
C API - Query . . . 54
C API - Data Chunks . . . 64
C API - Values . . . 76
C API - Types . . . 79
C API - Prepared Statements . . . 109
C API - Appender . . . 125
C API - Table Functions . . . 135
C API - Replacement Scans . . . 152
C API - Complete API . . . 155
C++ API . . . 273
CLI API . . . 279
Java JDBC API . . . 293
Julia Package . . . 297
Node.js . . . 298
Node.js API . . . 298
NodeJS API . . . 301
Python . . . 320
Python API . . . 320
Data Ingestion . . . 323
Result Conversion . . . 328
Python DB API . . . 330
Relational API . . . 333
Python Function API . . . 340
Types API . . . 344
Expression API . . . 348
Spark API . . . 352
Python Client API . . . 353
Known Python Issues . . . 353
R API . . . 354
Rust API . . . 358
Scala JDBC API . . . 359
Swift API . . . 361
Wasm . . . 361
DuckDB Wasm . . . 361

Instantiation . . . 362
Data Ingestion . . . 364
Query . . . 367
Extensions . . . 369
ADBC API . . . 372
ODBC . . . 380
ODBC API - Overview . . . 380
ODBC API - Linux . . . 381
ODBC API - Windows . . . 384
ODBC API - MacOS . . . 387

SQL **391**
SQL Introduction . . . 391
Statements . . . 401
Statements Overview . . . 401
Alter Table . . . 401
Alter View . . . 404
Attach/Detach . . . 405
Call . . . 408
Checkpoint . . . 408
Copy . . . 409
Create Macro . . . 415
Create Schema . . . 417
Create Sequence . . . 417
Create Table . . . 420
Create View . . . 424
Create Type . . . 425
Delete Statement . . . 426
Drop Statement . . . 426
Export & Import Database . . . 427
Insert Statement . . . 428
Pivot Statement . . . 431
Select Statement . . . 440
Set/Reset . . . 443
Unpivot Statement . . . 444
Update Statement . . . 452
Use . . . 454
Vacuum . . . 455

Query Syntax . . . 455
SELECT Clause . . . 455
FROM & JOIN Clauses . . . 458
WHERE Clause . . . 464
GROUP BY Clause . . . 464
GROUPING SETS . . . 466
HAVING Clause . . . 468
ORDER BY Clause . . . 469
LIMIT Clause . . . 471
SAMPLE Clause . . . 472
UNNEST . . . 473
WITH Clause . . . 474
WINDOW Clause . . . 481
QUALIFY Clause . . . 482
VALUES Clause . . . 484
FILTER Clause . . . 484
Set Operations . . . 488
Data Types . . . 490
Data Types . . . 490
Bitstring Type . . . 493
Blob Type . . . 493
Boolean Type . . . 494
Date Types . . . 495
Enum Types . . . 497
Interval Type . . . 500
List . . . 502
Map . . . 504
NULL Values . . . 506
Numeric Types . . . 507
Struct . . . 510
Text Types . . . 514
Time Types . . . 516
Timestamp Types . . . 517
Time Zones . . . 520
Union . . . 544
Expressions . . . 547
Expressions . . . 547
Case Statement . . . 547

Casting . . . 548
Collations . . . 549
Comparisons . . . 552
IN Operator . . . 553
Logical Operators . . . 554
Star Expression . . . 554
Subqueries . . . 557
Functions . . . 561
Functions . . . 561
Bitstring Functions . . . 561
Blob Functions . . . 564
Date Format . . . 564
Date Functions . . . 567
Date Parts . . . 571
Enum Functions . . . 575
Interval Functions . . . 576
Nested Functions . . . 578
Numeric Functions . . . 596
Pattern Matching . . . 600
Text Functions . . . 607
Time Functions . . . 620
Timestamp Functions . . . 622
Timestamp with Time Zone Functions . . . 629
Utility Functions . . . 639
Aggregate Functions . . . 642
Configuration . . . 649
Constraints . . . 656
Indexes . . . 658
Information Schema . . . 661
DuckDB_% Metadata Functions . . . 665
Pragmas . . . 680
Rules for Case Sensitivity . . . 686
Samples . . . 687
Window Functions . . . 690

Extensions **699**
Extensions . . . 699
Official Extensions . . . 701

Working with Extensions . . . 703
Arrow Extension . . . 704
AutoComplete Extension . . . 704
AWS Extension . . . 706
Azure Extension . . . 708
Excel Extension . . . 708
Full Text Search Extension . . . 709
httpfs Extension . . . 713
Iceberg Extension . . . 718
ICU Extension . . . 720
inet Extension . . . 720
jemalloc Extension . . . 721
JSON Extension . . . 721
MySQL Scanner Extension . . . 738
PostgreSQL Scanner Extension . . . 742
Spatial Extension . . . 743
SQLite Scanner Extension . . . 757
Substrait Extension . . . 761
TPC-DS Extension . . . 764
TPC-H Extension . . . 765

Guides **767**

Data Import & Export **769**
CSV Import . . . 769
CSV Export . . . 769
Parquet Import . . . 770
Parquet Export . . . 770
Parquet Import . . . 770
HTTP Parquet Import . . . 771
S3, GCS, or R2 Parquet Import . . . 771
S3 Parquet Export . . . 772
JSON Import . . . 773
JSON Export . . . 773
Excel Import . . . 774
Excel Export . . . 775
SQLite Import . . . 775

PostgreSQL Import 776

Meta Queries **777**
List Tables 777
Describe 778
Summarize 779
Explain 780
Profile Queries 782

ODBC **785**
ODBC 101: A Duck Themed Guide to ODBC 785

Python **795**
Install the Python Client 795
Execute SQL 795
Jupyter Notebooks 796
SQL on Pandas 801
Import from Pandas 802
Export to Pandas 802
SQL on Apache Arrow 802
Import from Apache Arrow 805
Export to Apache Arrow 806
Relational API and Pandas 807
Multiple Python Threads 808
DuckDB with Ibis 811
DuckDB with Polars 826
DuckDB with Vaex 827
DuckDB with DataFusion 829
Filesystems 831

SQL Features **833**
DuckDB ASOF Join 833
DuckDB Full Text Search 835

SQL Editors **839**
DBeaver SQL IDE 839

Data Viewers **841**
Tableau - A Data Visualisation Tool 841
CLI Charting - Using DuckDB with CLI Tools 846

Under the Hood **851**

Internals **853**
Overview of DuckDB Internals . . . 853
Storage . . . 855
Execution Format . . . 857

Developer Guides **861**
Building DuckDB from Source . . . 861
Profiling . . . 866
Testing . . . 870
SQLLogicTest . . . 871
SQLLogicTest - Debugging . . . 873
SQLLogicTest - Result Verification . . . 875
SQLLogicTest - Persistent Testing . . . 879
SQLLogicTest - Loops . . . 880
SQLLogicTest - Multiple Connections . . . 882
Catch C/C++ Tests . . . 883

Acknowledgments **885**

Summary

This document contains DuckDB's official documentation and guides 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. If you are considering submitting a contribution to the documentation, please consult our contributor guide.

Code repositories:

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

Documentation

Connect

Connect or Create a Database

To use DuckDB, you must first create a connection to a database. The exact process varies by client. Most clients take a parameter pointing to a database file to read and write from (the file extension may be anything, e.g., `.db`, `.duckdb`, etc.). If the database file does not exist, it will be created. The special value `:memory:` can be used to create an in-memory database where no data is persisted to disk (i.e., all data is lost when you exit the process).

See the API docs for client-specific details.

Data Import

Importing Data

The first step to using a database system is to insert data into that system. DuckDB provides several data ingestion methods that allow you to easily and efficiently fill up the database. In this section, we provide an overview of these methods so you can select which one is correct for you.

Insert Statements

Insert statements are the standard way of loading data into a database system. They are suitable for quick prototyping, but should be avoided for bulk loading as they have significant per-row overhead.

```
INSERT INTO people VALUES (1, 'Mark');
```

See here for a more detailed description of insert statements.

CSV Loading

Data can be efficiently loaded from CSV files using the `read_csv_auto` function or the COPY statement.

```
SELECT * FROM read_csv_auto('test.csv');
```

You can also load data from **compressed** (e.g., compressed with gzip) CSV files, for example:

```
SELECT * FROM read_csv_auto('test.csv.gz');
```

See here for a detailed description of CSV loading.

Parquet Loading

Parquet files can be efficiently loaded and queried using the `read_parquet` function.

```
SELECT * FROM read_parquet('test.parquet');
```

See here for a detailed description of Parquet loading.

JSON Loading

JSON files can be efficiently loaded and queried using the `read_json_auto` function.

```
SELECT * FROM read_json_auto('test.json');
```

See here for a detailed description of JSON loading.

Appender (C++ and Java)

In C++ and Java, the appender can be used as an alternative for bulk data loading. This class can be used to efficiently add rows to the database system without needing to use SQL.

C++:

```
Appender appender(con, "people");
appender.AppendRow(1, "Mark");
appender.Close();
```

Java:

```
con.createAppender("main", "people");
appender.beginRow();
appender.append("Mark");
appender.endRow();
appender.close();
```

See here for a detailed description of the C++ appender.

CSV Files

CSV Import

Examples

```
-- read a CSV file from disk, auto-infer options
SELECT * FROM 'flights.csv';
-- read_csv with custom options
```

```
SELECT * FROM read_csv('flights.csv', delim='|', header=true,
 ↪  columns={'FlightDate': 'DATE', 'UniqueCarrier': 'VARCHAR',
 ↪  'OriginCityName': 'VARCHAR', 'DestCityName': 'VARCHAR'});
-- read a CSV from stdin, auto-infer options
cat data/csv/issue2471.csv | duckdb -c "SELECT * FROM read_csv_
 ↪  auto('/dev/stdin')"

-- read a CSV file into a table
CREATE TABLE ontime(FlightDate DATE, UniqueCarrier VARCHAR, OriginCityName
 ↪  VARCHAR, DestCityName VARCHAR);
COPY ontime FROM 'flights.csv' (AUTO_DETECT true);
-- alternatively, create a table without specifying the schema manually
CREATE TABLE ontime AS SELECT * FROM 'flights.csv';
-- we can use the FROM-first syntax to omit 'SELECT *'
CREATE TABLE ontime AS FROM 'flights.csv';

-- write the result of a query to a CSV file
COPY (SELECT * FROM ontime) TO 'flights.csv' WITH (HEADER 1, DELIMITER '|');
-- we can use the FROM-first syntax to omit 'SELECT *'
COPY (FROM ontime) TO 'flights.csv' WITH (HEADER 1, DELIMITER '|');
```

CSV Loading

CSV loading, i.e., importing CSV files to the database, is a very common, and yet surprisingly tricky, task. While CSVs seem simple on the surface, there are a lot of inconsistencies found within CSV files that can make loading them a challenge. CSV files come in many different varieties, are often corrupt, and do not have a schema. The CSV reader needs to cope with all of these different situations.

The DuckDB CSV reader can automatically infer which configuration flags to use by analyzing the CSV file. This will work correctly in most situations, and should be the first option attempted. In rare situations where the CSV reader cannot figure out the correct configuration it is possible to manually configure the CSV reader to correctly parse the CSV file. See the auto detection page for more information.

Parameters

Below are parameters that can be passed to the CSV reader. These parameters are accepted by both the COPY statement and the CSV reader functions (`read_csv` and `read_csv_auto`).

Name	Description	Type	Default
`all_varchar`	Option to skip type detection for CSV parsing and assume all columns to be of type VARCHAR.	`BOOL`	`false`
`auto_detect`	Enables auto detection of parameters.	`BOOL`	`true`
`buffer_size`	The buffer size used by the CSV reader, specified in bytes. By default, it is set to 32MB or the size of the CSV file (if smaller). The buffer size must be at least as large as the longest line in the CSV file. Note: this is an advanced option that has a significant impact on performance and memory usage.	`BIGINT`	min(32000000, CSV file size)
`columns`	A struct that specifies the column names and column types contained within the CSV file (e.g., `{'col1': 'INTEGER', 'col2': 'VARCHAR'}`). Using this option implies that auto detection is not used.	`STRUCT`	(empty)
`compression`	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `t.csv.gz` will use gzip, `t.csv` will use `none`). Options are `none`, `gzip`, `zstd`.	`VARCHAR`	`auto`
`dateformat`	Specifies the date format to use when parsing dates. See Date Format.	`VARCHAR`	(empty)
`decimal_separator`	The decimal separator of numbers.	`VARCHAR`	`.`
`delim` or `sep`	Specifies the string that separates columns within each row (line) of the file.	`VARCHAR`	`,`
`escape`	Specifies the string that should appear before a data character sequence that matches the `quote` value.	`VARCHAR`	`"`

Name	Description	Type	Default
`filename`	Whether or not an extra `filename` column should be included in the result.	`BOOL`	`false`
`force_not_null`	Do not match the specified columns' values against the NULL string. In the default case where the `NULL` string is empty, this means that empty values will be read as zero-length strings rather than `NULL`s.	`VARCHAR[]`	`[]`
`header`	Specifies that the file contains a header line with the names of each column in the file.	`BOOL`	`false`
`hive_partitioning`	Whether or not to interpret the path as a hive partitioned path.	`BOOL`	`false`
`ignore_errors`	Option to ignore any parsing errors encountered - and instead ignore rows with errors.	`BOOL`	`false`
`max_line_size`	The maximum line size in bytes.	`BIGINT`	2097152
`names`	The column names as a list, see example.	`VARCHAR[]`	(empty)
`new_line`	Set the new line character(s) in the file. Options are `'\r'`,`'\n'`, or `'\r\n'`.	`VARCHAR`	(empty)
`normalize_names`	Boolean value that specifies whether or not column names should be normalized, removing any non-alphanumeric characters from them.	`BOOL`	`false`
`null_padding`	If this option is enabled, when a row lacks columns, it will pad the remaining columns on the right with null values.	`BOOL`	`false`
`nullstr`	Specifies the string that represents a NULL value.	`VARCHAR`	(empty)
`parallel`	Whether or not the parallel CSV reader is used.	`BOOL`	`true`

Name	Description	Type	Default
`quote`	Specifies the quoting string to be used when a data value is quoted.	`VARCHAR`	`"`
`sample_size`	The number of sample rows for auto detection of parameters.	`BIGINT`	20480
`skip`	The number of lines at the top of the file to skip.	`BIGINT`	0
`timestampformat`	Specifies the date format to use when parsing timestamps. See Date Format	`VARCHAR`	(empty)
`types` or `dtypes`	The column types as either a list (by position) or a struct (by name). Example here.	`VARCHAR[]` or `STRUCT`	(empty)
`union_by_name`	Whether the columns of multiple schemas should be unified by name, rather than by position.	`BOOL`	`false`

read_csv_auto Function

The `read_csv_auto` is the simplest method of loading CSV files: it automatically attempts to figure out the correct configuration of the CSV reader. It also automatically deduces types of columns. If the CSV file has a header, it will use the names found in that header to name the columns. Otherwise, the columns will be named `column0, column1, column2, ...`. An example with the `flights.csv` file:

```
SELECT * FROM read_csv_auto('flights.csv');
```

FlightDate	UniqueCarrier	OriginCityName	DestCityName
1988-01-01	AA	New York, NY	Los Angeles, CA
1988-01-02	AA	New York, NY	Los Angeles, CA
1988-01-03	AA	New York, NY	Los Angeles, CA

The path can either be a relative path (relative to the current working directory) or an absolute path.

We can use `read_csv_auto` to create a persistent table as well:

```
CREATE TABLE ontime AS SELECT * FROM read_csv_auto('flights.csv');
DESCRIBE ontime;
```

Field	Type	Null	Key	Default	Extra
FlightDate	DATE	YES	NULL	NULL	NULL
UniqueCarrier	VARCHAR	YES	NULL	NULL	NULL
OriginCityName	VARCHAR	YES	NULL	NULL	NULL
DestCityName	VARCHAR	YES	NULL	NULL	NULL

```
SELECT * FROM read_csv_auto('flights.csv', SAMPLE_SIZE=20000);
```

If we set DELIM/SEP, QUOTE, ESCAPE, or HEADER explicitly, we can bypass the automatic detection of this particular parameter:

```
SELECT * FROM read_csv_auto('flights.csv', HEADER=true);
```

Multiple files can be read at once by providing a glob or a list of files. Refer to the multiple files section for more information.

read_csv Function

The `read_csv` function accepts the same parameters that `read_csv_auto` does but does not assume `AUTO_DETECT=true`.

Writing Using the COPY Statement

The COPY statement can be used to load data from a CSV file into a table. This statement has the same syntax as the one used in PostgreSQL. To load the data using the COPY statement, we must first create a table with the correct schema (which matches the order of the columns in the CSV file and uses types that fit the values in the CSV file). We then specify the CSV file to load from plus any configuration options separately.

```
CREATE TABLE ontime(flightdate DATE, uniquecarrier VARCHAR, origincityname
↪  VARCHAR, destcityname VARCHAR);
COPY ontime FROM 'flights.csv' (DELIMITER '|', HEADER);
SELECT * FROM ontime;
```

flightdate	uniquecarrier	origincityname	destcityname
1988-01-01	AA	New York, NY	Los Angeles, CA
1988-01-02	AA	New York, NY	Los Angeles, CA
1988-01-03	AA	New York, NY	Los Angeles, CA

If we want to use the automatic format detection, we can set `AUTO_DETECT` to `true` and omit the otherwise required configuration options.

```
CREATE TABLE ontime(flightdate DATE, uniquecarrier VARCHAR, origincityname
 ↪  VARCHAR, destcityname VARCHAR);
COPY ontime FROM 'flights.csv' (AUTO_DETECT true);
SELECT * FROM ontime;
```

CSV Auto Detection

When using `read_csv_auto`, or reading a CSV file with the `auto_detect` flag set, the system tries to automatically infer how to read the CSV file. This step is necessary because CSV files are not self-describing and come in many different dialects. The auto-detection works roughly as follows:

- Detect the dialect of the CSV file (delimiter, quoting rule, escape)
- Detect the types of each of the columns
- Detect whether or not the file has a header row

By default the system will try to auto-detect all options. However, options can be individually overridden by the user. This can be useful in case the system makes a mistake. For example, if the delimiter is chosen incorrectly, we can override it by calling the `read_csv_auto` with an explicit delimiter (e.g., `read_csv_auto('file.csv', delim='|')`).

The detection works by operating on a sample of the file. The size of the sample can be modified by setting the `sample_size` parameter. The default sample size is `20480` rows. Setting the `sample_size` parameter to `-1` means the entire file is read for sampling. The way sampling is performed depends on the type of file. If we are reading from a regular file on disk, we will jump into the file and try to sample from different locations in the file. If we are reading from a file in which we cannot jump - such as a `.gz` compressed CSV file or `stdin` - samples are taken only from the beginning of the file.

Dialect Detection

Dialect detection works by attempting to parse the samples using the set of considered values. The detected dialect is the dialect that has (1) a consistent number of columns for each row, and (2) the highest number of columns for each row.

The following dialects are considered for automatic dialect detection.

Parameters	Considered values
`delim`	`,` `|` `;` `\t`
`quote`	`"` `'` (empty)
`escape`	`"` `'` `\` (empty)

Consider the example file `flights.csv`:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-01|AA|New York, NY|Los Angeles, CA
1988-01-02|AA|New York, NY|Los Angeles, CA
1988-01-03|AA|New York, NY|Los Angeles, CA
```

In this file, the dialect detection works as follows:

- If we split by a `|` every row is split into 4 columns
- If we split by a `,` rows 2-4 are split into 3 columns, while the first row is split into 1 column
- If we split by `;`, every row is split into 1 column
- If we split by `\t`, every row is split into 1 column

In this example - the system selects the `|` as the delimiter. All rows are split into the same amount of columns, and there is more than one column per row meaning the delimiter was actually found in the CSV file.

Type Detection

After detecting the dialect, the system will attempt to figure out the types of each of the columns. Note that this step is only performed if we are calling `read_csv_auto`. In case of the COPY statement the types of the table that we are copying into will be used instead.

The type detection works by attempting to convert the values in each column to the candidate types. If the conversion is unsuccessful, the candidate type is removed from the set of candidate types for

that column. After all samples have been handled - the remaining candidate type with the highest priority is chosen. The set of considered candidate types in order of priority is given below:

Types
`BOOLEAN`
`BIGINT`
`DOUBLE`
`TIME`
`DATE`
`TIMESTAMP`
`VARCHAR`

Note everything can be cast to VARCHAR. This type has the lowest priority - i.e., columns are converted to VARCHAR if they cannot be cast to anything else. In `flights.csv` the `FlightDate` column will be cast to a DATE, while the other columns will be cast to VARCHAR.

The detected types can be individually overridden using the `types` option. This option takes either a list of types (e.g., `types=[INT, VARCHAR, DATE]`) which overrides the types of the columns in-order of occurrence in the CSV file. Alternatively, `types` takes a `name -> type` map which overrides options of individual columns (e.g., `types={'quarter': INT}`).

The type detection can be entirely disabled by using the `all_varchar` option. If this is set all columns will remain as VARCHAR (as they originally occur in the CSV file).

Header Detection

Header detection works by checking if the candidate header row deviates from the other rows in the file in terms of types. For example, in `flights.csv`, we can see that the header row consists of only VARCHAR columns - whereas the values contain a DATE value for the `FlightDate` column. As such - the system defines the first row as the header row and extracts the column names from the header row.

In files that do not have a header row, the column names are generated as `column0`, `column1`, etc.

Note that headers cannot be detected correctly if all columns are of type VARCHAR - as in this case the system cannot distinguish the header row from the other rows in the file. In this case the system assumes the file has no header. This can be overridden using the `header` option.

Dates and Timestamps

DuckDB supports the ISO 8601 format format by default for timestamps, dates and times. Unfortunately, not all dates and times are formatted using this standard. For that reason, the CSV reader also supports the `dateformat` and `timestampformat` options. Using this format the user can specify a format string that specifies how the date or timestamp should be read.

As part of the auto-detection, the system tries to figure out if dates and times are stored in a different representation. This is not always possible - as there are ambiguities in the representation. For example, the date `01-02-2000` can be parsed as either January 2nd or February 1st. Often these ambiguities can be resolved. For example, if we later encounter the date `21-02-2000` then we know that the format must have been `DD-MM-YYYY`. `MM-DD-YYYY` is no longer possible as there is no 21nd month.

If the ambiguities cannot be resolved by looking at the data the system has a list of preferences for which date format to use. If the system choses incorrectly, the user can specify the `dateformat` and `timestampformat` options manually.

The system considers the following formats for dates (`dateformat`). Higher entries are chosen over lower entries in case of ambiguities (i.e., ISO 8601 is preferred over `MM-DD-YYYY`).

dateformat
`ISO 8601`
`%y-%m-%d`
`%Y-%m-%d`
`%d-%m-%y`
`%d-%m-%Y`
`%m-%d-%y`
`%m-%d-%Y`

The system considers the following formats for timestamps (`timestampformat`). Higher entries are chosen over lower entries in case of ambiguities.

timestampformat
`ISO 8601`

timestampformat
%y-%m-%d %H:%M:%S
%Y-%m-%d %H:%M:%S
%d-%m-%y %H:%M:%S
%d-%m-%Y %H:%M:%S
%m-%d-%y %I:%M:%S %p
%m-%d-%Y %I:%M:%S %p
%Y-%m-%d %H:%M:%S.%f

CSV Import Tips

Below is a collection of tips to help when attempting to import complex CSV files. In the examples, we use the `flights.csv` file.

Override the Header Flag if the Header Is Not Correctly Detected If a file contains only string columns the `header` auto-detection might fail. Provide the `header` option to override this behavior.

```
SELECT * FROM read_csv_auto('flights.csv', header=true);
```

Provide Names if the File Does Not Contain a Header If the file does not contain a header, names will be auto-generated by default. You can provide your own names with the `names` option.

```
SELECT * FROM read_csv_auto('flights.csv', names=['DateOfFlight',
↪  'CarrierName']);
```

Override the Types of Specific Columns The `types` flag can be used to override types of only certain columns by providing a struct of `name -> type` mappings.

```
SELECT * FROM read_csv_auto('flights.csv', types={'FlightDate': 'DATE'});
```

Use COPY When Loading Data into a Table The COPY statement copies data directly into a table. The CSV reader uses the schema of the table instead of auto-detecting types from the file. This speeds up the auto-detection, and prevents mistakes from being made during auto-detection.

```
COPY tbl FROM 'test.csv' (AUTO_DETECT 1);
```

Use `union_by_name` When Loading Files with Different Schemas The `union_by_name` option can be used to unify the schema of files that have different or missing columns. For files that do not have certain columns, `NULL` values are filled in.

```
SELECT * FROM read_csv_auto('flights*.csv', union_by_name=true);
```

JSON Files

JSON Loading

Examples

```
-- read a JSON file from disk, auto-infer options
SELECT * FROM 'todos.json';
-- read_json with custom options
SELECT *
FROM read_json('todos.json',
               format='array',
               columns={userId: 'UBIGINT',
                        id: 'UBIGINT',
                        title: 'VARCHAR',
                        completed: 'BOOLEAN'});
-- read a JSON file from stdin, auto-infer options
cat data/json/todos.json | duckdb -c "SELECT * FROM read_json_
 ↪  auto('/dev/stdin')"

-- read a JSON file into a table
CREATE TABLE todos(userId UBIGINT, id UBIGINT, title VARCHAR, completed
 ↪  BOOLEAN);
COPY todos FROM 'todos.json';
-- alternatively, create a table without specifying the schema manually
CREATE TABLE todos AS SELECT * FROM 'todos.json';

-- write the result of a query to a JSON file
COPY (SELECT * FROM todos) TO 'todos.json';
```

JSON Loading

JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable

values). While it is not a very efficient format for tabular data, it is very commonly used, especially as a data interchange format.

The DuckDB JSON reader can automatically infer which configuration flags to use by analyzing the JSON file. This will work correctly in most situations, and should be the first option attempted. In rare situations where the JSON reader cannot figure out the correct configuration, it is possible to manually configure the JSON reader to correctly parse the JSON file.

Below are parameters that can be passed in to the JSON reader.

Parameters

Name	Description	Type	Default
`maximum_object_size`	The maximum size of a JSON object (in bytes)	`UINTEGER`	`16777216`
`format`	Can be one of `['auto', 'unstructured', 'newline_delimited', 'array']`	`VARCHAR`	`'array'`
`ignore_errors`	Whether to ignore parse errors (only possible when `format` is `'newline_delimited'`)	`BOOL`	`false`
`compression`	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `t.json.gz` will use gzip, `t.json` will use none). Options are `'none'`, `'gzip'`, `'zstd'`, and `'auto'`.	`VARCHAR`	`'auto'`
`columns`	A struct that specifies the key names and value types contained within the JSON file (e.g., `{key1: 'INTEGER', key2: 'VARCHAR'}`). If `auto_detect` is enabled these will be inferred	`STRUCT`	`(empty)`
`records`	Can be one of `['auto', 'true', 'false']`	`VARCHAR`	`'records'`

Name	Description	Type	Default
`auto_detect`	Whether to auto-detect detect the names of the keys and data types of the values automatically	`BOOL`	`false`
`sample_size`	Option to define number of sample objects for automatic JSON type detection. Set to -1 to scan the entire input file	`UBIGINT`	`20480`
`maximum_depth`	Maximum nesting depth to which the automatic schema detection detects types. Set to -1 to fully detect nested JSON types	`BIGINT`	`-1`
`dateformat`	Specifies the date format to use when parsing dates. See Date Format	`VARCHAR`	`'iso'`
`timestampformat`	Specifies the date format to use when parsing timestamps. See Date Format	`VARCHAR`	`'iso'`
`filename`	Whether or not an extra `filename` column should be included in the result.	`BOOL`	`false`
`hive_partitioning`	Whether or not to interpret the path as a hive partitioned path.	`BOOL`	`false`
`union_by_name`	Whether the schema's of multiple JSON files should be unified.	`BOOL`	`false`

When using `read_json_auto`, every parameter that supports auto-detection is enabled.

Examples of Format Settings

The JSON extension can attempt to determine the format of a JSON file when setting `format` to `auto`.
Here are some example JSON files and the corresponding `format` settings that should be used.

In each of the below cases, the `format` setting was not needed, as DuckDB was able to infer it correctly, but it is included for illustrative purposes. A query of this shape would work in each case:

```
SELECT * FROM filename.json;
```

Format: newline_delimited With `format='newline_delimited'` newline-delimited JSON can be parsed. Each line is a JSON.

```
{"key1":"value1", "key2": "value1"}
{"key1":"value2", "key2": "value2"}
{"key1":"value3", "key2": "value3"}
```

```
SELECT * FROM read_json_auto('records.json', format='newline_delimited');
```

key1	key2
value1	value1
value2	value2
value3	value3

Format: array If the JSON file contains a JSON array of objects (pretty-printed or not), `array_of_objects` may be used.

```
[
    {"key1":"value1", "key2": "value1"},
    {"key1":"value2", "key2": "value2"},
    {"key1":"value3", "key2": "value3"}
]
```

```
SELECT * FROM read_json_auto('array.json', format='array');
```

key1	key2
value1	value1
value2	value2
value3	value3

Format: unstructured If the JSON file contains JSON that is not newline-delimited or an array, `unstructured` may be used.

```
{
    "key1":"value1",
    "key2": "value1"
}
```

```
{
    "key1":"value2",
    "key2": "value2"
}
{
    "key1":"value3",
    "key2": "value3"
}
```

```
SELECT * FROM read_json_auto('unstructured.json', format='unstructured');
```

key1	key2
value1	value1
value2	value2
value3	value3

Examples of Records Settings

The JSON extension can attempt to determine whether a JSON file contains records when setting `records=auto`. When `records=true`, the JSON extension expects JSON objects, and will unpack the fields of JSON objects into individual columns.

Continuing with the same example file from before:

```
{"key1":"value1", "key2": "value1"}
{"key1":"value2", "key2": "value2"}
{"key1":"value3", "key2": "value3"}
```

```
SELECT * FROM read_json_auto('records.json', records=true);
```

key1	key2
value1	value1
value2	value2
value3	value3

When `records=false`, the JSON extension will not unpack the top-level objects, and create STRUCTs instead:

```
SELECT * FROM read_json_auto('records.json', records=false);
```

json
{'key1': value1, 'key2': value1}
{'key1': value2, 'key2': value2}
{'key1': value3, 'key2': value3}

This is especially useful if we have non-object JSON, for example:

```
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

```
SELECT * FROM read_json_auto('arrays.json', records=false);
```

json
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]

Writing

The contents of tables or the result of queries can be written directly to a JSON file using the COPY statement. See the COPY documentation for more information.

read_json_auto Function

The `read_json_auto` is the simplest method of loading JSON files: it automatically attempts to figure out the correct configuration of the JSON reader. It also automatically deduces types of columns.

```
SELECT * FROM read_json_auto('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

The path can either be a relative path (relative to the current working directory) or an absolute path.

We can use `read_json_auto` to create a persistent table as well:

```
CREATE TABLE todos AS SELECT * FROM read_json_auto('todos.json');
DESCRIBE todos;
```

column_name	column_type	null	key	default	extra
userId	UBIGINT	YES			
id	UBIGINT	YES			
title	VARCHAR	YES			
completed	BOOLEAN	YES			

If we specify the columns, we can bypass the automatic detection. Note that not all columns need to be specified:

```
SELECT *
FROM read_json_auto('todos.json',
                    columns={userId: 'UBIGINT',
                             completed: 'BOOLEAN'});
```

Multiple files can be read at once by providing a glob or a list of files. Refer to the multiple files section for more information.

COPY Statement

The COPY statement can be used to load data from a JSON file into a table. For the COPY statement, we must first create a table with the correct schema to load the data into. We then specify the JSON file to load from plus any configuration options separately.

```
CREATE TABLE todos(userId UBIGINT, id UBIGINT, title VARCHAR, completed
↪  BOOLEAN);
COPY todos FROM 'todos.json';
SELECT * FROM todos 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

More on the COPY statement can be found here.

Multiple Files

Reading Multiple Files

DuckDB can read multiple files of different types (CSV, Parquet, JSON files) at the same time using either the glob syntax, or by providing a list of files to read. See the combining schemas page for tips on reading files with different schemas.

CSV

```
-- read all files with a name ending in ".csv" in the folder "dir"
SELECT * FROM 'dir/*.csv';
-- read all files with a name ending in ".csv", two directories deep
SELECT * FROM '*/*/*.csv';
-- read all files with a name ending in ".csv", at any depth in the folder
↪  "dir"
SELECT * FROM 'dir/**/*.csv';
-- read the CSV files 'flights1.csv' and 'flights2.csv'
SELECT * FROM read_csv_auto(['flights1.csv', 'flights2.csv']);
-- read the CSV files 'flights1.csv' and 'flights2.csv', unifying schemas by
↪  name and outputting a `filename` column
SELECT * FROM read_csv_auto(['flights1.csv', 'flights2.csv'], union_by_
↪  name=true, filename=true);
```

Parquet

```
-- read all files that match the glob pattern
SELECT * FROM 'test/*.parquet';
-- read 3 parquet files and treat them as a single table
SELECT * FROM read_parquet(['file1.parquet', 'file2.parquet',
 ↪  'file3.parquet']);
-- Read all parquet files from 2 specific folders
SELECT * FROM read_parquet(['folder1/*.parquet', 'folder2/*.parquet']);
-- read all parquet files that match the glob pattern at any depth
SELECT * FROM read_parquet('dir/**/*.parquet');
```

Multi-File Reads and Globs

DuckDB can also read a series of Parquet files and treat them as if they were a single table. Note that this only works if the Parquet files have the same schema. You can specify which Parquet files you want to read using a list parameter, glob pattern matching syntax, or a combination of both.

List Parameter The read_parquet function can accept a list of filenames as the input parameter.

```
-- read 3 parquet files and treat them as a single table
SELECT * FROM read_parquet(['file1.parquet', 'file2.parquet',
 ↪  'file3.parquet']);
```

Glob Syntax Any file name input to the read_parquet function can either be an exact filename, or use a glob syntax to read multiple files that match a pattern.

Wildcard	Description
`*`	matches any number of any characters (including none)
`**`	matches any number of subdirectories (including none)
`?`	matches any single character
`[abc]`	matches one character given in the bracket
`[a-z]`	matches one character from the range given in the bracket

Note that the `?` wildcard in globs is not supported for reads over S3 due to HTTP encoding issues.

Here is an example that reads all the files that end with `.parquet` located in the `test` folder:

```
-- read all files that match the glob pattern
SELECT * FROM read_parquet('test/*.parquet');
```

List of Globs The glob syntax and the list input parameter can be combined to scan files that meet one of multiple patterns.

```
-- Read all parquet files from 2 specific folders
SELECT * FROM read_parquet(['folder1/*.parquet', 'folder2/*.parquet']);
```

DuckDB can read multiple CSV files at the same time using either the glob syntax, or by providing a list of files to read.

Filename

The `filename` argument can be used to add an extra `filename` column to the result that indicates which row came from which file. For example:

```
SELECT * FROM read_csv_auto(['flights1.csv', 'flights2.csv'], union_by_
↪  name=true, filename=true);
```

FlightDate	OriginCityName	DestCityName	UniqueCarrier	filename
1988-01-01	New York, NY	Los Angeles, CA	NULL	flights1.csv
1988-01-02	New York, NY	Los Angeles, CA	NULL	flights1.csv
1988-01-03	New York, NY	Los Angeles, CA	AA	flights2.csv

Glob Function to Find Filenames

The glob pattern matching syntax can also be used to search for filenames using the `glob` table function. It accepts one parameter: the path to search (which may include glob patterns).

```
-- Search the current directory for all files
SELECT * FROM glob('*');
```

file
duckdb.exe
test.csv

file
test.json
test.parquet
test2.csv
test2.parquet
todos.json

Combining Schemas

Examples

```
-- read a set of CSV files combining columns by position
SELECT * FROM read_csv_auto('flights*.csv');
-- read a set of CSV files combining columns by name
SELECT * FROM read_csv_auto('flights*.csv', union_by_name=true);
```

Combining Schemas

When reading from multiple files, we have to **combine schemas** from those files. That is because each file has its own schema that can differ from the other files. DuckDB offers two ways of unifying schemas of multiple files: **by column position** and **by column name**.

By default, DuckDB reads the schema of the first file provided, and then unifies columns in subsequent files by column position. This works correctly as long as all files have the same schema. If the schema of the files differs, you might want to use the `union_by_name` option to allow DuckDB to construct the schema by reading all of the names instead.

Below is an example of how both methods work.

Union By Position

By default, DuckDB unifies the columns of these different files **by position**. This means that the first column in each file is combined together, as well as the second column in each file, etc. For example, consider the following two files.

`flights1.csv`:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-01|AA|New York, NY|Los Angeles, CA
1988-01-02|AA|New York, NY|Los Angeles, CA
```

flights2.csv:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-03|AA|New York, NY|Los Angeles, CA
```

Reading the two files at the same time will produce the following result set:

FlightDate	UniqueCarrier	OriginCityName	DestCityName
1988-01-01	AA	New York, NY	Los Angeles, CA
1988-01-02	AA	New York, NY	Los Angeles, CA
1988-01-03	AA	New York, NY	Los Angeles, CA

This is equivalent to the SQL construct `UNION ALL`.

Union By Name

If you are processing multiple files that have different schemas, perhaps because columns have been added or renamed, it might be desirable to unify the columns of different files **by name** instead. This can be done by providing the `union_by_name` option. For example, consider the following two files, where `flights4.csv` has an extra column (`UniqueCarrier`).

flights3.csv:

```
FlightDate|OriginCityName|DestCityName
1988-01-01|New York, NY|Los Angeles, CA
1988-01-02|New York, NY|Los Angeles, CA
```

flights4.csv:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-03|AA|New York, NY|Los Angeles, CA
```

Reading these when unifying column names **by position** results in an error - as the two files have a different number of columns. When specifying the `union_by_name` option, the columns are correctly unified, and any missing values are set to `NULL`.

```
SELECT * FROM read_csv_auto(['flights3.csv', 'flights4.csv'], union_by_
↪  name=true);
```

FlightDate	OriginCityName	DestCityName	UniqueCarrier
1988-01-01	New York, NY	Los Angeles, CA	NULL
1988-01-02	New York, NY	Los Angeles, CA	NULL
1988-01-03	New York, NY	Los Angeles, CA	AA

This is equivalent to the SQL construct `UNION ALL BY NAME`.

Parquet Files

Reading and Writing Parquet Files

Examples

```
-- read a single parquet file
SELECT * FROM 'test.parquet';
-- figure out which columns/types are in a parquet file
DESCRIBE SELECT * FROM 'test.parquet';
-- create a table from a parquet file
CREATE TABLE test AS SELECT * FROM 'test.parquet';
-- if the file does not end in ".parquet", use the read_parquet function
SELECT * FROM read_parquet('test.parq');
-- use list parameter to read 3 parquet files and treat them as a single
 ↪  table
SELECT * FROM read_parquet(['file1.parquet', 'file2.parquet',
 ↪  'file3.parquet']);
-- read all files that match the glob pattern
SELECT * FROM 'test/*.parquet';
-- read all files that match the glob pattern, and include a "filename"
 ↪  column that specifies which file each row came from
SELECT * FROM read_parquet('test/*.parquet', filename=true);
-- use a list of globs to read all parquet files from 2 specific folders
SELECT * FROM read_parquet(['folder1/*.parquet', 'folder2/*.parquet']);
-- query the metadata of a parquet file
SELECT * FROM parquet_metadata('test.parquet');
-- query the schema of a parquet file
SELECT * FROM parquet_schema('test.parquet');

-- write the results of a query to a parquet file
```

```
COPY (SELECT * FROM tbl) TO 'result-snappy.parquet' (FORMAT 'parquet');
-- write the results from a query to a parquet file with specific
 ↪  compression and row_group_size
COPY (FROM generate_series(100000)) TO 'test.parquet' (FORMAT 'parquet',
 ↪  COMPRESSION 'ZSTD', ROW_GROUP_SIZE 100000);

-- export the table contents of the entire database as parquet
EXPORT DATABASE 'target_directory' (FORMAT PARQUET);
```

Parquet Files

Parquet files are compressed columnar files that are efficient to load and process. DuckDB provides support for both reading and writing Parquet files in an efficient manner, as well as support for pushing filters and projections into the Parquet file scans.

read_parquet Function

Function	Description	Example
`read_parquet(`*`path(s)`*`, *)`	Read Parquet file(s)	`SELECT * FROM read_parquet('test.parquet');`
`parquet_scan(`*`path(s)`*`, *)`	Alias for `read_parquet`	`SELECT * FROM parquet_scan('test.parquet');`

If your file ends in `.parquet`, the function syntax is optional. The system will automatically infer that you are reading a Parquet file.

```
SELECT * FROM 'test.parquet';
```

Multiple files can be read at once by providing a glob or a list of files. Refer to the multiple files section for more information.

Parameters There are a number of options exposed that can be passed to the `read_parquet` function or the COPY statement.

Name	Description	Type	Default
`binary_as_string`	Parquet files generated by legacy writers do not correctly set the UTF8 flag for strings, causing string columns to be loaded as BLOB instead. Set this to true to load binary columns as strings.	BOOL	`false`
`filename`	Whether or not an extra `filename` column should be included in the result.	BOOL	`false`
`file_row_number`	Whether or not to include the `file_row_number` column.	BOOL	`false`
`hive_partitioning`	Whether or not to interpret the path as a hive partitioned path.	BOOL	`false`
`union_by_name`	Whether the columns of multiple schemas should be unified by name, rather than by position.	BOOL	`false`

Partial Reading

DuckDB supports projection pushdown into the Parquet file itself. That is to say, when querying a Parquet file, only the columns required for the query are read. This allows you to read only the part of the Parquet file that you are interested in. This will be done automatically by DuckDB.

DuckDB also supports filter pushdown into the Parquet reader. When you apply a filter to a column that is scanned from a Parquet file, the filter will be pushed down into the scan, and can even be used to skip parts of the file using the built-in zonemaps. Note that this will depend on whether or not your Parquet file contains zonemaps.

Filter and projection pushdown provide significant performance benefits. See our blog post on this for more information.

Inserts and Views

You can also insert the data into a table or create a table from the parquet file directly. This will load the data from the parquet file and insert it into the database.

```
-- insert the data from the parquet file in the table
INSERT INTO people SELECT * FROM read_parquet('test.parquet');
```

```
-- create a table directly from a parquet file
CREATE TABLE people AS SELECT * FROM read_parquet('test.parquet');
```

If you wish to keep the data stored inside the parquet file, but want to query the parquet file directly, you can create a view over the `read_parquet` function. You can then query the parquet file as if it were a built-in table.

```
-- create a view over the parquet file
CREATE VIEW people AS SELECT * FROM read_parquet('test.parquet');
-- query the parquet file
SELECT * FROM people;
```

Writing to Parquet Files

DuckDB also has support for writing to Parquet files using the COPY statement syntax. See the COPY Statement page for details, including all possible parameters for the COPY statement.

```
-- write a query to a snappy compressed parquet file
COPY (SELECT * FROM tbl) TO 'result-snappy.parquet' (FORMAT 'parquet')
-- write "tbl" to a zstd compressed parquet file
COPY tbl TO 'result-zstd.parquet' (FORMAT 'PARQUET', CODEC 'ZSTD')
-- write a csv file to an uncompressed parquet file
COPY 'test.csv' TO 'result-uncompressed.parquet' (FORMAT 'PARQUET', CODEC
 ↪  'UNCOMPRESSED')
-- write a query to a parquet file with ZSTD compression (same as CODEC) and
 ↪  row_group_size
COPY (FROM generate_series(100000)) TO 'row-groups-zstd.parquet' (FORMAT
 ↪  PARQUET, COMPRESSION ZSTD, ROW_GROUP_SIZE 100000);
```

DuckDB's EXPORT command can be used to export an entire database to a series of Parquet files. See the Export statement documentation for more details.

```
-- export the table contents of the entire database as parquet
EXPORT DATABASE 'target_directory' (FORMAT PARQUET);
```

Installing and Loading the Parquet Extension

The support for Parquet files is enabled via extension. The `parquet` extension is bundled with almost all clients. However, if your client does not bundle the `parquet` extension, the extension must be installed and loaded separately.

```
-- run once
INSTALL parquet;
```

```
-- run before usage
LOAD parquet;
```

Querying Parquet Metadata

Parquet Metadata

The `parquet_metadata` function can be used to query the metadata contained within a Parquet file, which reveals various internal details of the Parquet file such as the statistics of the different columns. This can be useful for figuring out what kind of skipping is possible in Parquet files, or even to obtain a quick overview of what the different columns contain.

```
SELECT * FROM parquet_metadata('test.parquet');
```

Below is a table of the columns returned by `parquet_metadata`.

Field	Type
file_name	VARCHAR
row_group_id	BIGINT
row_group_num_rows	BIGINT
row_group_num_columns	BIGINT
row_group_bytes	BIGINT
column_id	BIGINT
file_offset	BIGINT
num_values	BIGINT
path_in_schema	VARCHAR
type	VARCHAR
stats_min	VARCHAR
stats_max	VARCHAR
stats_null_count	BIGINT
stats_distinct_count	BIGINT
stats_min_value	VARCHAR
stats_max_value	VARCHAR

Field	Type
compression	VARCHAR
encodings	VARCHAR
index_page_offset	BIGINT
dictionary_page_offset	BIGINT
data_page_offset	BIGINT
total_compressed_size	BIGINT
total_uncompressed_size	BIGINT

Parquet Schema

The `parquet_schema` function can be used to query the internal schema contained within a Parquet file. Note that this is the schema as it is contained within the metadata of the Parquet file. If you want to figure out the column names and types contained within a Parquet file it is easier to use `DESCRIBE`.

```
-- fetch the column names and column types
DESCRIBE SELECT * FROM 'test.parquet';
-- fetch the internal schema of a parquet file
SELECT * FROM parquet_schema('test.parquet');
```

Below is a table of the columns returned by `parquet_schema`.

Field	Type
file_name	VARCHAR
name	VARCHAR
type	VARCHAR
type_length	VARCHAR
repetition_type	VARCHAR
num_children	BIGINT
converted_type	VARCHAR
scale	BIGINT

Field	Type
precision	BIGINT
field_id	BIGINT
logical_type	VARCHAR

Parquet Tips

Below is a collection of tips to help when dealing with Parquet files.

Tips for reading Parquet files

Use `union_by_name` when loading files with different schemas The `union_by_name` option can be used to unify the schema of files that have different or missing columns. For files that do not have certain columns, `NULL` values are filled in.

```
SELECT * FROM read_parquet('flights*.parquet', union_by_name=true);
```

Tips for writing Parquet files

Enabling `per_thread_output` If the final number of parquet files is not important, writing one file per thread can significantly improve performance. Using a glob pattern upon read or a hive partitioning structure are good ways to transparently handle multiple files.

```
COPY (FROM generate_series(10000000)) TO 'test.parquet' (FORMAT PARQUET,
↪  PER_THREAD_OUTPUT true);
```

Selecting a `row_group_size` The `ROW_GROUP_SIZE` parameter specifies the minimum number of rows in a parquet row group, with a minimum value equal to DuckDB's vector size (currently 2048, but adjustable when compiling DuckDB), and a default of 122880. A parquet row group is a partition of rows, consisting of a column chunk for each column in the dataset.

Compression algorithms are only applied per row group, so the larger the row group size, the more opportunities to compress the data. DuckDB can read parquet row groups in parallel even within the same file and uses predicate pushdown to only scan the row groups whose metadata ranges match the `WHERE` clause of the query. However there is some overhead associated with reading the metadata in each group. A good approach would be to ensure that within each file, the total number of row groups

is at least as large as the number of CPU threads used to query that file. More row groups beyond the thread count would improve the speed of highly selective queries, but slow down queries that must scan the whole file like aggregations.

```
-- write a query to a parquet file with a different row_group_size
COPY (FROM generate_series(100000)) TO 'row-groups.parquet' (FORMAT PARQUET,
 ↪  ROW_GROUP_SIZE 100000);
```

Partitioning

Hive Partitioning

Examples

```
-- read data from a hive partitioned data set
SELECT * FROM read_parquet('orders/*/*/*.parquet', hive_partitioning=1);
-- parquet_scan is an alias of read_parquet, so they are equivalent
SELECT * FROM parquet_scan('orders/*/*/*.parquet', hive_partitioning=1);
-- write a table to a hive partitioned data set
COPY orders TO 'orders' (FORMAT PARQUET, PARTITION_BY (year, month));
```

Hive Partitioning

Hive partitioning is a partitioning strategy that is used to split a table into multiple files based on **partition keys**. The files are organized into folders. Within each folder, the **partition key** has a value that is determined by the name of the folder.

Below is an example of a hive partitioned file hierarchy. The files are partitioned on two keys (`year` and `month`).

```
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
```

Files stored in this hierarchy can be read using the `hive_partitioning` flag.

```
SELECT * FROM read_parquet('orders/*/*/*.parquet', hive_partitioning=1);
```

When we specify the `hive_partitioning` flag, the values of the columns will be read from the directories.

Filter Pushdown Filters on the partition keys are automatically pushed down into the files. This way the system skips reading files that are not necessary to answer a query. For example, consider the following query on the above dataset:

```
SELECT *
FROM read_parquet('orders/*/*/*.parquet', hive_partitioning=1)
WHERE year=2022 AND month=11;
```

When executing this query, only the following files will be read:

```
orders
└── year=2022
    └── month=11
        ├── file4.parquet
        └── file5.parquet
```

Autodetection By default the system tries to infer if the provided files are in a hive partitioned hierarchy. And if so, the `hive_partitioning` flag is enabled automatically. The autodetection will look at the names of the folders and search for a 'key'='value' pattern. This behaviour can be overridden by setting the `hive_partitioning` flag manually.

Hive Types `hive_types` is a way to specify the logical types of the hive partitions in a struct:

```
FROM read_parquet('dir/**/*.parquet', hive_partitioning=1, hive_
↪  types={'release': date, 'orders': bigint});
```

`hive_types` will be autodetected for the following types: `DATE`, `TIMESTAMP` and `BIGINT`. To switch off the autodetection, the flag `hive_types_autocast=0` can be set.

Writing Partitioned Files See the Partitioned Writes section.

Partitioned Writes

Examples

```
-- write a table to a hive partitioned data set of parquet files
COPY orders TO 'orders' (FORMAT PARQUET, PARTITION_BY (year, month));
-- write a table to a hive partitioned data set of CSV files, allowing
 ↪  overwrites
COPY orders TO 'orders' (FORMAT CSV, PARTITION_BY (year, month), OVERWRITE_
 ↪  OR_IGNORE 1);
```

Partitioned Writes

When the `partition_by` clause is specified for the COPY statement, the files are written in a hive partitioned folder hierarchy. The target is the name of the root directory (in the example above: `orders`). The files are written in-order in the file hierarchy. Currently, one file is written per thread to each directory.

```
orders
├── year=2021
│   ├── month=1
│   │   ├── data_1.parquet
│   │   └── data_2.parquet
│   └── month=2
│       └── data_1.parquet
└── year=2022
    ├── month=11
    │   ├── data_1.parquet
    │   └── data_2.parquet
    └── month=12
        └── data_1.parquet
```

The values of the partitions are automatically extracted from the data. Note that it can be very expensive to write many partitions as many files will be created. The ideal partition count depends on how large your data set is.

> **Note.** Writing data into many small partitions is expensive. It is generally recommended to have at least `100MB` of data per partition.

Overwriting By default the partitioned write will not allow overwriting existing directories. Use the `OVERWRITE_OR_IGNORE` option to allow overwriting an existing directory.

Filename Pattern By default, files will be named `data_0.parquet` or `data_0.csv`. With the flag `FILENAME_PATTERN` a pattern with `{i}` or `{uuid}` can be defined to create specific filenames:

- `{i}` will be replaced by an index
- `{uuid}` will be replaced by a 128 bits long UUID

```
-- write a table to a hive partitioned data set of .parquet files, with an
 ↪  index in the filename
COPY orders TO 'orders' (FORMAT PARQUET, PARTITION_BY (year, month),
 ↪  OVERWRITE_OR_IGNORE, FILENAME_PATTERN "orders_{i}");
-- write a table to a hive partitioned data set of .parquet files, with
 ↪  unique filenames
COPY orders TO 'orders' (FORMAT PARQUET, PARTITION_BY (year, month),
 ↪  OVERWRITE_OR_IGNORE, FILENAME_PATTERN "file_{uuid}");
```

Appender

The C++ Appender can be used to load bulk data into a DuckDB database. The Appender is tied to a connection, and will use the transaction context of that connection when appending. An Appender always appends to a single table in the database file.

```
DuckDB db;
Connection con(db);
// create the table
con.Query("CREATE TABLE people(id INTEGER, name VARCHAR)");
// initialize the appender
Appender appender(con, "people");
```

The AppendRow function is the easiest way of appending data. It uses recursive templates to allow you to put all the values of a single row within one function call, as follows:

```
appender.AppendRow(1, "Mark");
```

Rows can also be individually constructed using the BeginRow, EndRow and Append methods. This is done internally by AppendRow, and hence has the same performance characteristics.

```
appender.BeginRow();
appender.Append<int32_t>(2);
appender.Append<string>("Hannes");
appender.EndRow();
```

Any values added to the appender are cached prior to being inserted into the database system for performance reasons. That means that, while appending, the rows might not be immediately visible in

the system. The cache is automatically flushed when the appender goes out of scope or when `appender.Close()` is called. The cache can also be manually flushed using the `appender.Flush()` method. After either `Flush` or `Close` is called, all the data has been written to the database system.

Date, Time and Timestamps

While numbers and strings are rather self-explanatory, dates, times and timestamps require some explanation. They can be directly appended using the methods provided by `duckdb::Date`, `duckdb::Time` or `duckdb::Timestamp`. They can also be appended using the internal `duckdb::Value` type, however, this adds some additional overheads and should be avoided if possible.

Below is a short example:

```
con.Query("CREATE TABLE dates(d DATE, t TIME, ts TIMESTAMP)");
Appender appender(con, "dates");

// construct the values using the Date/Time/Timestamp types - this is the
 ↪  most efficient
appender.AppendRow(Date::FromDate(1992, 1, 1), Time::FromTime(1, 1, 1, 0),
 ↪  Timestamp::FromDatetime(Date::FromDate(1992, 1, 1), Time::FromTime(1, 1,
 ↪  1, 0)));
// construct duckdb::Value objects
appender.AppendRow(Value::DATE(1992, 1, 1), Value::TIME(1, 1, 1, 0),
 ↪  Value::TIMESTAMP(1992, 1, 1, 1, 1, 1, 0));
```

Insert Statements

Insert statements are the standard way of loading data into a relational database. When using insert statements, the values are supplied row-by-row. While simple, there is significant overhead involved in parsing and processing individual insert statements. This makes lots of individual row-by-row insertions very inefficient for bulk insertion.

> **Note.** As a rule-of-thumb, avoid using lots of individual row-by-row insert statements when inserting more than a few rows (i.e., avoid using insert statements as part of a loop). When bulk inserting data, try to maximize the amount of data that is inserted per statement.

If you must use insert statements to load data in a loop, avoid executing the statements in auto-commit mode. After every commit, the database is required to sync the changes made to disk to

ensure no data is lost. In auto-commit mode every single statement will be wrapped in a separate transaction, meaning `fsync` will be called for every statement. This is typically unnecessary when bulk loading and will significantly slow down your program.

> **Note.** If you absolutely must use insert statements in a loop to load data, wrap them in calls to `BEGIN TRANSACTION` and `COMMIT`.

Syntax

An example of using `INSERT INTO` to load data in a table is as follows:

```
CREATE TABLE people(id INTEGER, name VARCHAR);
INSERT INTO people VALUES (1, 'Mark'), (2, 'Hannes');
```

A more detailed description together with syntax diagram can be found here.

Client APIs

Client APIs Overview

There are various client APIs for DuckDB. DuckDB's ”native” API is C++, with ”official” wrappers available for C, Python, R, Java, Node.js, WebAssembly/Wasm, ODBC API, Julia, and a Command Line Interface (CLI).

There are also contributed third-party DuckDB wrappers for:

- C#, by Giorgi
- Common Lisp, by ak-coram
- Crystal, by amauryt
- Go, by marcboeker
- Ruby, by suketa
- Rust, by wangfenjin
- Zig, by karlseguin

C

C API - Overview

DuckDB implements a custom C API modelled somewhat following the SQLite C API. The API is contained in the `duckdb.h` header. Continue to Startup & Shutdown to get started, or check out the Full API overview.

We also provide a SQLite API wrapper which means that if your applications is programmed against the SQLite C API, you can re-link to DuckDB and it should continue working. See the `sqlite_api_wrapper` folder in our source repository for more information.

Installation

The DuckDB C API can be installed as part of the `libduckdb` packages. Please see the installation page for details.

C API - Startup & Shutdown

To use DuckDB, you must first initialize a `duckdb_database` handle using `duckdb_open()`. `duckdb_open()` takes as parameter the database file to read and write from. The special value `NULL` (`nullptr`) can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the process).

With the `duckdb_database` handle, you can create one or many `duckdb_connection` using `duckdb_connect()`. While individual connections are thread-safe, they will be locked during querying. It is therefore recommended that each thread uses its own connection to allow for the best parallel performance.

All `duckdb_connections` have to explicitly be disconnected with `duckdb_disconnect()` and the `duckdb_database` has to be explicitly closed with `duckdb_close()` to avoid memory and file handle leaking.

Example

```
duckdb_database db;
duckdb_connection con;

if (duckdb_open(NULL, &db) == DuckDBError) {
    // handle error
}
if (duckdb_connect(db, &con) == DuckDBError) {
    // handle error
}

// run queries...

// cleanup
duckdb_disconnect(&con);
duckdb_close(&db);
```

API Reference

```
duckdb_state duckdb_open(const char *path, duckdb_database *out_database);
duckdb_state duckdb_open_ext(const char *path, duckdb_database *out_
 ↪  database, duckdb_config config, char **out_error);
void duckdb_close(duckdb_database *database);
duckdb_state duckdb_connect(duckdb_database database, duckdb_connection
 ↪  *out_connection);
void duckdb_interrupt(duckdb_connection connection);
double duckdb_query_progress(duckdb_connection connection);
void duckdb_disconnect(duckdb_connection *connection);
const char *duckdb_library_version();
```

duckdb_open Creates a new database or opens an existing database file stored at the given path. If no path is given a new in-memory database is created instead. The instantiated database should be closed with 'duckdb_close'

Syntax

```
duckdb_state duckdb_open(
  const char *path,
  duckdb_database *out_database
);
```

Parameters

- `path`

Path to the database file on disk, or `nullptr` or `:memory:` to open an in-memory database.

- `out_database`

The result database object.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_open_ext Extended version of duckdb_open. Creates a new database or opens an existing database file stored at the given path.

Syntax

```
duckdb_state duckdb_open_ext(
  const char *path,
  duckdb_database *out_database,
  duckdb_config config,
  char **out_error
);
```

Parameters

- `path`

Path to the database file on disk, or `nullptr` or `:memory:` to open an in-memory database.

- `out_database`

The result database object.

- `config`

(Optional) configuration used to start up the database system.

- `out_error`

If set and the function returns DuckDBError, this will contain the reason why the start-up failed. Note that the error must be freed using `duckdb_free`.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_close` Closes the specified database and de-allocates all memory allocated for that database. This should be called after you are done with any database allocated through `duckdb_open`. Note that failing to call `duckdb_close` (in case of e.g., a program crash) will not cause data corruption. Still it is recommended to always correctly close a database object after you are done with it.

Syntax

```
void duckdb_close(
  duckdb_database *database
);
```

Parameters

- `database`

The database object to shut down.

`duckdb_connect` Opens a connection to a database. Connections are required to query the database, and store transactional state associated with the connection. The instantiated connection should be closed using 'duckdb_disconnect'

Syntax

```
duckdb_state duckdb_connect(
  duckdb_database database,
  duckdb_connection *out_connection
);
```

Parameters

- `database`

The database file to connect to.

- `out_connection`

The result connection object.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_interrupt` Interrupt running query

Syntax

```
void duckdb_interrupt(
  duckdb_connection connection
);
```

Parameters

- `connection`

The connection to interruot

`duckdb_query_progress` Get progress of the running query

Syntax

```
double duckdb_query_progress(
  duckdb_connection connection
);
```

Parameters

- `connection`

The working connection

- `returns`

-1 if no progress or a percentage of the progress

`duckdb_disconnect` Closes the specified connection and de-allocates all memory allocated for that connection.

Syntax

```
void duckdb_disconnect(
  duckdb_connection *connection
);
```

Parameters

- `connection`

The connection to close.

`duckdb_library_version` Returns the version of the linked DuckDB, with a version postfix for dev versions

Usually used for developing C extensions that must return this for a compatibility check.

Syntax

```
const char *duckdb_library_version(

);
```

C API - Configuration

Configuration options can be provided to change different settings of the database system. Note that many of these settings can be changed later on using PRAGMA statements as well. The configuration object should be created, filled with values and passed to `duckdb_open_ext`.

Example

```
duckdb_database db;
duckdb_config config;

// create the configuration object
if (duckdb_create_config(&config) == DuckDBError) {
    // handle error
}
// set some configuration options
duckdb_set_config(config, "access_mode", "READ_WRITE"); // or READ_ONLY
duckdb_set_config(config, "threads", "8");
duckdb_set_config(config, "max_memory", "8GB");
duckdb_set_config(config, "default_order", "DESC");

// open the database using the configuration
if (duckdb_open_ext(NULL, &db, config, NULL) == DuckDBError) {
    // handle error
}
// cleanup the configuration object
duckdb_destroy_config(&config);

// run queries...

// cleanup
duckdb_close(&db);
```

API Reference

```
duckdb_state duckdb_create_config(duckdb_config *out_config);
size_t duckdb_config_count();
duckdb_state duckdb_get_config_flag(size_t index, const char **out_name,
 ↪  const char **out_description);
duckdb_state duckdb_set_config(duckdb_config config, const char *name, const
 ↪  char *option);
```

```
void duckdb_destroy_config(duckdb_config *config);
```

`duckdb_create_config` Initializes an empty configuration object that can be used to provide start-up options for the DuckDB instance through `duckdb_open_ext`.

This will always succeed unless there is a malloc failure.

Syntax

```
duckdb_state duckdb_create_config(
  duckdb_config *out_config
);
```

Parameters

- `out_config`

The result configuration object.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_config_count` This returns the total amount of configuration options available for usage with `duckdb_get_config_flag`.

This should not be called in a loop as it internally loops over all the options.

Syntax

```
size_t duckdb_config_count(

);
```

Parameters

- `returns`

The amount of config options available.

`duckdb_get_config_flag` Obtains a human-readable name and description of a specific configuration option. This can be used to e.g. display configuration options. This will succeed unless `index` is out of range (i.e., `>= duckdb_config_count`).

The result name or description MUST NOT be freed.

Syntax

```
duckdb_state duckdb_get_config_flag(
  size_t index,
  const char **out_name,
  const char **out_description
);
```

Parameters

- `index`

The index of the configuration option (between 0 and `duckdb_config_count`)

- `out_name`

A name of the configuration flag.

- `out_description`

A description of the configuration flag.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_set_config` Sets the specified option for the specified configuration. The configuration option is indicated by name. To obtain a list of config options, see `duckdb_get_config_flag`.

In the source code, configuration options are defined in `config.cpp`.

This can fail if either the name is invalid, or if the value provided for the option is invalid.

Syntax

```
duckdb_state duckdb_set_config(
  duckdb_config config,
  const char *name,
  const char *option
);
```

Parameters

- `duckdb_config`

The configuration object to set the option on.

- `name`

The name of the configuration flag to set.

- `option`

The value to set the configuration flag to.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_config` Destroys the specified configuration option and de-allocates all memory allocated for the object.

Syntax

```
void duckdb_destroy_config(
  duckdb_config *config
);
```

Parameters

- `config`

The configuration object to destroy.

C API - Query

The `duckdb_query` method allows SQL queries to be run in DuckDB from C. This method takes two parameters, a (null-terminated) SQL query string and a `duckdb_result` result pointer. The result pointer may be `NULL` if the application is not interested in the result set or if the query produces no result. After the result is consumed, the `duckdb_destroy_result` method should be used to clean up the result.

Elements can be extracted from the `duckdb_result` object using a variety of methods. The `duckdb_column_count` and `duckdb_row_count` methods can be used to extract the number of columns and the number of rows, respectively. `duckdb_column_name` and `duckdb_column_type` can be used to extract the names and types of individual columns.

Example

```
duckdb_state state;
duckdb_result result;

// create a table
state = duckdb_query(con, "CREATE TABLE integers(i INTEGER, j INTEGER);",
 ↪  NULL);
if (state == DuckDBError) {
    // handle error
}
// insert three rows into the table
state = duckdb_query(con, "INSERT INTO integers VALUES (3, 4), (5, 6), (7,
 ↪  NULL);", NULL);
if (state == DuckDBError) {
    // handle error
}
// query rows again
state = duckdb_query(con, "SELECT * FROM integers", &result);
if (state == DuckDBError) {
    // handle error
}
// handle the result
// ...

// destroy the result after we are done with it
duckdb_destroy_result(&result);
```

Value Extraction

Values can be extracted using either the `duckdb_column_data`/`duckdb_nullmask_data` functions, or using the `duckdb_value` convenience functions. The `duckdb_column_data`/`duckdb_nullmask_data` functions directly hand you a pointer to the result arrays in columnar format, and can therefore be very fast. The `duckdb_value` functions perform bounds- and type-checking, and will automatically cast values to the desired type. This makes them more convenient and easier to use, at the expense of being slower.

See the Types page for more information.

> **Note.** For optimal performance, use `duckdb_column_data` and `duckdb_nullmask_data` to extract data from the query result. The `duckdb_value` functions perform internal

> type-checking, bounds-checking and casting which makes them slower.

`duckdb_value` Below is an example that prints the above result to CSV format using the `duckdb_value_varchar` function. Note that the function is generic: we do not need to know about the types of the individual result columns.

```
// print the above result to CSV format using `duckdb_value_varchar`
idx_t row_count = duckdb_row_count(&result);
idx_t column_count = duckdb_column_count(&result);
for(idx_t row = 0; row < row_count; row++) {
    for(idx_t col = 0; col < column_count; col++) {
        if (col > 0) printf(",");
        auto str_val = duckdb_value_varchar(&result, col, row);
        printf("%s", str_val);
        duckdb_free(str_val);
    }
    printf("\n");
}
```

`duckdb_column_data` Below is an example that prints the above result to CSV format using the `duckdb_column_data` function. Note that the function is NOT generic: we do need to know exactly what the types of the result columns are.

```
int32_t *i_data = (int32_t *) duckdb_column_data(&result, 0);
int32_t *j_data = (int32_t *) duckdb_column_data(&result, 1);
bool    *i_mask = duckdb_nullmask_data(&result, 0);
bool    *j_mask = duckdb_nullmask_data(&result, 1);
idx_t row_count = duckdb_row_count(&result);
for(idx_t row = 0; row < row_count; row++) {
    if (i_mask[row]) {
        printf("NULL");
    } else {
        printf("%d", i_data[row]);
    }
    printf(",");
    if (j_mask[row]) {
        printf("NULL");
    } else {
        printf("%d", j_data[row]);
    }
    printf("\n");
}
```

> **Note.** When using `duckdb_column_data`, be careful that the type matches exactly what you expect it to be. As the code directly accesses an internal array, there is no type-checking. Accessing a `DUCKDB_TYPE_INTEGER` column as if it was a `DUCKDB_TYPE_BIGINT` column will provide unpredictable results!

API Reference

```
duckdb_state duckdb_query(duckdb_connection connection, const char *query,
 ↪  duckdb_result *out_result);
void duckdb_destroy_result(duckdb_result *result);
const char *duckdb_column_name(duckdb_result *result, idx_t col);
duckdb_type duckdb_column_type(duckdb_result *result, idx_t col);
duckdb_logical_type duckdb_column_logical_type(duckdb_result *result, idx_t
 ↪  col);
idx_t duckdb_column_count(duckdb_result *result);
idx_t duckdb_row_count(duckdb_result *result);
idx_t duckdb_rows_changed(duckdb_result *result);
void *duckdb_column_data(duckdb_result *result, idx_t col);
bool *duckdb_nullmask_data(duckdb_result *result, idx_t col);
const char *duckdb_result_error(duckdb_result *result);
```

`duckdb_query` Executes a SQL query within a connection and stores the full (materialized) result in the out_result pointer. If the query fails to execute, DuckDBError is returned and the error message can be retrieved by calling `duckdb_result_error`.

Note that after running `duckdb_query`, `duckdb_destroy_result` must be called on the result object even if the query fails, otherwise the error stored within the result will not be freed correctly.

Syntax

```
duckdb_state duckdb_query(
  duckdb_connection connection,
  const char *query,
  duckdb_result *out_result
);
```

Parameters

- `connection`

The connection to perform the query in.

- `query`

The SQL query to run.

- `out_result`

The query result.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_result` Closes the result and de-allocates all memory allocated for that connection.

Syntax

```
void duckdb_destroy_result(
  duckdb_result *result
);
```

Parameters

- `result`

The result to destroy.

`duckdb_column_name` Returns the column name of the specified column. The result should not need be freed; the column names will automatically be destroyed when the result is destroyed.

Returns `NULL` if the column is out of range.

Syntax

```
const char *duckdb_column_name(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`

The result object to fetch the column name from.

- `col`

The column index.

- `returns`

The column name of the specified column.

`duckdb_column_type` Returns the column type of the specified column.

Returns `DUCKDB_TYPE_INVALID` if the column is out of range.

Syntax

```
duckdb_type duckdb_column_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`

The result object to fetch the column type from.

- `col`

The column index.

- `returns`

The column type of the specified column.

`duckdb_column_logical_type` Returns the logical column type of the specified column.

The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Returns `NULL` if the column is out of range.

Syntax

```
duckdb_logical_type duckdb_column_logical_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`

The result object to fetch the column type from.

- `col`

The column index.

- `returns`

The logical column type of the specified column.

`duckdb_column_count` Returns the number of columns present in a the result object.

Syntax

```
idx_t duckdb_column_count(
  duckdb_result *result
);
```

Parameters

- `result`

The result object.

- `returns`

The number of columns present in the result object.

`duckdb_row_count` Returns the number of rows present in a the result object.

Syntax

```
idx_t duckdb_row_count(
  duckdb_result *result
);
```

Parameters

- `result`

The result object.

- `returns`

The number of rows present in the result object.

`duckdb_rows_changed` Returns the number of rows changed by the query stored in the result. This is relevant only for INSERT/UPDATE/DELETE queries. For other queries the rows_changed will be 0.

Syntax

```
idx_t duckdb_rows_changed(
  duckdb_result *result
);
```

Parameters

- `result`

The result object.

- `returns`

The number of rows changed.

`duckdb_column_data` **DEPRECATED**: Prefer using `duckdb_result_get_chunk` instead.

Returns the data of a specific column of a result in columnar format.

The function returns a dense array which contains the result data. The exact type stored in the array depends on the corresponding duckdb_type (as provided by `duckdb_column_type`). For the exact type by which the data should be accessed, see the comments in the types section or the DUCKDB_TYPE enum.

For example, for a column of type DUCKDB_TYPE_INTEGER, rows can be accessed in the following manner:

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
printf("Data for row %d: %d\n", row, data[row]);
```

Syntax

```
void *duckdb_column_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- result

The result object to fetch the column data from.

- col

The column index.

- returns

The column data of the specified column.

duckdb_nullmask_data **DEPRECATED**: Prefer using duckdb_result_get_chunk instead.

Returns the nullmask of a specific column of a result in columnar format. The nullmask indicates for every row whether or not the corresponding row is NULL. If a row is NULL, the values present in the array provided by duckdb_column_data are undefined.

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
bool *nullmask = duckdb_nullmask_data(&result, 0);
if (nullmask[row]) {
printf("Data for row %d: NULL\n", row);
} else {
printf("Data for row %d: %d\n", row, data[row]);
}
```

Syntax

```
bool *duckdb_nullmask_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`

The result object to fetch the nullmask from.

- `col`

The column index.

- `returns`

The nullmask of the specified column.

`duckdb_result_error` Returns the error message contained within the result. The error is only set if `duckdb_query` returns `DuckDBError`.

The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_result` is called.

Syntax

```
const char *duckdb_result_error(
  duckdb_result *result
);
```

Parameters

- `result`

The result object to fetch the error from.

- `returns`

The error of the result.

C API - Data Chunks

Data chunks represent a horizontal slice of a table. They hold a number of vectors, that can each hold up to the `VECTOR_SIZE` rows. The vector size can be obtained through the `duckdb_vector_size` function and is configurable, but is usually set to `2048`.

Data chunks and vectors are what DuckDB uses natively to store and represent data. For this reason, the data chunk interface is the most efficient way of interfacing with DuckDB. Be aware, however, that correctly interfacing with DuckDB using the data chunk API does require knowledge of DuckDB's internal vector format.

The primary manner of interfacing with data chunks is by obtaining the internal vectors of the data chunk using the `duckdb_data_chunk_get_vector` method, and subsequently using the `duckdb_vector_get_data` and `duckdb_vector_get_validity` methods to read the internal data and the validity mask of the vector. For composite types (list and struct vectors), `duckdb_list_vector_get_child` and `duckdb_struct_vector_get_child` should be used to read child vectors.

API Reference

```
duckdb_data_chunk duckdb_create_data_chunk(duckdb_logical_type *types, idx_t
 ↪  column_count);
void duckdb_destroy_data_chunk(duckdb_data_chunk *chunk);
void duckdb_data_chunk_reset(duckdb_data_chunk chunk);
idx_t duckdb_data_chunk_get_column_count(duckdb_data_chunk chunk);
duckdb_vector duckdb_data_chunk_get_vector(duckdb_data_chunk chunk, idx_t
 ↪  col_idx);
idx_t duckdb_data_chunk_get_size(duckdb_data_chunk chunk);
void duckdb_data_chunk_set_size(duckdb_data_chunk chunk, idx_t size);
```

Vector Interface

```
duckdb_logical_type duckdb_vector_get_column_type(duckdb_vector vector);
void *duckdb_vector_get_data(duckdb_vector vector);
uint64_t *duckdb_vector_get_validity(duckdb_vector vector);
void duckdb_vector_ensure_validity_writable(duckdb_vector vector);
void duckdb_vector_assign_string_element(duckdb_vector vector, idx_t index,
 ↪  const char *str);
void duckdb_vector_assign_string_element_len(duckdb_vector vector, idx_t
 ↪  index, const char *str, idx_t str_len);
duckdb_vector duckdb_list_vector_get_child(duckdb_vector vector);
```

```
idx_t duckdb_list_vector_get_size(duckdb_vector vector);
duckdb_state duckdb_list_vector_set_size(duckdb_vector vector, idx_t size);
duckdb_state duckdb_list_vector_reserve(duckdb_vector vector, idx_t
 ↪  required_capacity);
duckdb_vector duckdb_struct_vector_get_child(duckdb_vector vector, idx_t
 ↪  index);
```

Validity Mask Functions

```
bool duckdb_validity_row_is_valid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_validity(uint64_t *validity, idx_t row, bool
 ↪  valid);
void duckdb_validity_set_row_invalid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_valid(uint64_t *validity, idx_t row);
```

`duckdb_create_data_chunk` Creates an empty DataChunk with the specified set of types.

Syntax

```
duckdb_data_chunk duckdb_create_data_chunk(
  duckdb_logical_type *types,
  idx_t column_count
);
```

Parameters

- `types`

An array of types of the data chunk.

- `column_count`

The number of columns.

- `returns`

The data chunk.

`duckdb_destroy_data_chunk` Destroys the data chunk and de-allocates all memory allocated for that chunk.

Syntax

```
void duckdb_destroy_data_chunk(
  duckdb_data_chunk *chunk
);
```

Parameters

- chunk

The data chunk to destroy.

`duckdb_data_chunk_reset` Resets a data chunk, clearing the validity masks and setting the cardinality of the data chunk to 0.

Syntax

```
void duckdb_data_chunk_reset(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk

The data chunk to reset.

`duckdb_data_chunk_get_column_count` Retrieves the number of columns in a data chunk.

Syntax

```
idx_t duckdb_data_chunk_get_column_count(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk

The data chunk to get the data from

- `returns`

The number of columns in the data chunk

`duckdb_data_chunk_get_vector` Retrieves the vector at the specified column index in the data chunk.

The pointer to the vector is valid for as long as the chunk is alive. It does NOT need to be destroyed.

Syntax

```
duckdb_vector duckdb_data_chunk_get_vector(
  duckdb_data_chunk chunk,
  idx_t col_idx
);
```

Parameters

- `chunk`

The data chunk to get the data from

- `returns`

The vector

`duckdb_data_chunk_get_size` Retrieves the current number of tuples in a data chunk.

Syntax

```
idx_t duckdb_data_chunk_get_size(
  duckdb_data_chunk chunk
);
```

Parameters

- `chunk`

The data chunk to get the data from

- `returns`

The number of tuples in the data chunk

`duckdb_data_chunk_set_size` Sets the current number of tuples in a data chunk.

Syntax

```
void duckdb_data_chunk_set_size(
  duckdb_data_chunk chunk,
  idx_t size
);
```

Parameters

- `chunk`

The data chunk to set the size in

- `size`

The number of tuples in the data chunk

`duckdb_vector_get_column_type` Retrieves the column type of the specified vector.

The result must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_vector_get_column_type(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector get the data from

- `returns`

The type of the vector

`duckdb_vector_get_data` Retrieves the data pointer of the vector.

The data pointer can be used to read or write values from the vector. How to read or write values depends on the type of the vector.

Syntax

```
void *duckdb_vector_get_data(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector to get the data from

- `returns`

The data pointer

`duckdb_vector_get_validity` Retrieves the validity mask pointer of the specified vector.

If all values are valid, this function MIGHT return NULL!

The validity mask is a bitset that signifies null-ness within the data chunk. It is a series of uint64_t values, where each uint64_t value contains validity for 64 tuples. The bit is set to 1 if the value is valid (i.e., not NULL) or 0 if the value is invalid (i.e., NULL).

Validity of a specific value can be obtained like this:

idx_t entry_idx = row_idx / 64; idx_t idx_in_entry = row_idx % 64; bool is_valid = validity_mask[entry_idx] & (1 « idx_in_entry);

Alternatively, the (slower) duckdb_validity_row_is_valid function can be used.

Syntax

```
uint64_t *duckdb_vector_get_validity(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector to get the data from

- `returns`

The pointer to the validity mask, or NULL if no validity mask is present

`duckdb_vector_ensure_validity_writable` Ensures the validity mask is writable by allocating it.

After this function is called, `duckdb_vector_get_validity` will ALWAYS return non-NULL. This allows null values to be written to the vector, regardless of whether a validity mask was present before.

Syntax

```
void duckdb_vector_ensure_validity_writable(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector to alter

`duckdb_vector_assign_string_element` Assigns a string element in the vector at the specified location.

Syntax

```
void duckdb_vector_assign_string_element(
  duckdb_vector vector,
  idx_t index,
  const char *str
);
```

Parameters

- `vector`

The vector to alter

- `index`

The row position in the vector to assign the string to

- `str`

The null-terminated string

`duckdb_vector_assign_string_element_len` Assigns a string element in the vector at the specified location.

Syntax

```
void duckdb_vector_assign_string_element_len(
  duckdb_vector vector,
  idx_t index,
  const char *str,
  idx_t str_len
);
```

Parameters

- `vector`

The vector to alter

- `index`

The row position in the vector to assign the string to

- `str`

The string

- `str_len`

The length of the string (in bytes)

`duckdb_list_vector_get_child` Retrieves the child vector of a list vector.

The resulting vector is valid as long as the parent vector is valid.

Syntax

```
duckdb_vector duckdb_list_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector

- `returns`

The child vector

`duckdb_list_vector_get_size` Returns the size of the child vector of the list

Syntax

```
idx_t duckdb_list_vector_get_size(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector

- `returns`

The size of the child list

`duckdb_list_vector_set_size` Sets the total size of the underlying child-vector of a list vector.

Syntax

```
duckdb_state duckdb_list_vector_set_size(
  duckdb_vector vector,
  idx_t size
);
```

Parameters

- `vector`

The list vector.

- `size`

The size of the child list.

- `returns`

The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_list_vector_reserve` Sets the total capacity of the underlying child-vector of a list.

Syntax

```
duckdb_state duckdb_list_vector_reserve(
  duckdb_vector vector,
  idx_t required_capacity
);
```

Parameters

- `vector`

The list vector.

- `required_capacity`

the total capacity to reserve.

- `return`

The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_struct_vector_get_child` Retrieves the child vector of a struct vector.

The resulting vector is valid as long as the parent vector is valid.

Syntax

```
duckdb_vector duckdb_struct_vector_get_child(
  duckdb_vector vector,
  idx_t index
);
```

Parameters

- `vector`

The vector

- `index`

The child index

- `returns`

The child vector

`duckdb_validity_row_is_valid` Returns whether or not a row is valid (i.e., not NULL) in the given validity mask.

Syntax

```
bool duckdb_validity_row_is_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`

The validity mask, as obtained through `duckdb_vector_get_validity`

- `row`

The row index

- `returns`

true if the row is valid, false otherwise

`duckdb_validity_set_row_validity` In a validity mask, sets a specific row to either valid or invalid.

Note that `duckdb_vector_ensure_validity_writable` should be called before calling `duckdb_vector_get_validity`, to ensure that there is a validity mask to write to.

Syntax

```
void duckdb_validity_set_row_validity(
  uint64_t *validity,
  idx_t row,
  bool valid
);
```

Parameters

- `validity`

The validity mask, as obtained through `duckdb_vector_get_validity`.

- `row`

The row index

- `valid`

Whether or not to set the row to valid, or invalid

`duckdb_validity_set_row_invalid` In a validity mask, sets a specific row to invalid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to false.

Syntax

```
void duckdb_validity_set_row_invalid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`

The validity mask

- `row`

The row index

`duckdb_validity_set_row_valid` In a validity mask, sets a specific row to valid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to true.

Syntax

```
void duckdb_validity_set_row_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`

The validity mask

- `row`

The row index

C API - Values

The value class represents a single value of any type.

API Reference

```
void duckdb_destroy_value(duckdb_value *value);
duckdb_value duckdb_create_varchar(const char *text);
duckdb_value duckdb_create_varchar_length(const char *text, idx_t length);
duckdb_value duckdb_create_int64(int64_t val);
char *duckdb_get_varchar(duckdb_value value);
int64_t duckdb_get_int64(duckdb_value value);
```

`duckdb_destroy_value` Destroys the value and de-allocates all memory allocated for that type.

Syntax

```
void duckdb_destroy_value(
  duckdb_value *value
);
```

Parameters

- `value`

The value to destroy.

`duckdb_create_varchar` Creates a value from a null-terminated string

Syntax

```
duckdb_value duckdb_create_varchar(
  const char *text
);
```

Parameters

- `value`

The null-terminated string

- `returns`

The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_varchar_length` Creates a value from a string

Syntax

```
duckdb_value duckdb_create_varchar_length(
  const char *text,
  idx_t length
);
```

Parameters

- `value`

The text

- `length`

The length of the text

- `returns`

The value. This must be destroyed with `duckdb_destroy_value`.

duckdb_create_int64 Creates a value from an int64

Syntax

```
duckdb_value duckdb_create_int64(
  int64_t val
);
```

Parameters

- value

The bigint value

- returns

The value. This must be destroyed with duckdb_destroy_value.

duckdb_get_varchar Obtains a string representation of the given value. The result must be destroyed with duckdb_free.

Syntax

```
char *duckdb_get_varchar(
  duckdb_value value
);
```

Parameters

- value

The value

- returns

The string value. This must be destroyed with duckdb_free.

duckdb_get_int64 Obtains an int64 of the given value.

Syntax

```
int64_t duckdb_get_int64(
  duckdb_value value
);
```

Parameters

- `value`

The value

- `returns`

The int64 value, or 0 if no conversion is possible

C API - Types

DuckDB is a strongly typed database system. As such, every column has a single type specified. This type is constant over the entire column. That is to say, a column that is labeled as an `INTEGER` column will only contain `INTEGER` values.

DuckDB also supports columns of composite types. For example, it is possible to define an array of integers (`INT[]`). It is also possible to define types as arbitrary structs (`ROW(i INTEGER, j VARCHAR)`). For that reason, native DuckDB type objects are not mere enums, but a class that can potentially be nested.

Types in the C API are modeled using an enum (`duckdb_type`) and a complex class (`duckdb_logical_type`). For most primitive types, e.g., integers or varchars, the enum is sufficient. For more complex types, such as lists, structs or decimals, the logical type must be used.

```
typedef enum DUCKDB_TYPE {
  DUCKDB_TYPE_INVALID,
  DUCKDB_TYPE_BOOLEAN,
  DUCKDB_TYPE_TINYINT,
  DUCKDB_TYPE_SMALLINT,
  DUCKDB_TYPE_INTEGER,
  DUCKDB_TYPE_BIGINT,
  DUCKDB_TYPE_UTINYINT,
  DUCKDB_TYPE_USMALLINT,
  DUCKDB_TYPE_UINTEGER,
  DUCKDB_TYPE_UBIGINT,
  DUCKDB_TYPE_FLOAT,
  DUCKDB_TYPE_DOUBLE,
```

```
  DUCKDB_TYPE_TIMESTAMP,
  DUCKDB_TYPE_DATE,
  DUCKDB_TYPE_TIME,
  DUCKDB_TYPE_INTERVAL,
  DUCKDB_TYPE_HUGEINT,
  DUCKDB_TYPE_VARCHAR,
  DUCKDB_TYPE_BLOB,
  DUCKDB_TYPE_DECIMAL,
  DUCKDB_TYPE_TIMESTAMP_S,
  DUCKDB_TYPE_TIMESTAMP_MS,
  DUCKDB_TYPE_TIMESTAMP_NS,
  DUCKDB_TYPE_ENUM,
  DUCKDB_TYPE_LIST,
  DUCKDB_TYPE_STRUCT,
  DUCKDB_TYPE_MAP,
  DUCKDB_TYPE_UUID,
  DUCKDB_TYPE_UNION,
  DUCKDB_TYPE_BIT,
} duckdb_type;
```

Functions

The enum type of a column in the result can be obtained using the `duckdb_column_type` function. The logical type of a column can be obtained using the `duckdb_column_logical_type` function.

`duckdb_value` The `duckdb_value` functions will auto-cast values as required. For example, it is no problem to use `duckdb_value_double` on a column of type `duckdb_value_int32`. The value will be auto-cast and returned as a double. Note that in certain cases the cast may fail. For example, this can happen if we request a `duckdb_value_int8` and the value does not fit within an `int8` value. In this case, a default value will be returned (usually `0` or `nullptr`). The same default value will also be returned if the corresponding value is `NULL`.

The `duckdb_value_is_null` function can be used to check if a specific value is `NULL` or not.

The exception to the auto-cast rule is the `duckdb_value_varchar_internal` function. This function does not auto-cast and only works for `VARCHAR` columns. The reason this function exists is that the result does not need to be freed.

> **Note.** Note that `duckdb_value_varchar` and `duckdb_value_blob` require the result to be de-allocated using `duckdb_free`.

`duckdb_result_get_chunk` The `duckdb_result_get_chunk` function can be used to read data chunks from a DuckDB result set, and is the most efficient way of reading data from a DuckDB result using the C API. It is also the only way of reading data of certain types from a DuckDB result. For example, the `duckdb_value` functions do not support structural reading of composite types (lists or structs) or more complex types like enums and decimals.

For more information about data chunks, see the documentation on data chunks.

API Reference

```
duckdb_data_chunk duckdb_result_get_chunk(duckdb_result result, idx_t chunk_
 ↪  index);
bool duckdb_result_is_streaming(duckdb_result result);
idx_t duckdb_result_chunk_count(duckdb_result result);
bool duckdb_value_boolean(duckdb_result *result, idx_t col, idx_t row);
int8_t duckdb_value_int8(duckdb_result *result, idx_t col, idx_t row);
int16_t duckdb_value_int16(duckdb_result *result, idx_t col, idx_t row);
int32_t duckdb_value_int32(duckdb_result *result, idx_t col, idx_t row);
int64_t duckdb_value_int64(duckdb_result *result, idx_t col, idx_t row);
duckdb_hugeint duckdb_value_hugeint(duckdb_result *result, idx_t col, idx_t
 ↪  row);
duckdb_decimal duckdb_value_decimal(duckdb_result *result, idx_t col, idx_t
 ↪  row);
uint8_t duckdb_value_uint8(duckdb_result *result, idx_t col, idx_t row);
uint16_t duckdb_value_uint16(duckdb_result *result, idx_t col, idx_t row);
uint32_t duckdb_value_uint32(duckdb_result *result, idx_t col, idx_t row);
uint64_t duckdb_value_uint64(duckdb_result *result, idx_t col, idx_t row);
float duckdb_value_float(duckdb_result *result, idx_t col, idx_t row);
double duckdb_value_double(duckdb_result *result, idx_t col, idx_t row);
duckdb_date duckdb_value_date(duckdb_result *result, idx_t col, idx_t row);
duckdb_time duckdb_value_time(duckdb_result *result, idx_t col, idx_t row);
duckdb_timestamp duckdb_value_timestamp(duckdb_result *result, idx_t col,
 ↪  idx_t row);
duckdb_interval duckdb_value_interval(duckdb_result *result, idx_t col, idx_
 ↪  t row);
char *duckdb_value_varchar(duckdb_result *result, idx_t col, idx_t row);
char *duckdb_value_varchar_internal(duckdb_result *result, idx_t col, idx_t
 ↪  row);
duckdb_string duckdb_value_string_internal(duckdb_result *result, idx_t col,
 ↪  idx_t row);
duckdb_blob duckdb_value_blob(duckdb_result *result, idx_t col, idx_t row);
bool duckdb_value_is_null(duckdb_result *result, idx_t col, idx_t row);
```

Date/Time/Timestamp Helpers

```
duckdb_date_struct duckdb_from_date(duckdb_date date);
duckdb_date duckdb_to_date(duckdb_date_struct date);
duckdb_time_struct duckdb_from_time(duckdb_time time);
duckdb_time duckdb_to_time(duckdb_time_struct time);
duckdb_timestamp_struct duckdb_from_timestamp(duckdb_timestamp ts);
duckdb_timestamp duckdb_to_timestamp(duckdb_timestamp_struct ts);
```

Hugeint Helpers

```
double duckdb_hugeint_to_double(duckdb_hugeint val);
duckdb_hugeint duckdb_double_to_hugeint(double val);
duckdb_decimal duckdb_double_to_decimal(double val, uint8_t width, uint8_t
 ↪  scale);
```

Decimal Helpers

```
double duckdb_decimal_to_double(duckdb_decimal val);
```

Logical Type Interface

```
duckdb_logical_type duckdb_create_logical_type(duckdb_type type);
duckdb_logical_type duckdb_create_list_type(duckdb_logical_type type);
duckdb_logical_type duckdb_create_map_type(duckdb_logical_type key_type,
 ↪  duckdb_logical_type value_type);
duckdb_logical_type duckdb_create_union_type(duckdb_logical_type member_
 ↪  types, const char **member_names, idx_t member_count);
duckdb_logical_type duckdb_create_struct_type(duckdb_logical_type *member_
 ↪  types, const char **member_names, idx_t member_count);
duckdb_logical_type duckdb_create_decimal_type(uint8_t width, uint8_t
 ↪  scale);
duckdb_type duckdb_get_type_id(duckdb_logical_type type);
uint8_t duckdb_decimal_width(duckdb_logical_type type);
uint8_t duckdb_decimal_scale(duckdb_logical_type type);
duckdb_type duckdb_decimal_internal_type(duckdb_logical_type type);
duckdb_type duckdb_enum_internal_type(duckdb_logical_type type);
uint32_t duckdb_enum_dictionary_size(duckdb_logical_type type);
char *duckdb_enum_dictionary_value(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_list_type_child_type(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_key_type(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_value_type(duckdb_logical_type type);
idx_t duckdb_struct_type_child_count(duckdb_logical_type type);
```

```
char *duckdb_struct_type_child_name(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_struct_type_child_type(duckdb_logical_type type,
 ↪  idx_t index);
idx_t duckdb_union_type_member_count(duckdb_logical_type type);
char *duckdb_union_type_member_name(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_union_type_member_type(duckdb_logical_type type,
 ↪  idx_t index);
void duckdb_destroy_logical_type(duckdb_logical_type *type);
```

`duckdb_result_get_chunk` Fetches a data chunk from the duckdb_result. This function should be called repeatedly until the result is exhausted.

The result must be destroyed with `duckdb_destroy_data_chunk`.

This function supersedes all `duckdb_value` functions, as well as the `duckdb_column_data` and `duckdb_nullmask_data` functions. It results in significantly better performance, and should be preferred in newer code-bases.

If this function is used, none of the other result functions can be used and vice versa (i.e., this function cannot be mixed with the legacy result functions).

Use `duckdb_result_chunk_count` to figure out how many chunks there are in the result.

Syntax

```
duckdb_data_chunk duckdb_result_get_chunk(
  duckdb_result result,
  idx_t chunk_index
);
```

Parameters

- `result`

The result object to fetch the data chunk from.

- `chunk_index`

The chunk index to fetch from.

- `returns`

The resulting data chunk. Returns `NULL` if the chunk index is out of bounds.

`duckdb_result_is_streaming` Checks if the type of the internal result is StreamQueryResult.

Syntax

```
bool duckdb_result_is_streaming(
  duckdb_result result
);
```

Parameters

- `result`

The result object to check.

- `returns`

Whether or not the result object is of the type StreamQueryResult

`duckdb_result_chunk_count` Returns the number of data chunks present in the result.

Syntax

```
idx_t duckdb_result_chunk_count(
  duckdb_result result
);
```

Parameters

- `result`

The result object

- `returns`

Number of data chunks present in the result.

`duckdb_value_boolean`

Syntax

```
bool duckdb_value_boolean(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The boolean value at the specified location, or false if the value cannot be converted.

duckdb_value_int8

Syntax

```
int8_t duckdb_value_int8(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The int8_t value at the specified location, or 0 if the value cannot be converted.

duckdb_value_int16

Syntax

```
int16_t duckdb_value_int16(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The int16_t value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_int32`

Syntax

```
int32_t duckdb_value_int32(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The int32_t value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_int64`

Syntax

```
int64_t duckdb_value_int64(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The int64_t value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_hugeint`

Syntax

```
duckdb_hugeint duckdb_value_hugeint(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The duckdb_hugeint value at the specified location, or 0 if the value cannot be converted.

duckdb_value_decimal

Syntax

```
duckdb_decimal duckdb_value_decimal(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The duckdb_decimal value at the specified location, or 0 if the value cannot be converted.

duckdb_value_uint8

Syntax

```
uint8_t duckdb_value_uint8(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The uint8_t value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_uint16`

Syntax

```
uint16_t duckdb_value_uint16(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The uint16_t value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_uint32`

Syntax

```
uint32_t duckdb_value_uint32(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The uint32_t value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_uint64`

Syntax

```
uint64_t duckdb_value_uint64(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The uint64_t value at the specified location, or 0 if the value cannot be converted.

duckdb_value_float

Syntax

```
float duckdb_value_float(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The float value at the specified location, or 0 if the value cannot be converted.

duckdb_value_double

Syntax

```
double duckdb_value_double(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The double value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_date`

Syntax

```
duckdb_date duckdb_value_date(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The duckdb_date value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_time`

Syntax

```
duckdb_time duckdb_value_time(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The duckdb_time value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_timestamp`

Syntax

```
duckdb_timestamp duckdb_value_timestamp(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The duckdb_timestamp value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_interval`

Syntax

```
duckdb_interval duckdb_value_interval(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The duckdb_interval value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_varchar`

Syntax

```
char *duckdb_value_varchar(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- DEPRECATED

use duckdb_value_string instead. This function does not work correctly if the string contains null bytes.

- returns

The text value at the specified location as a null-terminated string, or nullptr if the value cannot be converted. The result must be freed with duckdb_free.

duckdb_value_varchar_internal

Syntax

```
char *duckdb_value_varchar_internal(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- DEPRECATED

use duckdb_value_string_internal instead. This function does not work correctly if the string contains null bytes.

- returns

The char* value at the specified location. ONLY works on VARCHAR columns and does not auto-cast. If the column is NOT a VARCHAR column this function will return NULL.

The result must NOT be freed.

duckdb_value_string_internal

Syntax

```
duckdb_string duckdb_value_string_internal(
  duckdb_result *result,
  idx_t col,
```

```
  idx_t row
);
```

Parameters

- DEPRECATED

use duckdb_value_string_internal instead. This function does not work correctly if the string contains null bytes.

- returns

The char* value at the specified location. ONLY works on VARCHAR columns and does not auto-cast. If the column is NOT a VARCHAR column this function will return NULL.

The result must NOT be freed.

duckdb_value_blob

Syntax

```
duckdb_blob duckdb_value_blob(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The duckdb_blob value at the specified location. Returns a blob with blob.data set to nullptr if the value cannot be converted. The resulting "blob.data" must be freed with `duckdb_free`.

duckdb_value_is_null

Syntax

```
bool duckdb_value_is_null(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

Returns true if the value at the specified index is NULL, and false otherwise.

`duckdb_from_date` Decompose a `duckdb_date` object into year, month and date (stored as `duckdb_date_struct`).

Syntax

```
duckdb_date_struct duckdb_from_date(
  duckdb_date date
);
```

Parameters

- `date`

The date object, as obtained from a `DUCKDB_TYPE_DATE` column.

- `returns`

The `duckdb_date_struct` with the decomposed elements.

`duckdb_to_date` Re-compose a `duckdb_date` from year, month and date (`duckdb_date_struct`).

Syntax

```
duckdb_date duckdb_to_date(
  duckdb_date_struct date
);
```

Parameters

- `date`

The year, month and date stored in a `duckdb_date_struct`.

- `returns`

The `duckdb_date` element.

`duckdb_from_time` Decompose a `duckdb_time` object into hour, minute, second and microsecond (stored as `duckdb_time_struct`).

Syntax

```
duckdb_time_struct duckdb_from_time(
  duckdb_time time
);
```

Parameters

- `time`

The time object, as obtained from a `DUCKDB_TYPE_TIME` column.

- `returns`

The `duckdb_time_struct` with the decomposed elements.

`duckdb_to_time` Re-compose a `duckdb_time` from hour, minute, second and microsecond (`duckdb_time_struct`).

Syntax

```
duckdb_time duckdb_to_time(
  duckdb_time_struct time
);
```

Parameters

- `time`

The hour, minute, second and microsecond in a `duckdb_time_struct`.

- `returns`

The `duckdb_time` element.

`duckdb_from_timestamp` Decompose a `duckdb_timestamp` object into a `duckdb_timestamp_struct`.

Syntax

```
duckdb_timestamp_struct duckdb_from_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- `ts`

The ts object, as obtained from a `DUCKDB_TYPE_TIMESTAMP` column.

- `returns`

The `duckdb_timestamp_struct` with the decomposed elements.

`duckdb_to_timestamp` Re-compose a `duckdb_timestamp` from a duckdb_timestamp_struct.

Syntax

```
duckdb_timestamp duckdb_to_timestamp(
  duckdb_timestamp_struct ts
);
```

Parameters

- `ts`

The de-composed elements in a `duckdb_timestamp_struct`.

- `returns`

The `duckdb_timestamp` element.

`duckdb_hugeint_to_double` Converts a duckdb_hugeint object (as obtained from a `DUCKDB_TYPE_HUGEINT` column) into a double.

Syntax

```
double duckdb_hugeint_to_double(
  duckdb_hugeint val
);
```

Parameters

- `val`

The hugeint value.

- `returns`

The converted `double` element.

`duckdb_double_to_hugeint` Converts a double value to a duckdb_hugeint object.

If the conversion fails because the double value is too big the result will be 0.

Syntax

```
duckdb_hugeint duckdb_double_to_hugeint(
  double val
);
```

Parameters

- `val`

The double value.

- `returns`

The converted `duckdb_hugeint` element.

`duckdb_double_to_decimal` Converts a double value to a duckdb_decimal object.

If the conversion fails because the double value is too big, or the width/scale are invalid the result will be 0.

Syntax

```
duckdb_decimal duckdb_double_to_decimal(
  double val,
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `val`

The double value.

- `returns`

The converted `duckdb_decimal` element.

`duckdb_decimal_to_double` Converts a duckdb_decimal object (as obtained from a `DUCKDB_TYPE_DECIMAL` column) into a double.

Syntax

```
double duckdb_decimal_to_double(
  duckdb_decimal val
);
```

Parameters

- `val`

The decimal value.

- `returns`

The converted `double` element.

`duckdb_create_logical_type` Creates a `duckdb_logical_type` from a standard primitive type. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

This should not be used with `DUCKDB_TYPE_DECIMAL`.

Syntax

```
duckdb_logical_type duckdb_create_logical_type(
  duckdb_type type
);
```

Parameters

- `type`

The primitive type to create.

- `returns`

The logical type.

`duckdb_create_list_type` Creates a list type from its child type. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_create_list_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The child type of list type to create.

- `returns`

The logical type.

`duckdb_create_map_type` Creates a map type from its key type and value type. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_create_map_type(
  duckdb_logical_type key_type,
  duckdb_logical_type value_type
);
```

Parameters

- `type`

The key type and value type of map type to create.

- `returns`

The logical type.

`duckdb_create_union_type` Creates a UNION type from the passed types array The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_create_union_type(
  duckdb_logical_type member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `types`

The array of types that the union should consist of.

- `type_amount`

The size of the types array.

- `returns`

The logical type.

`duckdb_create_struct_type` Creates a STRUCT type from the passed member name and type arrays. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_create_struct_type(
  duckdb_logical_type *member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_types`

The array of types that the struct should consist of.

- `member_names`

The array of names that the struct should consist of.

- `member_count`

The number of members that were specified for both arrays.

- `returns`

The logical type.

`duckdb_create_decimal_type` Creates a `duckdb_logical_type` of type decimal with the specified width and scale The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_create_decimal_type(
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `width`

The width of the decimal type

- `scale`

The scale of the decimal type

- `returns`

The logical type.

`duckdb_get_type_id` Retrieves the type class of a `duckdb_logical_type`.

Syntax

```
duckdb_type duckdb_get_type_id(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The type id

`duckdb_decimal_width` Retrieves the width of a decimal type.

Syntax

```
uint8_t duckdb_decimal_width(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The width of the decimal type

`duckdb_decimal_scale` Retrieves the scale of a decimal type.

Syntax

```
uint8_t duckdb_decimal_scale(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The scale of the decimal type

`duckdb_decimal_internal_type` Retrieves the internal storage type of a decimal type.

Syntax

```
duckdb_type duckdb_decimal_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The internal type of the decimal type

`duckdb_enum_internal_type` Retrieves the internal storage type of an enum type.

Syntax

```
duckdb_type duckdb_enum_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The internal type of the enum type

`duckdb_enum_dictionary_size` Retrieves the dictionary size of the enum type

Syntax

```
uint32_t duckdb_enum_dictionary_size(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The dictionary size of the enum type

`duckdb_enum_dictionary_value` Retrieves the dictionary value at the specified position from the enum.

The result must be freed with `duckdb_free`

Syntax

```
char *duckdb_enum_dictionary_value(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`

The logical type object

- `index`

The index in the dictionary

- `returns`

The string value of the enum type. Must be freed with `duckdb_free`.

`duckdb_list_type_child_type` Retrieves the child type of the given list type.

The result must be freed with `duckdb_destroy_logical_type`

Syntax

```
duckdb_logical_type duckdb_list_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The child type of the list type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_map_type_key_type` Retrieves the key type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`

Syntax

```
duckdb_logical_type duckdb_map_type_key_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The key type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_map_type_value_type` Retrieves the value type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`

Syntax

```
duckdb_logical_type duckdb_map_type_value_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The value type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_struct_type_child_count` Returns the number of children of a struct type.

Syntax

```
idx_t duckdb_struct_type_child_count(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The number of children of a struct type.

`duckdb_struct_type_child_name` Retrieves the name of the struct child.

The result must be freed with `duckdb_free`

Syntax

```
char *duckdb_struct_type_child_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`

The logical type object

- `index`

The child index

- `returns`

The name of the struct type. Must be freed with `duckdb_free`.

`duckdb_struct_type_child_type` Retrieves the child type of the given struct type at the specified index.

The result must be freed with `duckdb_destroy_logical_type`

Syntax

```
duckdb_logical_type duckdb_struct_type_child_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`

The logical type object

- `index`

The child index

- `returns`

The child type of the struct type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_union_type_member_count` Returns the number of members that the union type has.

Syntax

```
idx_t duckdb_union_type_member_count(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type (union) object

- `returns`

The number of members of a union type.

`duckdb_union_type_member_name` Retrieves the name of the union member.

The result must be freed with `duckdb_free`

Syntax

```
char *duckdb_union_type_member_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`

The logical type object

- `index`

The child index

- `returns`

The name of the union member. Must be freed with `duckdb_free`.

`duckdb_union_type_member_type` Retrieves the child type of the given union member at the specified index.

The result must be freed with `duckdb_destroy_logical_type`

Syntax

```
duckdb_logical_type duckdb_union_type_member_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`

The logical type object

- `index`

The child index

- `returns`

The child type of the union member. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_destroy_logical_type` Destroys the logical type and de-allocates all memory allocated for that type.

Syntax

```
void duckdb_destroy_logical_type(
  duckdb_logical_type *type
);
```

Parameters

- `type`

The logical type to destroy.

C API - Prepared Statements

A prepared statement is a parameterized query. The query is prepared with question marks (`?`) or dollar symbols (`$1`) indicating the parameters of the query. Values can then be bound to these parameters, after which the prepared statement can be executed using those parameters. A single query can be prepared once and executed many times.

Prepared statements are useful to:

- Easily supply parameters to functions while avoiding string concatenation/SQL injection attacks.
- Speeding up queries that will be executed many times with different parameters.

DuckDB supports prepared statements in the C API with the `duckdb_prepare` method. The `duckdb_bind` family of functions is used to supply values for subsequent execution of the prepared statement using `duckdb_execute_prepared`. After we are done with the prepared statement it can be cleaned up using the `duckdb_destroy_prepare` method.

Example

```
duckdb_prepared_statement stmt;
duckdb_result result;
if (duckdb_prepare(con, "INSERT INTO integers VALUES ($1, $2)", &stmt) ==
 ↪  DuckDBError) {
    // handle error
}

duckdb_bind_int32(stmt, 1, 42); // the parameter index starts counting at 1!
duckdb_bind_int32(stmt, 2, 43);
// NULL as second parameter means no result set is requested
duckdb_execute_prepared(stmt, NULL);
duckdb_destroy_prepare(&stmt);

// we can also query result sets using prepared statements
if (duckdb_prepare(con, "SELECT * FROM integers WHERE i = ?", &stmt) ==
 ↪  DuckDBError) {
    // handle error
}
duckdb_bind_int32(stmt, 1, 42);
duckdb_execute_prepared(stmt, &result);

// do something with result

// clean up
duckdb_destroy_result(&result);
duckdb_destroy_prepare(&stmt);
```

After calling `duckdb_prepare`, the prepared statement parameters can be inspected using `duckdb_nparams` and `duckdb_param_type`. In case the prepare fails, the error can be

obtained through `duckdb_prepare_error`.

It is not required that the `duckdb_bind` family of functions matches the prepared statement parameter type exactly. The values will be auto-cast to the required value as required. For example, calling `duckdb_bind_int8` on a parameter type of `DUCKDB_TYPE_INTEGER` will work as expected.

> **Note.** Do **not** use prepared statements to insert large amounts of data into DuckDB. Instead it is recommended to use the Appender.

API Reference

```
duckdb_state duckdb_prepare(duckdb_connection connection, const char *query,
 ↪  duckdb_prepared_statement *out_prepared_statement);
void duckdb_destroy_prepare(duckdb_prepared_statement *prepared_statement);
const char *duckdb_prepare_error(duckdb_prepared_statement prepared_
 ↪  statement);
idx_t duckdb_nparams(duckdb_prepared_statement prepared_statement);
const char *duckdb_parameter_name(duckdb_prepared_statement prepared_
 ↪  statement, idx_t index);
duckdb_type duckdb_param_type(duckdb_prepared_statement prepared_statement,
 ↪  idx_t param_idx);
duckdb_state duckdb_clear_bindings(duckdb_prepared_statement prepared_
 ↪  statement);
duckdb_state duckdb_bind_value(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, duckdb_value val);
duckdb_state duckdb_bind_parameter_index(duckdb_prepared_statement
 ↪  prepared_statement, idx_t *param_idx_out, const char *name);
duckdb_state duckdb_bind_boolean(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, bool val);
duckdb_state duckdb_bind_int8(duckdb_prepared_statement prepared_statement,
 ↪  idx_t param_idx, int8_t val);
duckdb_state duckdb_bind_int16(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, int16_t val);
duckdb_state duckdb_bind_int32(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, int32_t val);
duckdb_state duckdb_bind_int64(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, int64_t val);
duckdb_state duckdb_bind_hugeint(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, duckdb_hugeint val);
duckdb_state duckdb_bind_decimal(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, duckdb_decimal val);
duckdb_state duckdb_bind_uint8(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, uint8_t val);
```

```
duckdb_state duckdb_bind_uint16(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, uint16_t val);
duckdb_state duckdb_bind_uint32(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, uint32_t val);
duckdb_state duckdb_bind_uint64(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, uint64_t val);
duckdb_state duckdb_bind_float(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, float val);
duckdb_state duckdb_bind_double(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, double val);
duckdb_state duckdb_bind_date(duckdb_prepared_statement prepared_statement,
 ↪  idx_t param_idx, duckdb_date val);
duckdb_state duckdb_bind_time(duckdb_prepared_statement prepared_statement,
 ↪  idx_t param_idx, duckdb_time val);
duckdb_state duckdb_bind_timestamp(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, duckdb_timestamp val);
duckdb_state duckdb_bind_interval(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, duckdb_interval val);
duckdb_state duckdb_bind_varchar(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, const char *val);
duckdb_state duckdb_bind_varchar_length(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, const char *val, idx_t length);
duckdb_state duckdb_bind_blob(duckdb_prepared_statement prepared_statement,
 ↪  idx_t param_idx, const void *data, idx_t length);
duckdb_state duckdb_bind_null(duckdb_prepared_statement prepared_statement,
 ↪  idx_t param_idx);
duckdb_state duckdb_execute_prepared(duckdb_prepared_statement prepared_
 ↪  statement, duckdb_result *out_result);
duckdb_state duckdb_execute_prepared_arrow(duckdb_prepared_statement
 ↪  prepared_statement, duckdb_arrow *out_result);
duckdb_state duckdb_arrow_scan(duckdb_connection connection, const char
 ↪  *table_name, duckdb_arrow_stream arrow);
duckdb_state duckdb_arrow_array_scan(duckdb_connection connection, const
 ↪  char *table_name, duckdb_arrow_schema arrow_schema, duckdb_arrow_array
 ↪  arrow_array, duckdb_arrow_stream *out_stream);
```

`duckdb_prepare` Create a prepared statement object from a query.

Note that after calling `duckdb_prepare`, the prepared statement should always be destroyed using `duckdb_destroy_prepare`, even if the prepare fails.

If the prepare fails, `duckdb_prepare_error` can be called to obtain the reason why the prepare failed.

Syntax

```
duckdb_state duckdb_prepare(
  duckdb_connection connection,
  const char *query,
  duckdb_prepared_statement *out_prepared_statement
);
```

Parameters

- `connection`

The connection object

- `query`

The SQL query to prepare

- `out_prepared_statement`

The resulting prepared statement object

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_prepare` Closes the prepared statement and de-allocates all memory allocated for the statement.

Syntax

```
void duckdb_destroy_prepare(
  duckdb_prepared_statement *prepared_statement
);
```

Parameters

- `prepared_statement`

The prepared statement to destroy.

`duckdb_prepare_error` Returns the error message associated with the given prepared statement. If the prepared statement has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_destroy_prepare` is called.

Syntax

```
const char *duckdb_prepare_error(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`

The prepared statement to obtain the error from.

- `returns`

The error message, or `nullptr` if there is none.

`duckdb_nparams` Returns the number of parameters that can be provided to the given prepared statement.

Returns 0 if the query was not successfully prepared.

Syntax

```
idx_t duckdb_nparams(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`

The prepared statement to obtain the number of parameters for.

`duckdb_parameter_name` Returns the name used to identify the parameter The returned string should be freed using `duckdb_free`.

Returns NULL if the index is out of range for the provided prepared statement.

Syntax

```
const char *duckdb_parameter_name(
  duckdb_prepared_statement prepared_statement,
  idx_t index
);
```

Parameters

- `prepared_statement`

The prepared statement for which to get the parameter name from.

`duckdb_param_type` Returns the parameter type for the parameter at the given index.

Returns `DUCKDB_TYPE_INVALID` if the parameter index is out of range or the statement was not successfully prepared.

Syntax

```
duckdb_type duckdb_param_type(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

Parameters

- `prepared_statement`

The prepared statement.

- `param_idx`

The parameter index.

- `returns`

The parameter type

`duckdb_clear_bindings` Clear the params bind to the prepared statement.

Syntax

```
duckdb_state duckdb_clear_bindings(
  duckdb_prepared_statement prepared_statement
);
```

`duckdb_bind_value` Binds a value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_value(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_value val
);
```

`duckdb_bind_parameter_index` Retrieve the index of the parameter for the prepared statement, identified by name

Syntax

```
duckdb_state duckdb_bind_parameter_index(
  duckdb_prepared_statement prepared_statement,
  idx_t *param_idx_out,
  const char *name
);
```

`duckdb_bind_boolean` Binds a bool value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_boolean(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  bool val
);
```

`duckdb_bind_int8` Binds an int8_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_int8(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int8_t val
);
```

`duckdb_bind_int16` Binds an int16_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_int16(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int16_t val
);
```

`duckdb_bind_int32` Binds an int32_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_int32(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int32_t val
);
```

`duckdb_bind_int64` Binds an int64_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_int64(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int64_t val
);
```

`duckdb_bind_hugeint` Binds a duckdb_hugeint value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_hugeint(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_hugeint val
);
```

`duckdb_bind_decimal` Binds a duckdb_decimal value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_decimal(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_decimal val
);
```

`duckdb_bind_uint8` Binds an uint8_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_uint8(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint8_t val
);
```

`duckdb_bind_uint16` Binds an uint16_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_uint16(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint16_t val
);
```

`duckdb_bind_uint32` Binds an uint32_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_uint32(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint32_t val
);
```

`duckdb_bind_uint64` Binds an uint64_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_uint64(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint64_t val
);
```

`duckdb_bind_float` Binds a float value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_float(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  float val
);
```

`duckdb_bind_double` Binds a double value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_double(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
```

```
  double val
);
```

duckdb_bind_date Binds a duckdb_date value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_date(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_date val
);
```

duckdb_bind_time Binds a duckdb_time value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_time(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_time val
);
```

duckdb_bind_timestamp Binds a duckdb_timestamp value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_timestamp(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_timestamp val
);
```

duckdb_bind_interval Binds a duckdb_interval value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_interval(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_interval val
);
```

`duckdb_bind_varchar` Binds a null-terminated varchar value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_varchar(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const char *val
);
```

`duckdb_bind_varchar_length` Binds a varchar value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_varchar_length(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const char *val,
  idx_t length
);
```

`duckdb_bind_blob` Binds a blob value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_blob(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const void *data,
  idx_t length
);
```

`duckdb_bind_null` Binds a NULL value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_null(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

`duckdb_execute_prepared` Executes the prepared statement with the given bound parameters, and returns a materialized query result.

This method can be called multiple times for each prepared statement, and the parameters can be modified between calls to this function.

Syntax

```
duckdb_state duckdb_execute_prepared(
  duckdb_prepared_statement prepared_statement,
  duckdb_result *out_result
);
```

Parameters

- `prepared_statement`

The prepared statement to execute.

- `out_result`

The query result.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_execute_prepared_arrow` Executes the prepared statement with the given bound parameters, and returns an arrow query result.

Syntax

```
duckdb_state duckdb_execute_prepared_arrow(
  duckdb_prepared_statement prepared_statement,
  duckdb_arrow *out_result
);
```

Parameters

- `prepared_statement`

The prepared statement to execute.

- `out_result`

The query result.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_arrow_scan` Scans the Arrow stream and creates a view with the given name.

Syntax

```
duckdb_state duckdb_arrow_scan(
  duckdb_connection connection,
  const char *table_name,
  duckdb_arrow_stream arrow
);
```

Parameters

- `connection`

The connection on which to execute the scan.

- `table_name`

Name of the temporary view to create.

- `arrow`

Arrow stream wrapper.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_arrow_array_scan` Scans the Arrow array and creates a view with the given name.

Syntax

```
duckdb_state duckdb_arrow_array_scan(
  duckdb_connection connection,
  const char *table_name,
  duckdb_arrow_schema arrow_schema,
  duckdb_arrow_array arrow_array,
  duckdb_arrow_stream *out_stream
);
```

Parameters

- `connection`

The connection on which to execute the scan.

- `table_name`

Name of the temporary view to create.

- `arrow_schema`

Arrow schema wrapper.

- `arrow_array`

Arrow array wrapper.

- `out_stream`

Output array stream that wraps around the passed schema, for releasing/deleting once done.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

C API - Appender

Appenders are the most efficient way of loading data into DuckDB from within the C interface, and are recommended for fast data loading. The appender is much faster than using prepared statements or individual `INSERT INTO` statements.

Appends are made in row-wise format. For every column, a `duckdb_append_[type]` call should be made, after which the row should be finished by calling `duckdb_appender_end_row`. After all rows have been appended, `duckdb_appender_destroy` should be used to finalize the appender and clean up the resulting memory.

Note that `duckdb_appender_destroy` should always be called on the resulting appender, even if the function returns `DuckDBError`.

Example

```
duckdb_query(con, "CREATE TABLE people(id INTEGER, name VARCHAR)", NULL);

duckdb_appender appender;
if (duckdb_appender_create(con, NULL, "people", &appender) == DuckDBError) {
  // handle error
}
// append the first row (1, Mark)
duckdb_append_int32(appender, 1);
duckdb_append_varchar(appender, "Mark");
duckdb_appender_end_row(appender);

// append the second row (2, Hannes)
duckdb_append_int32(appender, 2);
duckdb_append_varchar(appender, "Hannes");
duckdb_appender_end_row(appender);

// finish appending and flush all the rows to the table
duckdb_appender_destroy(&appender);
```

API Reference

```
duckdb_state duckdb_appender_create(duckdb_connection connection, const char
↪  *schema, const char *table, duckdb_appender *out_appender);
const char *duckdb_appender_error(duckdb_appender appender);
duckdb_state duckdb_appender_flush(duckdb_appender appender);
```

```
duckdb_state duckdb_appender_close(duckdb_appender appender);
duckdb_state duckdb_appender_destroy(duckdb_appender *appender);
duckdb_state duckdb_appender_begin_row(duckdb_appender appender);
duckdb_state duckdb_appender_end_row(duckdb_appender appender);
duckdb_state duckdb_append_bool(duckdb_appender appender, bool value);
duckdb_state duckdb_append_int8(duckdb_appender appender, int8_t value);
duckdb_state duckdb_append_int16(duckdb_appender appender, int16_t value);
duckdb_state duckdb_append_int32(duckdb_appender appender, int32_t value);
duckdb_state duckdb_append_int64(duckdb_appender appender, int64_t value);
duckdb_state duckdb_append_hugeint(duckdb_appender appender, duckdb_hugeint
 ↪  value);
duckdb_state duckdb_append_uint8(duckdb_appender appender, uint8_t value);
duckdb_state duckdb_append_uint16(duckdb_appender appender, uint16_t value);
duckdb_state duckdb_append_uint32(duckdb_appender appender, uint32_t value);
duckdb_state duckdb_append_uint64(duckdb_appender appender, uint64_t value);
duckdb_state duckdb_append_float(duckdb_appender appender, float value);
duckdb_state duckdb_append_double(duckdb_appender appender, double value);
duckdb_state duckdb_append_date(duckdb_appender appender, duckdb_date
 ↪  value);
duckdb_state duckdb_append_time(duckdb_appender appender, duckdb_time
 ↪  value);
duckdb_state duckdb_append_timestamp(duckdb_appender appender, duckdb_
 ↪  timestamp value);
duckdb_state duckdb_append_interval(duckdb_appender appender, duckdb_
 ↪  interval value);
duckdb_state duckdb_append_varchar(duckdb_appender appender, const char
 ↪  *val);
duckdb_state duckdb_append_varchar_length(duckdb_appender appender, const
 ↪  char *val, idx_t length);
duckdb_state duckdb_append_blob(duckdb_appender appender, const void *data,
 ↪  idx_t length);
duckdb_state duckdb_append_null(duckdb_appender appender);
duckdb_state duckdb_append_data_chunk(duckdb_appender appender, duckdb_data_
 ↪  chunk chunk);
```

`duckdb_appender_create` Creates an appender object.

Syntax

```
duckdb_state duckdb_appender_create(
  duckdb_connection connection,
  const char *schema,
```

```
  const char *table,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`

The connection context to create the appender in.

- `schema`

The schema of the table to append to, or `nullptr` for the default schema.

- `table`

The table name to append to.

- `out_appender`

The resulting appender object.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_error` Returns the error message associated with the given appender. If the appender has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_appender_destroy` is called.

Syntax

```
const char *duckdb_appender_error(
  duckdb_appender appender
);
```

Parameters

- `appender`

The appender to get the error from.

- `returns`

The error message, or `nullptr` if there is none.

`duckdb_appender_flush` Flush the appender to the table, forcing the cache of the appender to be cleared and the data to be appended to the base table.

This should generally not be used unless you know what you are doing. Instead, call `duckdb_appender_destroy` when you are done with the appender.

Syntax

```
duckdb_state duckdb_appender_flush(
  duckdb_appender appender
);
```

Parameters

- `appender`

The appender to flush.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_close` Close the appender, flushing all intermediate state in the appender to the table and closing it for further appends.

This is generally not necessary. Call `duckdb_appender_destroy` instead.

Syntax

```
duckdb_state duckdb_appender_close(
  duckdb_appender appender
);
```

Parameters

- `appender`

The appender to flush and close.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_destroy` Close the appender and destroy it. Flushing all intermediate state in the appender to the table, and de-allocating all memory associated with the appender.

Syntax

```
duckdb_state duckdb_appender_destroy(
  duckdb_appender *appender
);
```

Parameters

- `appender`

The appender to flush, close and destroy.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_begin_row` A nop function, provided for backwards compatibility reasons. Does nothing. Only `duckdb_appender_end_row` is required.

Syntax

```
duckdb_state duckdb_appender_begin_row(
  duckdb_appender appender
);
```

`duckdb_appender_end_row` Finish the current row of appends. After end_row is called, the next row can be appended.

Syntax

```
duckdb_state duckdb_appender_end_row(
  duckdb_appender appender
);
```

Parameters

- `appender`

The appender.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_append_bool` Append a bool value to the appender.

Syntax

```
duckdb_state duckdb_append_bool(
  duckdb_appender appender,
  bool value
);
```

`duckdb_append_int8` Append an int8_t value to the appender.

Syntax

```
duckdb_state duckdb_append_int8(
  duckdb_appender appender,
  int8_t value
);
```

`duckdb_append_int16` Append an int16_t value to the appender.

Syntax

```
duckdb_state duckdb_append_int16(
  duckdb_appender appender,
  int16_t value
);
```

`duckdb_append_int32` Append an int32_t value to the appender.

Syntax

```
duckdb_state duckdb_append_int32(
  duckdb_appender appender,
  int32_t value
);
```

`duckdb_append_int64` Append an int64_t value to the appender.

Syntax

```
duckdb_state duckdb_append_int64(
  duckdb_appender appender,
  int64_t value
);
```

`duckdb_append_hugeint` Append a duckdb_hugeint value to the appender.

Syntax

```
duckdb_state duckdb_append_hugeint(
  duckdb_appender appender,
  duckdb_hugeint value
);
```

`duckdb_append_uint8` Append a uint8_t value to the appender.

Syntax

```
duckdb_state duckdb_append_uint8(
  duckdb_appender appender,
  uint8_t value
);
```

`duckdb_append_uint16` Append a uint16_t value to the appender.

Syntax

```
duckdb_state duckdb_append_uint16(
  duckdb_appender appender,
  uint16_t value
);
```

`duckdb_append_uint32` Append a uint32_t value to the appender.

Syntax

```
duckdb_state duckdb_append_uint32(
  duckdb_appender appender,
  uint32_t value
);
```

`duckdb_append_uint64` Append a uint64_t value to the appender.

Syntax

```
duckdb_state duckdb_append_uint64(
  duckdb_appender appender,
  uint64_t value
);
```

`duckdb_append_float` Append a float value to the appender.

Syntax

```
duckdb_state duckdb_append_float(
  duckdb_appender appender,
  float value
);
```

`duckdb_append_double` Append a double value to the appender.

Syntax

```
duckdb_state duckdb_append_double(
  duckdb_appender appender,
  double value
);
```

`duckdb_append_date` Append a duckdb_date value to the appender.

Syntax

```
duckdb_state duckdb_append_date(
  duckdb_appender appender,
  duckdb_date value
);
```

`duckdb_append_time` Append a duckdb_time value to the appender.

Syntax

```
duckdb_state duckdb_append_time(
  duckdb_appender appender,
  duckdb_time value
);
```

`duckdb_append_timestamp` Append a duckdb_timestamp value to the appender.

Syntax

```
duckdb_state duckdb_append_timestamp(
  duckdb_appender appender,
  duckdb_timestamp value
);
```

`duckdb_append_interval` Append a duckdb_interval value to the appender.

Syntax

```
duckdb_state duckdb_append_interval(
  duckdb_appender appender,
  duckdb_interval value
);
```

`duckdb_append_varchar` Append a varchar value to the appender.

Syntax

```
duckdb_state duckdb_append_varchar(
  duckdb_appender appender,
  const char *val
);
```

`duckdb_append_varchar_length` Append a varchar value to the appender.

Syntax

```
duckdb_state duckdb_append_varchar_length(
  duckdb_appender appender,
  const char *val,
  idx_t length
);
```

`duckdb_append_blob` Append a blob value to the appender.

Syntax

```
duckdb_state duckdb_append_blob(
  duckdb_appender appender,
  const void *data,
  idx_t length
);
```

`duckdb_append_null` Append a NULL value to the appender (of any type).

Syntax

```
duckdb_state duckdb_append_null(
  duckdb_appender appender
);
```

`duckdb_append_data_chunk` Appends a pre-filled data chunk to the specified appender.

The types of the data chunk must exactly match the types of the table, no casting is performed. If the types do not match or the appender is in an invalid state, DuckDBError is returned. If the append is successful, DuckDBSuccess is returned.

Syntax

```
duckdb_state duckdb_append_data_chunk(
  duckdb_appender appender,
  duckdb_data_chunk chunk
);
```

Parameters

- `appender`

The appender to append to.

- `chunk`

The data chunk to append.

- `returns`

The return state.

C API - Table Functions

The table function API can be used to define a table function that can then be called from within DuckDB in the `FROM` clause of a query.

API Reference

```
duckdb_table_function duckdb_create_table_function();
void duckdb_destroy_table_function(duckdb_table_function *table_function);
```

```
void duckdb_table_function_set_name(duckdb_table_function table_function,
 ↪  const char *name);
void duckdb_table_function_add_parameter(duckdb_table_function table_
 ↪  function, duckdb_logical_type type);
void duckdb_table_function_add_named_parameter(duckdb_table_function table_
 ↪  function, const char *name, duckdb_logical_type type);
void duckdb_table_function_set_extra_info(duckdb_table_function table_
 ↪  function, void *extra_info, duckdb_delete_callback_t destroy);
void duckdb_table_function_set_bind(duckdb_table_function table_function,
 ↪  duckdb_table_function_bind_t bind);
void duckdb_table_function_set_init(duckdb_table_function table_function,
 ↪  duckdb_table_function_init_t init);
void duckdb_table_function_set_local_init(duckdb_table_function table_
 ↪  function, duckdb_table_function_init_t init);
void duckdb_table_function_set_function(duckdb_table_function table_
 ↪  function, duckdb_table_function_t function);
void duckdb_table_function_supports_projection_pushdown(duckdb_table_
 ↪  function table_function, bool pushdown);
duckdb_state duckdb_register_table_function(duckdb_connection con, duckdb_
 ↪  table_function function);
```

Table Function Bind

```
void *duckdb_bind_get_extra_info(duckdb_bind_info info);
void duckdb_bind_add_result_column(duckdb_bind_info info, const char *name,
 ↪  duckdb_logical_type type);
idx_t duckdb_bind_get_parameter_count(duckdb_bind_info info);
duckdb_value duckdb_bind_get_parameter(duckdb_bind_info info, idx_t index);
duckdb_value duckdb_bind_get_named_parameter(duckdb_bind_info info, const
 ↪  char *name);
void duckdb_bind_set_bind_data(duckdb_bind_info info, void *bind_data,
 ↪  duckdb_delete_callback_t destroy);
void duckdb_bind_set_cardinality(duckdb_bind_info info, idx_t cardinality,
 ↪  bool is_exact);
void duckdb_bind_set_error(duckdb_bind_info info, const char *error);
```

Table Function Init

```
void *duckdb_init_get_extra_info(duckdb_init_info info);
void *duckdb_init_get_bind_data(duckdb_init_info info);
void duckdb_init_set_init_data(duckdb_init_info info, void *init_data,
 ↪  duckdb_delete_callback_t destroy);
idx_t duckdb_init_get_column_count(duckdb_init_info info);
```

```
idx_t duckdb_init_get_column_index(duckdb_init_info info, idx_t column_
↪  index);
void duckdb_init_set_max_threads(duckdb_init_info info, idx_t max_threads);
void duckdb_init_set_error(duckdb_init_info info, const char *error);
```

Table Function

```
void *duckdb_function_get_extra_info(duckdb_function_info info);
void *duckdb_function_get_bind_data(duckdb_function_info info);
void *duckdb_function_get_init_data(duckdb_function_info info);
void *duckdb_function_get_local_init_data(duckdb_function_info info);
void duckdb_function_set_error(duckdb_function_info info, const char
↪  *error);
```

`duckdb_create_table_function` Creates a new empty table function.

The return value should be destroyed with `duckdb_destroy_table_function`.

Syntax

```
duckdb_table_function duckdb_create_table_function(

);
```

Parameters

- `returns`

The table function object.

`duckdb_destroy_table_function` Destroys the given table function object.

Syntax

```
void duckdb_destroy_table_function(
  duckdb_table_function *table_function
);
```

Parameters

- `table_function`

The table function to destroy

`duckdb_table_function_set_name` Sets the name of the given table function.

Syntax

```
void duckdb_table_function_set_name(
  duckdb_table_function table_function,
  const char *name
);
```

Parameters

- `table_function`

The table function

- `name`

The name of the table function

`duckdb_table_function_add_parameter` Adds a parameter to the table function.

Syntax

```
void duckdb_table_function_add_parameter(
  duckdb_table_function table_function,
  duckdb_logical_type type
);
```

Parameters

- `table_function`

The table function

- `type`

The type of the parameter to add.

`duckdb_table_function_add_named_parameter` Adds a named parameter to the table function.

Syntax

```
void duckdb_table_function_add_named_parameter(
  duckdb_table_function table_function,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `table_function`

The table function

- `name`

The name of the parameter

- `type`

The type of the parameter to add.

`duckdb_table_function_set_extra_info` Assigns extra information to the table function that can be fetched during binding, etc.

Syntax

```
void duckdb_table_function_set_extra_info(
  duckdb_table_function table_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `table_function`

The table function

- `extra_info`

The extra information

- `destroy`

The callback that will be called to destroy the bind data (if any)

`duckdb_table_function_set_bind` Sets the bind function of the table function

Syntax

```
void duckdb_table_function_set_bind(
  duckdb_table_function table_function,
  duckdb_table_function_bind_t bind
);
```

Parameters

- `table_function`

The table function

- `bind`

The bind function

`duckdb_table_function_set_init` Sets the init function of the table function

Syntax

```
void duckdb_table_function_set_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`

The table function

- `init`

The init function

`duckdb_table_function_set_local_init` Sets the thread-local init function of the table function

Syntax

```
void duckdb_table_function_set_local_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`

The table function

- `init`

The init function

`duckdb_table_function_set_function` Sets the main function of the table function

Syntax

```
void duckdb_table_function_set_function(
  duckdb_table_function table_function,
  duckdb_table_function_t function
);
```

Parameters

- `table_function`

The table function

- `function`

The function

`duckdb_table_function_supports_projection_pushdown` Sets whether or not the given table function supports projection pushdown.

If this is set to true, the system will provide a list of all required columns in the `init` stage through the `duckdb_init_get_column_count` and `duckdb_init_get_column_index` functions. If this is set to false (the default), the system will expect all columns to be projected.

Syntax

```
void duckdb_table_function_supports_projection_pushdown(
  duckdb_table_function table_function,
  bool pushdown
);
```

Parameters

- `table_function`

The table function

- `pushdown`

True if the table function supports projection pushdown, false otherwise.

`duckdb_register_table_function` Register the table function object within the given connection.

The function requires at least a name, a bind function, an init function and a main function.

If the function is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
duckdb_state duckdb_register_table_function(
  duckdb_connection con,
  duckdb_table_function function
);
```

Parameters

- `con`

The connection to register it in.

- `function`

The function pointer

- `returns`

Whether or not the registration was successful.

`duckdb_bind_get_extra_info` Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`

Syntax

```
void *duckdb_bind_get_extra_info(
  duckdb_bind_info info
);
```

Parameters

- `info`

The info object

- `returns`

The extra info

`duckdb_bind_add_result_column` Adds a result column to the output of the table function.

Syntax

```
void duckdb_bind_add_result_column(
  duckdb_bind_info info,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `info`

The info object

- `name`

The name of the column

- `type`

The logical type of the column

`duckdb_bind_get_parameter_count` Retrieves the number of regular (non-named) parameters to the function.

Syntax

```
idx_t duckdb_bind_get_parameter_count(
  duckdb_bind_info info
);
```

Parameters

- `info`

The info object

- `returns`

The number of parameters

`duckdb_bind_get_parameter` Retrieves the parameter at the given index.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_bind_get_parameter(
  duckdb_bind_info info,
  idx_t index
);
```

Parameters

- `info`

The info object

- `index`

The index of the parameter to get

- `returns`

The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_get_named_parameter` Retrieves a named parameter with the given name.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_bind_get_named_parameter(
  duckdb_bind_info info,
  const char *name
);
```

Parameters

- `info`

The info object

- `name`

The name of the parameter

- `returns`

The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_set_bind_data` Sets the user-provided bind data in the bind object. This object can be retrieved again during execution.

Syntax

```
void duckdb_bind_set_bind_data(
  duckdb_bind_info info,
  void *bind_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`

The info object

- `extra_data`

The bind data object.

- `destroy`

The callback that will be called to destroy the bind data (if any)

`duckdb_bind_set_cardinality` Sets the cardinality estimate for the table function, used for optimization.

Syntax

```
void duckdb_bind_set_cardinality(
  duckdb_bind_info info,
  idx_t cardinality,
  bool is_exact
);
```

Parameters

- `info`

The bind data object.

- `is_exact`

Whether or not the cardinality estimate is exact, or an approximation

`duckdb_bind_set_error` Report that an error has occurred while calling bind.

Syntax

```
void duckdb_bind_set_error(
  duckdb_bind_info info,
  const char *error
);
```

Parameters

- `info`

The info object

- `error`

The error message

`duckdb_init_get_extra_info` Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`

Syntax

```
void *duckdb_init_get_extra_info(
  duckdb_init_info info
);
```

Parameters

- `info`

The info object

- `returns`

The extra info

`duckdb_init_get_bind_data` Gets the bind data set by `duckdb_bind_set_bind_data` during the bind.

Note that the bind data should be considered as read-only. For tracking state, use the init data instead.

Syntax

```
void *duckdb_init_get_bind_data(
  duckdb_init_info info
);
```

Parameters

- `info`

The info object

- `returns`

The bind data object

`duckdb_init_set_init_data` Sets the user-provided init data in the init object. This object can be retrieved again during execution.

Syntax

```
void duckdb_init_set_init_data(
  duckdb_init_info info,
  void *init_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`

The info object

- `extra_data`

The init data object.

- `destroy`

The callback that will be called to destroy the init data (if any)

`duckdb_init_get_column_count` Returns the number of projected columns.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
idx_t duckdb_init_get_column_count(
  duckdb_init_info info
);
```

Parameters

- `info`

The info object

- `returns`

The number of projected columns.

`duckdb_init_get_column_index` Returns the column index of the projected column at the specified position.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
idx_t duckdb_init_get_column_index(
  duckdb_init_info info,
  idx_t column_index
);
```

Parameters

- `info`

The info object

- `column_index`

The index at which to get the projected column index, from 0..duckdb_init_get_column_count(info)

- `returns`

The column index of the projected column.

`duckdb_init_set_max_threads` Sets how many threads can process this table function in parallel (default: 1)

Syntax

```
void duckdb_init_set_max_threads(
  duckdb_init_info info,
  idx_t max_threads
);
```

Parameters

- `info`

The info object

- `max_threads`

The maximum amount of threads that can process this table function

`duckdb_init_set_error` Report that an error has occurred while calling init.

Syntax

```
void duckdb_init_set_error(
  duckdb_init_info info,
  const char *error
);
```

Parameters

- `info`

The info object

- `error`

The error message

`duckdb_function_get_extra_info` Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`

Syntax

```
void *duckdb_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`

The info object

- `returns`

The extra info

`duckdb_function_get_bind_data` Gets the bind data set by `duckdb_bind_set_bind_data` during the bind.

Note that the bind data should be considered as read-only. For tracking state, use the init data instead.

Syntax

```
void *duckdb_function_get_bind_data(
  duckdb_function_info info
);
```

Parameters

- `info`

The info object

- `returns`

The bind data object

`duckdb_function_get_init_data` Gets the init data set by `duckdb_init_set_init_data` during the init.

Syntax

```
void *duckdb_function_get_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`

The info object

- `returns`

The init data object

`duckdb_function_get_local_init_data` Gets the thread-local init data set by `duckdb_init_set_init_data` during the local_init.

Syntax

```
void *duckdb_function_get_local_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`

The info object

- `returns`

The init data object

`duckdb_function_set_error` Report that an error has occurred while executing the function.

Syntax

```
void duckdb_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`

The info object

- `error`

The error message

C API - Replacement Scans

The replacement scan API can be used to register a callback that is called when a table is read that does not exist in the catalog. For example, when a query such as `SELECT * FROM my_table` is executed and `my_table` does not exist, the replacement scan callback will be called with `my_table` as parameter. The replacement scan can then insert a table function with a specific parameter to replace the read of the table.

API Reference

```
void duckdb_add_replacement_scan(duckdb_database db, duckdb_replacement_
 ↪  callback_t replacement, void *extra_data, duckdb_delete_callback_t
 ↪  delete_callback);
void duckdb_replacement_scan_set_function_name(duckdb_replacement_scan_info
 ↪  info, const char *function_name);
void duckdb_replacement_scan_add_parameter(duckdb_replacement_scan_info
 ↪  info, duckdb_value parameter);
void duckdb_replacement_scan_set_error(duckdb_replacement_scan_info info,
 ↪  const char *error);
```

`duckdb_add_replacement_scan` Add a replacement scan definition to the specified database

Syntax

```
void duckdb_add_replacement_scan(
  duckdb_database db,
  duckdb_replacement_callback_t replacement,
  void *extra_data,
  duckdb_delete_callback_t delete_callback
);
```

Parameters

- db

The database object to add the replacement scan to

- replacement

The replacement scan callback

- extra_data

Extra data that is passed back into the specified callback

- delete_callback

The delete callback to call on the extra data, if any

`duckdb_replacement_scan_set_function_name` Sets the replacement function name to use. If this function is called in the replacement callback, the replacement scan is performed. If it is not called, the replacement callback is not performed.

Syntax

```
void duckdb_replacement_scan_set_function_name(
  duckdb_replacement_scan_info info,
  const char *function_name
);
```

Parameters

- `info`

The info object

- `function_name`

The function name to substitute.

`duckdb_replacement_scan_add_parameter` Adds a parameter to the replacement scan function.

Syntax

```
void duckdb_replacement_scan_add_parameter(
  duckdb_replacement_scan_info info,
  duckdb_value parameter
);
```

Parameters

- `info`

The info object

- `parameter`

The parameter to add.

`duckdb_replacement_scan_set_error` Report that an error has occurred while executing the replacement scan.

Syntax

```
void duckdb_replacement_scan_set_error(
  duckdb_replacement_scan_info info,
  const char *error
);
```

Parameters

- `info`

The info object

- `error`

The error message

C API - Complete API

API Reference

Open/Connect

```
duckdb_state duckdb_open(const char *path, duckdb_database *out_database);
duckdb_state duckdb_open_ext(const char *path, duckdb_database *out_
 ↪  database, duckdb_config config, char **out_error);
void duckdb_close(duckdb_database *database);
duckdb_state duckdb_connect(duckdb_database database, duckdb_connection
 ↪  *out_connection);
void duckdb_interrupt(duckdb_connection connection);
double duckdb_query_progress(duckdb_connection connection);
void duckdb_disconnect(duckdb_connection *connection);
const char *duckdb_library_version();
```

Configuration

```
duckdb_state duckdb_create_config(duckdb_config *out_config);
size_t duckdb_config_count();
duckdb_state duckdb_get_config_flag(size_t index, const char **out_name,
 ↪  const char **out_description);
duckdb_state duckdb_set_config(duckdb_config config, const char *name, const
 ↪  char *option);
void duckdb_destroy_config(duckdb_config *config);
```

Query Execution

```
duckdb_state duckdb_query(duckdb_connection connection, const char *query,
 ↪  duckdb_result *out_result);
void duckdb_destroy_result(duckdb_result *result);
const char *duckdb_column_name(duckdb_result *result, idx_t col);
duckdb_type duckdb_column_type(duckdb_result *result, idx_t col);
duckdb_logical_type duckdb_column_logical_type(duckdb_result *result, idx_t
 ↪  col);
idx_t duckdb_column_count(duckdb_result *result);
idx_t duckdb_row_count(duckdb_result *result);
idx_t duckdb_rows_changed(duckdb_result *result);
void *duckdb_column_data(duckdb_result *result, idx_t col);
bool *duckdb_nullmask_data(duckdb_result *result, idx_t col);
const char *duckdb_result_error(duckdb_result *result);
```

Result Functions

```
duckdb_data_chunk duckdb_result_get_chunk(duckdb_result result, idx_t chunk_
 ↪  index);
bool duckdb_result_is_streaming(duckdb_result result);
idx_t duckdb_result_chunk_count(duckdb_result result);
bool duckdb_value_boolean(duckdb_result *result, idx_t col, idx_t row);
int8_t duckdb_value_int8(duckdb_result *result, idx_t col, idx_t row);
int16_t duckdb_value_int16(duckdb_result *result, idx_t col, idx_t row);
int32_t duckdb_value_int32(duckdb_result *result, idx_t col, idx_t row);
int64_t duckdb_value_int64(duckdb_result *result, idx_t col, idx_t row);
duckdb_hugeint duckdb_value_hugeint(duckdb_result *result, idx_t col, idx_t
 ↪  row);
duckdb_decimal duckdb_value_decimal(duckdb_result *result, idx_t col, idx_t
 ↪  row);
uint8_t duckdb_value_uint8(duckdb_result *result, idx_t col, idx_t row);
uint16_t duckdb_value_uint16(duckdb_result *result, idx_t col, idx_t row);
uint32_t duckdb_value_uint32(duckdb_result *result, idx_t col, idx_t row);
uint64_t duckdb_value_uint64(duckdb_result *result, idx_t col, idx_t row);
float duckdb_value_float(duckdb_result *result, idx_t col, idx_t row);
double duckdb_value_double(duckdb_result *result, idx_t col, idx_t row);
duckdb_date duckdb_value_date(duckdb_result *result, idx_t col, idx_t row);
duckdb_time duckdb_value_time(duckdb_result *result, idx_t col, idx_t row);
duckdb_timestamp duckdb_value_timestamp(duckdb_result *result, idx_t col,
 ↪  idx_t row);
duckdb_interval duckdb_value_interval(duckdb_result *result, idx_t col, idx_
 ↪  t row);
```

```
char *duckdb_value_varchar(duckdb_result *result, idx_t col, idx_t row);
char *duckdb_value_varchar_internal(duckdb_result *result, idx_t col, idx_t
 ↪  row);
duckdb_string duckdb_value_string_internal(duckdb_result *result, idx_t col,
 ↪  idx_t row);
duckdb_blob duckdb_value_blob(duckdb_result *result, idx_t col, idx_t row);
bool duckdb_value_is_null(duckdb_result *result, idx_t col, idx_t row);
```

Helpers

```
void *duckdb_malloc(size_t size);
void duckdb_free(void *ptr);
idx_t duckdb_vector_size();
bool duckdb_string_is_inlined(duckdb_string_t string);
```

Date/Time/Timestamp Helpers

```
duckdb_date_struct duckdb_from_date(duckdb_date date);
duckdb_date duckdb_to_date(duckdb_date_struct date);
duckdb_time_struct duckdb_from_time(duckdb_time time);
duckdb_time duckdb_to_time(duckdb_time_struct time);
duckdb_timestamp_struct duckdb_from_timestamp(duckdb_timestamp ts);
duckdb_timestamp duckdb_to_timestamp(duckdb_timestamp_struct ts);
```

Hugeint Helpers

```
double duckdb_hugeint_to_double(duckdb_hugeint val);
duckdb_hugeint duckdb_double_to_hugeint(double val);
duckdb_decimal duckdb_double_to_decimal(double val, uint8_t width, uint8_t
 ↪  scale);
```

Decimal Helpers

```
double duckdb_decimal_to_double(duckdb_decimal val);
```

Prepared Statements

```
duckdb_state duckdb_prepare(duckdb_connection connection, const char *query,
 ↪  duckdb_prepared_statement *out_prepared_statement);
void duckdb_destroy_prepare(duckdb_prepared_statement *prepared_statement);
const char *duckdb_prepare_error(duckdb_prepared_statement prepared_
 ↪  statement);
```

```
idx_t duckdb_nparams(duckdb_prepared_statement prepared_statement);
const char *duckdb_parameter_name(duckdb_prepared_statement prepared_
↪  statement, idx_t index);
duckdb_type duckdb_param_type(duckdb_prepared_statement prepared_statement,
↪  idx_t param_idx);
duckdb_state duckdb_clear_bindings(duckdb_prepared_statement prepared_
↪  statement);
duckdb_state duckdb_bind_value(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, duckdb_value val);
duckdb_state duckdb_bind_parameter_index(duckdb_prepared_statement
↪  prepared_statement, idx_t *param_idx_out, const char *name);
duckdb_state duckdb_bind_boolean(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, bool val);
duckdb_state duckdb_bind_int8(duckdb_prepared_statement prepared_statement,
↪  idx_t param_idx, int8_t val);
duckdb_state duckdb_bind_int16(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, int16_t val);
duckdb_state duckdb_bind_int32(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, int32_t val);
duckdb_state duckdb_bind_int64(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, int64_t val);
duckdb_state duckdb_bind_hugeint(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, duckdb_hugeint val);
duckdb_state duckdb_bind_decimal(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, duckdb_decimal val);
duckdb_state duckdb_bind_uint8(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, uint8_t val);
duckdb_state duckdb_bind_uint16(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, uint16_t val);
duckdb_state duckdb_bind_uint32(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, uint32_t val);
duckdb_state duckdb_bind_uint64(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, uint64_t val);
duckdb_state duckdb_bind_float(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, float val);
duckdb_state duckdb_bind_double(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, double val);
duckdb_state duckdb_bind_date(duckdb_prepared_statement prepared_statement,
↪  idx_t param_idx, duckdb_date val);
duckdb_state duckdb_bind_time(duckdb_prepared_statement prepared_statement,
↪  idx_t param_idx, duckdb_time val);
duckdb_state duckdb_bind_timestamp(duckdb_prepared_statement prepared_
↪  statement, idx_t param_idx, duckdb_timestamp val);
```

```
duckdb_state duckdb_bind_interval(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, duckdb_interval val);
duckdb_state duckdb_bind_varchar(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, const char *val);
duckdb_state duckdb_bind_varchar_length(duckdb_prepared_statement prepared_
 ↪  statement, idx_t param_idx, const char *val, idx_t length);
duckdb_state duckdb_bind_blob(duckdb_prepared_statement prepared_statement,
 ↪  idx_t param_idx, const void *data, idx_t length);
duckdb_state duckdb_bind_null(duckdb_prepared_statement prepared_statement,
 ↪  idx_t param_idx);
duckdb_state duckdb_execute_prepared(duckdb_prepared_statement prepared_
 ↪  statement, duckdb_result *out_result);
duckdb_state duckdb_execute_prepared_arrow(duckdb_prepared_statement
 ↪  prepared_statement, duckdb_arrow *out_result);
duckdb_state duckdb_arrow_scan(duckdb_connection connection, const char
 ↪  *table_name, duckdb_arrow_stream arrow);
duckdb_state duckdb_arrow_array_scan(duckdb_connection connection, const
 ↪  char *table_name, duckdb_arrow_schema arrow_schema, duckdb_arrow_array
 ↪  arrow_array, duckdb_arrow_stream *out_stream);
```

Extract Statements

```
idx_t duckdb_extract_statements(duckdb_connection connection, const char
 ↪  *query, duckdb_extracted_statements *out_extracted_statements);
duckdb_state duckdb_prepare_extracted_statement(duckdb_connection
 ↪  connection, duckdb_extracted_statements extracted_statements, idx_t
 ↪  index, duckdb_prepared_statement *out_prepared_statement);
const char *duckdb_extract_statements_error(duckdb_extracted_statements
 ↪  extracted_statements);
void duckdb_destroy_extracted(duckdb_extracted_statements *extracted_
 ↪  statements);
```

Pending Result Interface

```
duckdb_state duckdb_pending_prepared(duckdb_prepared_statement prepared_
 ↪  statement, duckdb_pending_result *out_result);
duckdb_state duckdb_pending_prepared_streaming(duckdb_prepared_statement
 ↪  prepared_statement, duckdb_pending_result *out_result);
void duckdb_destroy_pending(duckdb_pending_result *pending_result);
const char *duckdb_pending_error(duckdb_pending_result pending_result);
duckdb_pending_state duckdb_pending_execute_task(duckdb_pending_result
 ↪  pending_result);
```

```
duckdb_state duckdb_execute_pending(duckdb_pending_result pending_result,
 ↪  duckdb_result *out_result);
bool duckdb_pending_execution_is_finished(duckdb_pending_state pending_
 ↪  state);
```

Value Interface

```
void duckdb_destroy_value(duckdb_value *value);
duckdb_value duckdb_create_varchar(const char *text);
duckdb_value duckdb_create_varchar_length(const char *text, idx_t length);
duckdb_value duckdb_create_int64(int64_t val);
char *duckdb_get_varchar(duckdb_value value);
int64_t duckdb_get_int64(duckdb_value value);
```

Logical Type Interface

```
duckdb_logical_type duckdb_create_logical_type(duckdb_type type);
duckdb_logical_type duckdb_create_list_type(duckdb_logical_type type);
duckdb_logical_type duckdb_create_map_type(duckdb_logical_type key_type,
 ↪  duckdb_logical_type value_type);
duckdb_logical_type duckdb_create_union_type(duckdb_logical_type member_
 ↪  types, const char **member_names, idx_t member_count);
duckdb_logical_type duckdb_create_struct_type(duckdb_logical_type *member_
 ↪  types, const char **member_names, idx_t member_count);
duckdb_logical_type duckdb_create_decimal_type(uint8_t width, uint8_t
 ↪  scale);
duckdb_type duckdb_get_type_id(duckdb_logical_type type);
uint8_t duckdb_decimal_width(duckdb_logical_type type);
uint8_t duckdb_decimal_scale(duckdb_logical_type type);
duckdb_type duckdb_decimal_internal_type(duckdb_logical_type type);
duckdb_type duckdb_enum_internal_type(duckdb_logical_type type);
uint32_t duckdb_enum_dictionary_size(duckdb_logical_type type);
char *duckdb_enum_dictionary_value(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_list_type_child_type(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_key_type(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_value_type(duckdb_logical_type type);
idx_t duckdb_struct_type_child_count(duckdb_logical_type type);
char *duckdb_struct_type_child_name(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_struct_type_child_type(duckdb_logical_type type,
 ↪  idx_t index);
idx_t duckdb_union_type_member_count(duckdb_logical_type type);
char *duckdb_union_type_member_name(duckdb_logical_type type, idx_t index);
```

```
duckdb_logical_type duckdb_union_type_member_type(duckdb_logical_type type,
 ↪  idx_t index);
void duckdb_destroy_logical_type(duckdb_logical_type *type);
```

Data Chunk Interface

```
duckdb_data_chunk duckdb_create_data_chunk(duckdb_logical_type *types, idx_t
 ↪  column_count);
void duckdb_destroy_data_chunk(duckdb_data_chunk *chunk);
void duckdb_data_chunk_reset(duckdb_data_chunk chunk);
idx_t duckdb_data_chunk_get_column_count(duckdb_data_chunk chunk);
duckdb_vector duckdb_data_chunk_get_vector(duckdb_data_chunk chunk, idx_t
 ↪  col_idx);
idx_t duckdb_data_chunk_get_size(duckdb_data_chunk chunk);
void duckdb_data_chunk_set_size(duckdb_data_chunk chunk, idx_t size);
```

Vector Interface

```
duckdb_logical_type duckdb_vector_get_column_type(duckdb_vector vector);
void *duckdb_vector_get_data(duckdb_vector vector);
uint64_t *duckdb_vector_get_validity(duckdb_vector vector);
void duckdb_vector_ensure_validity_writable(duckdb_vector vector);
void duckdb_vector_assign_string_element(duckdb_vector vector, idx_t index,
 ↪  const char *str);
void duckdb_vector_assign_string_element_len(duckdb_vector vector, idx_t
 ↪  index, const char *str, idx_t str_len);
duckdb_vector duckdb_list_vector_get_child(duckdb_vector vector);
idx_t duckdb_list_vector_get_size(duckdb_vector vector);
duckdb_state duckdb_list_vector_set_size(duckdb_vector vector, idx_t size);
duckdb_state duckdb_list_vector_reserve(duckdb_vector vector, idx_t
 ↪  required_capacity);
duckdb_vector duckdb_struct_vector_get_child(duckdb_vector vector, idx_t
 ↪  index);
```

Validity Mask Functions

```
bool duckdb_validity_row_is_valid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_validity(uint64_t *validity, idx_t row, bool
 ↪  valid);
void duckdb_validity_set_row_invalid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_valid(uint64_t *validity, idx_t row);
```

Table Functions

```
duckdb_table_function duckdb_create_table_function();
void duckdb_destroy_table_function(duckdb_table_function *table_function);
void duckdb_table_function_set_name(duckdb_table_function table_function,
 ↪  const char *name);
void duckdb_table_function_add_parameter(duckdb_table_function table_
 ↪  function, duckdb_logical_type type);
void duckdb_table_function_add_named_parameter(duckdb_table_function table_
 ↪  function, const char *name, duckdb_logical_type type);
void duckdb_table_function_set_extra_info(duckdb_table_function table_
 ↪  function, void *extra_info, duckdb_delete_callback_t destroy);
void duckdb_table_function_set_bind(duckdb_table_function table_function,
 ↪  duckdb_table_function_bind_t bind);
void duckdb_table_function_set_init(duckdb_table_function table_function,
 ↪  duckdb_table_function_init_t init);
void duckdb_table_function_set_local_init(duckdb_table_function table_
 ↪  function, duckdb_table_function_init_t init);
void duckdb_table_function_set_function(duckdb_table_function table_
 ↪  function, duckdb_table_function_t function);
void duckdb_table_function_supports_projection_pushdown(duckdb_table_
 ↪  function table_function, bool pushdown);
duckdb_state duckdb_register_table_function(duckdb_connection con, duckdb_
 ↪  table_function function);
```

Table Function Bind

```
void *duckdb_bind_get_extra_info(duckdb_bind_info info);
void duckdb_bind_add_result_column(duckdb_bind_info info, const char *name,
 ↪  duckdb_logical_type type);
idx_t duckdb_bind_get_parameter_count(duckdb_bind_info info);
duckdb_value duckdb_bind_get_parameter(duckdb_bind_info info, idx_t index);
duckdb_value duckdb_bind_get_named_parameter(duckdb_bind_info info, const
 ↪  char *name);
void duckdb_bind_set_bind_data(duckdb_bind_info info, void *bind_data,
 ↪  duckdb_delete_callback_t destroy);
void duckdb_bind_set_cardinality(duckdb_bind_info info, idx_t cardinality,
 ↪  bool is_exact);
void duckdb_bind_set_error(duckdb_bind_info info, const char *error);
```

Table Function Init

```
void *duckdb_init_get_extra_info(duckdb_init_info info);
void *duckdb_init_get_bind_data(duckdb_init_info info);
```

```
void duckdb_init_set_init_data(duckdb_init_info info, void *init_data,
 ↪  duckdb_delete_callback_t destroy);
idx_t duckdb_init_get_column_count(duckdb_init_info info);
idx_t duckdb_init_get_column_index(duckdb_init_info info, idx_t column_
 ↪  index);
void duckdb_init_set_max_threads(duckdb_init_info info, idx_t max_threads);
void duckdb_init_set_error(duckdb_init_info info, const char *error);
```

Table Function

```
void *duckdb_function_get_extra_info(duckdb_function_info info);
void *duckdb_function_get_bind_data(duckdb_function_info info);
void *duckdb_function_get_init_data(duckdb_function_info info);
void *duckdb_function_get_local_init_data(duckdb_function_info info);
void duckdb_function_set_error(duckdb_function_info info, const char
 ↪  *error);
```

Replacement Scans

```
void duckdb_add_replacement_scan(duckdb_database db, duckdb_replacement_
 ↪  callback_t replacement, void *extra_data, duckdb_delete_callback_t
 ↪  delete_callback);
void duckdb_replacement_scan_set_function_name(duckdb_replacement_scan_info
 ↪  info, const char *function_name);
void duckdb_replacement_scan_add_parameter(duckdb_replacement_scan_info
 ↪  info, duckdb_value parameter);
void duckdb_replacement_scan_set_error(duckdb_replacement_scan_info info,
 ↪  const char *error);
```

Appender

```
duckdb_state duckdb_appender_create(duckdb_connection connection, const char
 ↪  *schema, const char *table, duckdb_appender *out_appender);
const char *duckdb_appender_error(duckdb_appender appender);
duckdb_state duckdb_appender_flush(duckdb_appender appender);
duckdb_state duckdb_appender_close(duckdb_appender appender);
duckdb_state duckdb_appender_destroy(duckdb_appender *appender);
duckdb_state duckdb_appender_begin_row(duckdb_appender appender);
duckdb_state duckdb_appender_end_row(duckdb_appender appender);
duckdb_state duckdb_append_bool(duckdb_appender appender, bool value);
duckdb_state duckdb_append_int8(duckdb_appender appender, int8_t value);
duckdb_state duckdb_append_int16(duckdb_appender appender, int16_t value);
duckdb_state duckdb_append_int32(duckdb_appender appender, int32_t value);
```

```
duckdb_state duckdb_append_int64(duckdb_appender appender, int64_t value);
duckdb_state duckdb_append_hugeint(duckdb_appender appender, duckdb_hugeint
 ↪  value);
duckdb_state duckdb_append_uint8(duckdb_appender appender, uint8_t value);
duckdb_state duckdb_append_uint16(duckdb_appender appender, uint16_t value);
duckdb_state duckdb_append_uint32(duckdb_appender appender, uint32_t value);
duckdb_state duckdb_append_uint64(duckdb_appender appender, uint64_t value);
duckdb_state duckdb_append_float(duckdb_appender appender, float value);
duckdb_state duckdb_append_double(duckdb_appender appender, double value);
duckdb_state duckdb_append_date(duckdb_appender appender, duckdb_date
 ↪  value);
duckdb_state duckdb_append_time(duckdb_appender appender, duckdb_time
 ↪  value);
duckdb_state duckdb_append_timestamp(duckdb_appender appender, duckdb_
 ↪  timestamp value);
duckdb_state duckdb_append_interval(duckdb_appender appender, duckdb_
 ↪  interval value);
duckdb_state duckdb_append_varchar(duckdb_appender appender, const char
 ↪  *val);
duckdb_state duckdb_append_varchar_length(duckdb_appender appender, const
 ↪  char *val, idx_t length);
duckdb_state duckdb_append_blob(duckdb_appender appender, const void *data,
 ↪  idx_t length);
duckdb_state duckdb_append_null(duckdb_appender appender);
duckdb_state duckdb_append_data_chunk(duckdb_appender appender, duckdb_data_
 ↪  chunk chunk);
```

Arrow Interface

```
duckdb_state duckdb_query_arrow(duckdb_connection connection, const char
 ↪  *query, duckdb_arrow *out_result);
duckdb_state duckdb_query_arrow_schema(duckdb_arrow result, duckdb_arrow_
 ↪  schema *out_schema);
duckdb_state duckdb_prepared_arrow_schema(duckdb_prepared_statement
 ↪  prepared, duckdb_arrow_schema *out_schema);
duckdb_state duckdb_query_arrow_array(duckdb_arrow result, duckdb_arrow_
 ↪  array *out_array);
idx_t duckdb_arrow_column_count(duckdb_arrow result);
idx_t duckdb_arrow_row_count(duckdb_arrow result);
idx_t duckdb_arrow_rows_changed(duckdb_arrow result);
const char *duckdb_query_arrow_error(duckdb_arrow result);
void duckdb_destroy_arrow(duckdb_arrow *result);
```

Threading Information

```
void duckdb_execute_tasks(duckdb_database database, idx_t max_tasks);
duckdb_task_state duckdb_create_task_state(duckdb_database database);
void duckdb_execute_tasks_state(duckdb_task_state state);
idx_t duckdb_execute_n_tasks_state(duckdb_task_state state, idx_t max_
↪  tasks);
void duckdb_finish_execution(duckdb_task_state state);
bool duckdb_task_state_is_finished(duckdb_task_state state);
void duckdb_destroy_task_state(duckdb_task_state state);
bool duckdb_execution_is_finished(duckdb_connection con);
```

Streaming Result Interface

```
duckdb_data_chunk duckdb_stream_fetch_chunk(duckdb_result result);
```

duckdb_open Creates a new database or opens an existing database file stored at the given path. If no path is given a new in-memory database is created instead. The instantiated database should be closed with 'duckdb_close'

Syntax

```
duckdb_state duckdb_open(
  const char *path,
  duckdb_database *out_database
);
```

Parameters

- `path`

Path to the database file on disk, or `nullptr` or `:memory:` to open an in-memory database.

- `out_database`

The result database object.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_open_ext Extended version of duckdb_open. Creates a new database or opens an existing database file stored at the given path.

Syntax

```
duckdb_state duckdb_open_ext(
  const char *path,
  duckdb_database *out_database,
  duckdb_config config,
  char **out_error
);
```

Parameters

- `path`

Path to the database file on disk, or `nullptr` or `:memory:` to open an in-memory database.

- `out_database`

The result database object.

- `config`

(Optional) configuration used to start up the database system.

- `out_error`

If set and the function returns DuckDBError, this will contain the reason why the start-up failed. Note that the error must be freed using `duckdb_free`.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_close` Closes the specified database and de-allocates all memory allocated for that database. This should be called after you are done with any database allocated through `duckdb_open`. Note that failing to call `duckdb_close` (in case of e.g., a program crash) will not cause data corruption. Still it is recommended to always correctly close a database object after you are done with it.

Syntax

```
void duckdb_close(
  duckdb_database *database
);
```

Parameters

- `database`

The database object to shut down.

`duckdb_connect` Opens a connection to a database. Connections are required to query the database, and store transactional state associated with the connection. The instantiated connection should be closed using 'duckdb_disconnect'

Syntax

```
duckdb_state duckdb_connect(
  duckdb_database database,
  duckdb_connection *out_connection
);
```

Parameters

- `database`

The database file to connect to.

- `out_connection`

The result connection object.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_interrupt` Interrupt running query

Syntax

```
void duckdb_interrupt(
  duckdb_connection connection
);
```

Parameters

- `connection`

The connection to interruot

`duckdb_query_progress` Get progress of the running query

Syntax

```
double duckdb_query_progress(
  duckdb_connection connection
);
```

Parameters

- `connection`

The working connection

- `returns`

-1 if no progress or a percentage of the progress

`duckdb_disconnect` Closes the specified connection and de-allocates all memory allocated for that connection.

Syntax

```
void duckdb_disconnect(
  duckdb_connection *connection
);
```

Parameters

- `connection`

The connection to close.

`duckdb_library_version` Returns the version of the linked DuckDB, with a version postfix for dev versions

Usually used for developing C extensions that must return this for a compatibility check.

Syntax

```
const char *duckdb_library_version(

);
```

`duckdb_create_config` Initializes an empty configuration object that can be used to provide start-up options for the DuckDB instance through `duckdb_open_ext`.

This will always succeed unless there is a malloc failure.

Syntax

```
duckdb_state duckdb_create_config(
  duckdb_config *out_config
);
```

Parameters

- `out_config`

The result configuration object.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_config_count` This returns the total amount of configuration options available for usage with `duckdb_get_config_flag`.

This should not be called in a loop as it internally loops over all the options.

Syntax

```
size_t duckdb_config_count(

);
```

Parameters

- `returns`

The amount of config options available.

`duckdb_get_config_flag` Obtains a human-readable name and description of a specific configuration option. This can be used to e.g. display configuration options. This will succeed unless `index` is out of range (i.e., `>= duckdb_config_count`).

The result name or description MUST NOT be freed.

Syntax

```
duckdb_state duckdb_get_config_flag(
  size_t index,
  const char **out_name,
  const char **out_description
);
```

Parameters

- `index`

The index of the configuration option (between 0 and `duckdb_config_count`)

- `out_name`

A name of the configuration flag.

- `out_description`

A description of the configuration flag.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_set_config` Sets the specified option for the specified configuration. The configuration option is indicated by name. To obtain a list of config options, see `duckdb_get_config_flag`.

In the source code, configuration options are defined in `config.cpp`.

This can fail if either the name is invalid, or if the value provided for the option is invalid.

Syntax

```
duckdb_state duckdb_set_config(
  duckdb_config config,
  const char *name,
  const char *option
);
```

Parameters

- `duckdb_config`

The configuration object to set the option on.

- `name`

The name of the configuration flag to set.

- `option`

The value to set the configuration flag to.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_config` Destroys the specified configuration option and de-allocates all memory allocated for the object.

Syntax

```
void duckdb_destroy_config(
  duckdb_config *config
);
```

Parameters

- `config`

The configuration object to destroy.

`duckdb_query` Executes a SQL query within a connection and stores the full (materialized) result in the out_result pointer. If the query fails to execute, DuckDBError is returned and the error message can be retrieved by calling `duckdb_result_error`.

Note that after running `duckdb_query`, `duckdb_destroy_result` must be called on the result object even if the query fails, otherwise the error stored within the result will not be freed correctly.

Syntax

```
duckdb_state duckdb_query(
  duckdb_connection connection,
  const char *query,
  duckdb_result *out_result
);
```

Parameters

- `connection`

The connection to perform the query in.

- `query`

The SQL query to run.

- `out_result`

The query result.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_result` Closes the result and de-allocates all memory allocated for that connection.

Syntax

```
void duckdb_destroy_result(
  duckdb_result *result
);
```

Parameters

- `result`

The result to destroy.

`duckdb_column_name` Returns the column name of the specified column. The result should not need be freed; the column names will automatically be destroyed when the result is destroyed.

Returns `NULL` if the column is out of range.

Syntax

```
const char *duckdb_column_name(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`

The result object to fetch the column name from.

- `col`

The column index.

- `returns`

The column name of the specified column.

`duckdb_column_type` Returns the column type of the specified column.

Returns `DUCKDB_TYPE_INVALID` if the column is out of range.

Syntax

```
duckdb_type duckdb_column_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`

The result object to fetch the column type from.

- `col`

The column index.

- `returns`

The column type of the specified column.

`duckdb_column_logical_type` Returns the logical column type of the specified column.

The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Returns `NULL` if the column is out of range.

Syntax

```
duckdb_logical_type duckdb_column_logical_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`

The result object to fetch the column type from.

- `col`

The column index.

- `returns`

The logical column type of the specified column.

`duckdb_column_count` Returns the number of columns present in a the result object.

Syntax

```
idx_t duckdb_column_count(
  duckdb_result *result
);
```

Parameters

- `result`

The result object.

- `returns`

The number of columns present in the result object.

`duckdb_row_count` Returns the number of rows present in a the result object.

Syntax

```
idx_t duckdb_row_count(
  duckdb_result *result
);
```

Parameters

- `result`

The result object.

- `returns`

The number of rows present in the result object.

`duckdb_rows_changed` Returns the number of rows changed by the query stored in the result. This is relevant only for INSERT/UPDATE/DELETE queries. For other queries the rows_changed will be 0.

Syntax

```
idx_t duckdb_rows_changed(
  duckdb_result *result
);
```

Parameters

- `result`

The result object.

- `returns`

The number of rows changed.

duckdb_column_data **DEPRECATED**: Prefer using duckdb_result_get_chunk instead.

Returns the data of a specific column of a result in columnar format.

The function returns a dense array which contains the result data. The exact type stored in the array depends on the corresponding duckdb_type (as provided by duckdb_column_type). For the exact type by which the data should be accessed, see the comments in the types section or the DUCKDB_TYPE enum.

For example, for a column of type DUCKDB_TYPE_INTEGER, rows can be accessed in the following manner:

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
printf("Data for row %d: %d\n", row, data[row]);
```

Syntax

```
void *duckdb_column_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- result

The result object to fetch the column data from.

- col

The column index.

- returns

The column data of the specified column.

duckdb_nullmask_data **DEPRECATED**: Prefer using duckdb_result_get_chunk instead.

Returns the nullmask of a specific column of a result in columnar format. The nullmask indicates for every row whether or not the corresponding row is NULL. If a row is NULL, the values present in the array provided by duckdb_column_data are undefined.

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
bool *nullmask = duckdb_nullmask_data(&result, 0);
if (nullmask[row]) {
printf("Data for row %d: NULL\n", row);
} else {
printf("Data for row %d: %d\n", row, data[row]);
}
```

Syntax

```
bool *duckdb_nullmask_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`

The result object to fetch the nullmask from.

- `col`

The column index.

- `returns`

The nullmask of the specified column.

`duckdb_result_error` Returns the error message contained within the result. The error is only set if `duckdb_query` returns `DuckDBError`.

The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_result` is called.

Syntax

```
const char *duckdb_result_error(
  duckdb_result *result
);
```

Parameters

- `result`

The result object to fetch the error from.

- `returns`

The error of the result.

`duckdb_result_get_chunk` Fetches a data chunk from the duckdb_result. This function should be called repeatedly until the result is exhausted.

The result must be destroyed with `duckdb_destroy_data_chunk`.

This function supersedes all `duckdb_value` functions, as well as the `duckdb_column_data` and `duckdb_nullmask_data` functions. It results in significantly better performance, and should be preferred in newer code-bases.

If this function is used, none of the other result functions can be used and vice versa (i.e., this function cannot be mixed with the legacy result functions).

Use `duckdb_result_chunk_count` to figure out how many chunks there are in the result.

Syntax

```
duckdb_data_chunk duckdb_result_get_chunk(
  duckdb_result result,
  idx_t chunk_index
);
```

Parameters

- `result`

The result object to fetch the data chunk from.

- `chunk_index`

The chunk index to fetch from.

- `returns`

The resulting data chunk. Returns `NULL` if the chunk index is out of bounds.

`duckdb_result_is_streaming` Checks if the type of the internal result is StreamQueryResult.

Syntax

```
bool duckdb_result_is_streaming(
  duckdb_result result
);
```

Parameters

- `result`

The result object to check.

- `returns`

Whether or not the result object is of the type StreamQueryResult

`duckdb_result_chunk_count` Returns the number of data chunks present in the result.

Syntax

```
idx_t duckdb_result_chunk_count(
  duckdb_result result
);
```

Parameters

- `result`

The result object

- `returns`

Number of data chunks present in the result.

`duckdb_value_boolean`

Syntax

```
bool duckdb_value_boolean(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The boolean value at the specified location, or false if the value cannot be converted.

duckdb_value_int8

Syntax

```
int8_t duckdb_value_int8(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The int8_t value at the specified location, or 0 if the value cannot be converted.

duckdb_value_int16

Syntax

```
int16_t duckdb_value_int16(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The int16_t value at the specified location, or 0 if the value cannot be converted.

duckdb_value_int32

Syntax

```
int32_t duckdb_value_int32(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The int32_t value at the specified location, or 0 if the value cannot be converted.

duckdb_value_int64

Syntax

```
int64_t duckdb_value_int64(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The int64_t value at the specified location, or 0 if the value cannot be converted.

duckdb_value_hugeint

Syntax

```
duckdb_hugeint duckdb_value_hugeint(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The duckdb_hugeint value at the specified location, or 0 if the value cannot be converted.

duckdb_value_decimal

Syntax

```
duckdb_decimal duckdb_value_decimal(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The duckdb_decimal value at the specified location, or 0 if the value cannot be converted.

duckdb_value_uint8

Syntax

```
uint8_t duckdb_value_uint8(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The uint8_t value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_uint16`

Syntax

```
uint16_t duckdb_value_uint16(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The uint16_t value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_uint32`

Syntax

```
uint32_t duckdb_value_uint32(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The uint32_t value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_uint64`

Syntax

```
uint64_t duckdb_value_uint64(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The uint64_t value at the specified location, or 0 if the value cannot be converted.

duckdb_value_float

Syntax

```
float duckdb_value_float(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The float value at the specified location, or 0 if the value cannot be converted.

duckdb_value_double

Syntax

```
double duckdb_value_double(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The double value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_date`

Syntax

```
duckdb_date duckdb_value_date(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The duckdb_date value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_time`

Syntax

```
duckdb_time duckdb_value_time(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

The duckdb_time value at the specified location, or 0 if the value cannot be converted.

`duckdb_value_timestamp`

Syntax

```
duckdb_timestamp duckdb_value_timestamp(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The duckdb_timestamp value at the specified location, or 0 if the value cannot be converted.

duckdb_value_interval

Syntax

```
duckdb_interval duckdb_value_interval(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The duckdb_interval value at the specified location, or 0 if the value cannot be converted.

duckdb_value_varchar

Syntax

```
char *duckdb_value_varchar(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- DEPRECATED

use duckdb_value_string instead. This function does not work correctly if the string contains null bytes.

- returns

The text value at the specified location as a null-terminated string, or nullptr if the value cannot be converted. The result must be freed with duckdb_free.

duckdb_value_varchar_internal

Syntax

```
char *duckdb_value_varchar_internal(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- DEPRECATED

use duckdb_value_string_internal instead. This function does not work correctly if the string contains null bytes.

- returns

The char* value at the specified location. ONLY works on VARCHAR columns and does not auto-cast. If the column is NOT a VARCHAR column this function will return NULL.

The result must NOT be freed.

duckdb_value_string_internal

Syntax

```
duckdb_string duckdb_value_string_internal(
  duckdb_result *result,
  idx_t col,
```

```
  idx_t row
);
```

Parameters

- DEPRECATED

use duckdb_value_string_internal instead. This function does not work correctly if the string contains null bytes.

- returns

The char* value at the specified location. ONLY works on VARCHAR columns and does not auto-cast. If the column is NOT a VARCHAR column this function will return NULL.

The result must NOT be freed.

duckdb_value_blob

Syntax

```
duckdb_blob duckdb_value_blob(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The duckdb_blob value at the specified location. Returns a blob with blob.data set to nullptr if the value cannot be converted. The resulting "blob.data" must be freed with `duckdb_free`.

duckdb_value_is_null

Syntax

```
bool duckdb_value_is_null(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

Returns true if the value at the specified index is NULL, and false otherwise.

`duckdb_malloc` Allocate `size` bytes of memory using the duckdb internal malloc function. Any memory allocated in this manner should be freed using `duckdb_free`.

Syntax

```
void *duckdb_malloc(
  size_t size
);
```

Parameters

- `size`

The number of bytes to allocate.

- `returns`

A pointer to the allocated memory region.

`duckdb_free` Free a value returned from `duckdb_malloc`, `duckdb_value_varchar` or `duckdb_value_blob`.

Syntax

```
void duckdb_free(
  void *ptr
);
```

Parameters

- `ptr`

The memory region to de-allocate.

`duckdb_vector_size` The internal vector size used by DuckDB. This is the amount of tuples that will fit into a data chunk created by `duckdb_create_data_chunk`.

Syntax

```
idx_t duckdb_vector_size(

);
```

Parameters

- `returns`

The vector size.

`duckdb_string_is_inlined` Whether or not the duckdb_string_t value is inlined. This means that the data of the string does not have a separate allocation.

Syntax

```
bool duckdb_string_is_inlined(
  duckdb_string_t string
);
```

`duckdb_from_date` Decompose a `duckdb_date` object into year, month and date (stored as `duckdb_date_struct`).

Syntax

```
duckdb_date_struct duckdb_from_date(
  duckdb_date date
);
```

Parameters

- `date`

The date object, as obtained from a `DUCKDB_TYPE_DATE` column.

- `returns`

The `duckdb_date_struct` with the decomposed elements.

`duckdb_to_date` Re-compose a `duckdb_date` from year, month and date (`duckdb_date_struct`).

Syntax

```
duckdb_date duckdb_to_date(
  duckdb_date_struct date
);
```

Parameters

- `date`

The year, month and date stored in a `duckdb_date_struct`.

- `returns`

The `duckdb_date` element.

`duckdb_from_time` Decompose a `duckdb_time` object into hour, minute, second and microsecond (stored as `duckdb_time_struct`).

Syntax

```
duckdb_time_struct duckdb_from_time(
  duckdb_time time
);
```

Parameters

- `time`

The time object, as obtained from a `DUCKDB_TYPE_TIME` column.

- `returns`

The `duckdb_time_struct` with the decomposed elements.

`duckdb_to_time` Re-compose a `duckdb_time` from hour, minute, second and microsecond (`duckdb_time_struct`).

Syntax

```
duckdb_time duckdb_to_time(
  duckdb_time_struct time
);
```

Parameters

- `time`

The hour, minute, second and microsecond in a `duckdb_time_struct`.

- `returns`

The `duckdb_time` element.

`duckdb_from_timestamp` Decompose a `duckdb_timestamp` object into a `duckdb_timestamp_struct`.

Syntax

```
duckdb_timestamp_struct duckdb_from_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- `ts`

The ts object, as obtained from a `DUCKDB_TYPE_TIMESTAMP` column.

- `returns`

The `duckdb_timestamp_struct` with the decomposed elements.

`duckdb_to_timestamp` Re-compose a `duckdb_timestamp` from a duckdb_timestamp_struct.

Syntax

```
duckdb_timestamp duckdb_to_timestamp(
  duckdb_timestamp_struct ts
);
```

Parameters

- `ts`

The de-composed elements in a `duckdb_timestamp_struct`.

- `returns`

The `duckdb_timestamp` element.

`duckdb_hugeint_to_double` Converts a duckdb_hugeint object (as obtained from a `DUCKDB_TYPE_HUGEINT` column) into a double.

Syntax

```
double duckdb_hugeint_to_double(
  duckdb_hugeint val
);
```

Parameters

- `val`

The hugeint value.

- `returns`

The converted `double` element.

`duckdb_double_to_hugeint` Converts a double value to a duckdb_hugeint object.

If the conversion fails because the double value is too big the result will be 0.

Syntax

```
duckdb_hugeint duckdb_double_to_hugeint(
  double val
);
```

Parameters

- `val`

The double value.

- `returns`

The converted `duckdb_hugeint` element.

`duckdb_double_to_decimal` Converts a double value to a duckdb_decimal object.

If the conversion fails because the double value is too big, or the width/scale are invalid the result will be 0.

Syntax

```
duckdb_decimal duckdb_double_to_decimal(
  double val,
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `val`

The double value.

- `returns`

The converted `duckdb_decimal` element.

`duckdb_decimal_to_double` Converts a duckdb_decimal object (as obtained from a `DUCKDB_TYPE_DECIMAL` column) into a double.

Syntax

```
double duckdb_decimal_to_double(
  duckdb_decimal val
);
```

Parameters

- `val`

The decimal value.

- `returns`

The converted `double` element.

`duckdb_prepare` Create a prepared statement object from a query.

Note that after calling `duckdb_prepare`, the prepared statement should always be destroyed using `duckdb_destroy_prepare`, even if the prepare fails.

If the prepare fails, `duckdb_prepare_error` can be called to obtain the reason why the prepare failed.

Syntax

```
duckdb_state duckdb_prepare(
  duckdb_connection connection,
  const char *query,
  duckdb_prepared_statement *out_prepared_statement
);
```

Parameters

- `connection`

The connection object

- `query`

The SQL query to prepare

- `out_prepared_statement`

The resulting prepared statement object

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_prepare` Closes the prepared statement and de-allocates all memory allocated for the statement.

Syntax

```
void duckdb_destroy_prepare(
  duckdb_prepared_statement *prepared_statement
);
```

Parameters

- `prepared_statement`

The prepared statement to destroy.

`duckdb_prepare_error` Returns the error message associated with the given prepared statement. If the prepared statement has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_destroy_prepare` is called.

Syntax

```
const char *duckdb_prepare_error(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`

The prepared statement to obtain the error from.

- `returns`

The error message, or `nullptr` if there is none.

`duckdb_nparams` Returns the number of parameters that can be provided to the given prepared statement.

Returns 0 if the query was not successfully prepared.

Syntax

```
idx_t duckdb_nparams(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`

The prepared statement to obtain the number of parameters for.

`duckdb_parameter_name` Returns the name used to identify the parameter The returned string should be freed using `duckdb_free`.

Returns NULL if the index is out of range for the provided prepared statement.

Syntax

```
const char *duckdb_parameter_name(
  duckdb_prepared_statement prepared_statement,
  idx_t index
);
```

Parameters

- `prepared_statement`

The prepared statement for which to get the parameter name from.

`duckdb_param_type` Returns the parameter type for the parameter at the given index.

Returns `DUCKDB_TYPE_INVALID` if the parameter index is out of range or the statement was not successfully prepared.

Syntax

```
duckdb_type duckdb_param_type(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

Parameters

- `prepared_statement`

The prepared statement.

- `param_idx`

The parameter index.

- `returns`

The parameter type

`duckdb_clear_bindings` Clear the params bind to the prepared statement.

Syntax

```
duckdb_state duckdb_clear_bindings(
  duckdb_prepared_statement prepared_statement
);
```

`duckdb_bind_value` Binds a value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_value(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_value val
);
```

`duckdb_bind_parameter_index` Retrieve the index of the parameter for the prepared statement, identified by name

Syntax

```
duckdb_state duckdb_bind_parameter_index(
  duckdb_prepared_statement prepared_statement,
  idx_t *param_idx_out,
  const char *name
);
```

`duckdb_bind_boolean` Binds a bool value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_boolean(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  bool val
);
```

`duckdb_bind_int8` Binds an int8_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_int8(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int8_t val
);
```

`duckdb_bind_int16` Binds an int16_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_int16(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int16_t val
);
```

`duckdb_bind_int32` Binds an int32_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_int32(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int32_t val
);
```

`duckdb_bind_int64` Binds an int64_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_int64(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int64_t val
);
```

`duckdb_bind_hugeint` Binds a duckdb_hugeint value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_hugeint(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_hugeint val
);
```

`duckdb_bind_decimal` Binds a duckdb_decimal value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_decimal(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_decimal val
);
```

`duckdb_bind_uint8` Binds an uint8_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_uint8(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint8_t val
);
```

`duckdb_bind_uint16` Binds an uint16_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_uint16(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
```

```
  uint16_t val
);
```

`duckdb_bind_uint32` Binds an uint32_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_uint32(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint32_t val
);
```

`duckdb_bind_uint64` Binds an uint64_t value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_uint64(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint64_t val
);
```

`duckdb_bind_float` Binds a float value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_float(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  float val
);
```

`duckdb_bind_double` Binds a double value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_double(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  double val
);
```

`duckdb_bind_date` Binds a duckdb_date value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_date(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_date val
);
```

`duckdb_bind_time` Binds a duckdb_time value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_time(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_time val
);
```

`duckdb_bind_timestamp` Binds a duckdb_timestamp value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_timestamp(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_timestamp val
);
```

`duckdb_bind_interval` Binds a duckdb_interval value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_interval(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_interval val
);
```

`duckdb_bind_varchar` Binds a null-terminated varchar value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_varchar(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const char *val
);
```

`duckdb_bind_varchar_length` Binds a varchar value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_varchar_length(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const char *val,
  idx_t length
);
```

`duckdb_bind_blob` Binds a blob value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_blob(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const void *data,
  idx_t length
);
```

`duckdb_bind_null` Binds a NULL value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_null(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

`duckdb_execute_prepared` Executes the prepared statement with the given bound parameters, and returns a materialized query result.

This method can be called multiple times for each prepared statement, and the parameters can be modified between calls to this function.

Syntax

```
duckdb_state duckdb_execute_prepared(
  duckdb_prepared_statement prepared_statement,
  duckdb_result *out_result
);
```

Parameters

- `prepared_statement`

The prepared statement to execute.

- `out_result`

The query result.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_execute_prepared_arrow` Executes the prepared statement with the given bound parameters, and returns an arrow query result.

Syntax

```
duckdb_state duckdb_execute_prepared_arrow(
  duckdb_prepared_statement prepared_statement,
  duckdb_arrow *out_result
);
```

Parameters

- `prepared_statement`

The prepared statement to execute.

- `out_result`

The query result.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_arrow_scan` Scans the Arrow stream and creates a view with the given name.

Syntax

```
duckdb_state duckdb_arrow_scan(
  duckdb_connection connection,
  const char *table_name,
  duckdb_arrow_stream arrow
);
```

Parameters

- `connection`

The connection on which to execute the scan.

- `table_name`

Name of the temporary view to create.

- `arrow`

Arrow stream wrapper.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_arrow_array_scan` Scans the Arrow array and creates a view with the given name.

Syntax

```
duckdb_state duckdb_arrow_array_scan(
  duckdb_connection connection,
  const char *table_name,
  duckdb_arrow_schema arrow_schema,
  duckdb_arrow_array arrow_array,
  duckdb_arrow_stream *out_stream
);
```

Parameters

- `connection`

The connection on which to execute the scan.

- `table_name`

Name of the temporary view to create.

- `arrow_schema`

Arrow schema wrapper.

- `arrow_array`

Arrow array wrapper.

- `out_stream`

Output array stream that wraps around the passed schema, for releasing/deleting once done.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_extract_statements` Extract all statements from a query. Note that after calling `duckdb_extract_statements`, the extracted statements should always be destroyed using `duckdb_destroy_extracted`, even if no statements were extracted. If the extract fails, `duckdb_extract_statements_error` can be called to obtain the reason why the extract failed.

Syntax

```
idx_t duckdb_extract_statements(
  duckdb_connection connection,
  const char *query,
  duckdb_extracted_statements *out_extracted_statements
);
```

Parameters

- `connection`

The connection object

- `query`

The SQL query to extract

- `out_extracted_statements`

The resulting extracted statements object

- `returns`

The number of extracted statements or 0 on failure.

`duckdb_prepare_extracted_statement` Prepare an extracted statement. Note that after calling `duckdb_prepare_extracted_statement`, the prepared statement should always be destroyed using `duckdb_destroy_prepare`, even if the prepare fails. If the prepare fails, `duckdb_prepare_error` can be called to obtain the reason why the prepare failed.

Syntax

```
duckdb_state duckdb_prepare_extracted_statement(
  duckdb_connection connection,
  duckdb_extracted_statements extracted_statements,
  idx_t index,
```

```
  duckdb_prepared_statement *out_prepared_statement
);
```

Parameters

- `connection`

The connection object

- `extracted_statements`

The extracted statements object

- `index`

The index of the extracted statement to prepare

- `out_prepared_statement`

The resulting prepared statement object

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_extract_statements_error` Returns the error message contained within the extracted statements. The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_extracted` is called.

Syntax

```
const char *duckdb_extract_statements_error(
  duckdb_extracted_statements extracted_statements
);
```

Parameters

- `result`

The extracted statements to fetch the error from.

- `returns`

The error of the extracted statements.

`duckdb_destroy_extracted` De-allocates all memory allocated for the extracted statements.

Syntax

```
void duckdb_destroy_extracted(
  duckdb_extracted_statements *extracted_statements
);
```

Parameters

- `extracted_statements`

The extracted statements to destroy.

`duckdb_pending_prepared` Executes the prepared statement with the given bound parameters, and returns a pending result. The pending result represents an intermediate structure for a query that is not yet fully executed. The pending result can be used to incrementally execute a query, returning control to the client between tasks.

Note that after calling `duckdb_pending_prepared`, the pending result should always be destroyed using `duckdb_destroy_pending`, even if this function returns DuckDBError.

Syntax

```
duckdb_state duckdb_pending_prepared(
  duckdb_prepared_statement prepared_statement,
  duckdb_pending_result *out_result
);
```

Parameters

- `prepared_statement`

The prepared statement to execute.

- `out_result`

The pending query result.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_pending_prepared_streaming` Executes the prepared statement with the given bound parameters, and returns a pending result. This pending result will create a streaming duckdb_result when executed. The pending result represents an intermediate structure for a query that is not yet fully executed.

Note that after calling `duckdb_pending_prepared_streaming`, the pending result should always be destroyed using `duckdb_destroy_pending`, even if this function returns DuckDBError.

Syntax

```
duckdb_state duckdb_pending_prepared_streaming(
  duckdb_prepared_statement prepared_statement,
  duckdb_pending_result *out_result
);
```

Parameters

- `prepared_statement`

The prepared statement to execute.

- `out_result`

The pending query result.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_pending` Closes the pending result and de-allocates all memory allocated for the result.

Syntax

```
void duckdb_destroy_pending(
  duckdb_pending_result *pending_result
);
```

Parameters

- `pending_result`

The pending result to destroy.

`duckdb_pending_error` Returns the error message contained within the pending result.

The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_pending` is called.

Syntax

```
const char *duckdb_pending_error(
  duckdb_pending_result pending_result
);
```

Parameters

- `result`

The pending result to fetch the error from.

- `returns`

The error of the pending result.

`duckdb_pending_execute_task` Executes a single task within the query, returning whether or not the query is ready.

If this returns DUCKDB_PENDING_RESULT_READY, the duckdb_execute_pending function can be called to obtain the result. If this returns DUCKDB_PENDING_RESULT_NOT_READY, the duckdb_pending_execute_task function should be called again. If this returns DUCKDB_PENDING_ERROR, an error occurred during execution.

The error message can be obtained by calling duckdb_pending_error on the pending_result.

Syntax

```
duckdb_pending_state duckdb_pending_execute_task(
  duckdb_pending_result pending_result
);
```

Parameters

- `pending_result`

The pending result to execute a task within..

- `returns`

The state of the pending result after the execution.

`duckdb_execute_pending` Fully execute a pending query result, returning the final query result.

If duckdb_pending_execute_task has been called until DUCKDB_PENDING_RESULT_READY was returned, this will return fast. Otherwise, all remaining tasks must be executed first.

Syntax

```
duckdb_state duckdb_execute_pending(
  duckdb_pending_result pending_result,
  duckdb_result *out_result
);
```

Parameters

- `pending_result`

The pending result to execute.

- `out_result`

The result object.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_pending_execution_is_finished` Returns whether a duckdb_pending_state is finished executing. For example if `pending_state` is DUCKDB_PENDING_RESULT_READY, this function will return true.

Syntax

```
bool duckdb_pending_execution_is_finished(
  duckdb_pending_state pending_state
);
```

Parameters

- `pending_state`

The pending state on which to decide whether to finish execution.

- `returns`

Boolean indicating pending execution should be considered finished.

`duckdb_destroy_value` Destroys the value and de-allocates all memory allocated for that type.

Syntax

```
void duckdb_destroy_value(
  duckdb_value *value
);
```

Parameters

- `value`

The value to destroy.

`duckdb_create_varchar` Creates a value from a null-terminated string

Syntax

```
duckdb_value duckdb_create_varchar(
  const char *text
);
```

Parameters

- `value`

The null-terminated string

- `returns`

The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_varchar_length` Creates a value from a string

Syntax

```
duckdb_value duckdb_create_varchar_length(
  const char *text,
  idx_t length
);
```

Parameters

- `value`

The text

- `length`

The length of the text

- `returns`

The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int64` Creates a value from an int64

Syntax

```
duckdb_value duckdb_create_int64(
  int64_t val
);
```

Parameters

- `value`

The bigint value

- `returns`

The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_get_varchar` Obtains a string representation of the given value. The result must be destroyed with `duckdb_free`.

Syntax

```
char *duckdb_get_varchar(
  duckdb_value value
);
```

Parameters

- `value`

The value

- `returns`

The string value. This must be destroyed with `duckdb_free`.

`duckdb_get_int64` Obtains an int64 of the given value.

Syntax

```
int64_t duckdb_get_int64(
  duckdb_value value
);
```

Parameters

- `value`

The value

- `returns`

The int64 value, or 0 if no conversion is possible

`duckdb_create_logical_type` Creates a `duckdb_logical_type` from a standard primitive type. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

This should not be used with `DUCKDB_TYPE_DECIMAL`.

Syntax

```
duckdb_logical_type duckdb_create_logical_type(
  duckdb_type type
);
```

Parameters

- `type`

The primitive type to create.

- `returns`

The logical type.

`duckdb_create_list_type` Creates a list type from its child type. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_create_list_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The child type of list type to create.

- `returns`

The logical type.

`duckdb_create_map_type` Creates a map type from its key type and value type. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_create_map_type(
  duckdb_logical_type key_type,
  duckdb_logical_type value_type
);
```

Parameters

- `type`

The key type and value type of map type to create.

- `returns`

The logical type.

`duckdb_create_union_type` Creates a UNION type from the passed types array The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_create_union_type(
  duckdb_logical_type member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `types`

The array of types that the union should consist of.

- `type_amount`

The size of the types array.

- `returns`

The logical type.

`duckdb_create_struct_type` Creates a STRUCT type from the passed member name and type arrays. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_create_struct_type(
  duckdb_logical_type *member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_types`

The array of types that the struct should consist of.

- `member_names`

The array of names that the struct should consist of.

- `member_count`

The number of members that were specified for both arrays.

- `returns`

The logical type.

`duckdb_create_decimal_type` Creates a `duckdb_logical_type` of type decimal with the specified width and scale The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_create_decimal_type(
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `width`

The width of the decimal type

- `scale`

The scale of the decimal type

- `returns`

The logical type.

`duckdb_get_type_id` Retrieves the type class of a `duckdb_logical_type`.

Syntax

```
duckdb_type duckdb_get_type_id(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The type id

`duckdb_decimal_width` Retrieves the width of a decimal type.

Syntax

```
uint8_t duckdb_decimal_width(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The width of the decimal type

`duckdb_decimal_scale` Retrieves the scale of a decimal type.

Syntax

```
uint8_t duckdb_decimal_scale(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The scale of the decimal type

`duckdb_decimal_internal_type` Retrieves the internal storage type of a decimal type.

Syntax

```
duckdb_type duckdb_decimal_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The internal type of the decimal type

`duckdb_enum_internal_type` Retrieves the internal storage type of an enum type.

Syntax

```
duckdb_type duckdb_enum_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The internal type of the enum type

`duckdb_enum_dictionary_size` Retrieves the dictionary size of the enum type

Syntax

```
uint32_t duckdb_enum_dictionary_size(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The dictionary size of the enum type

`duckdb_enum_dictionary_value` Retrieves the dictionary value at the specified position from the enum.

The result must be freed with `duckdb_free`

Syntax

```
char *duckdb_enum_dictionary_value(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`

The logical type object

- `index`

The index in the dictionary

- `returns`

The string value of the enum type. Must be freed with `duckdb_free`.

`duckdb_list_type_child_type` Retrieves the child type of the given list type.

The result must be freed with `duckdb_destroy_logical_type`

Syntax

```
duckdb_logical_type duckdb_list_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The child type of the list type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_map_type_key_type` Retrieves the key type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`

Syntax

```
duckdb_logical_type duckdb_map_type_key_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The key type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_map_type_value_type` Retrieves the value type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`

Syntax

```
duckdb_logical_type duckdb_map_type_value_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The value type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_struct_type_child_count` Returns the number of children of a struct type.

Syntax

```
idx_t duckdb_struct_type_child_count(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The number of children of a struct type.

`duckdb_struct_type_child_name` Retrieves the name of the struct child.

The result must be freed with `duckdb_free`

Syntax

```
char *duckdb_struct_type_child_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`

The logical type object

- `index`

The child index

- `returns`

The name of the struct type. Must be freed with `duckdb_free`.

`duckdb_struct_type_child_type` Retrieves the child type of the given struct type at the specified index.

The result must be freed with `duckdb_destroy_logical_type`

Syntax

```
duckdb_logical_type duckdb_struct_type_child_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`

The logical type object

- `index`

The child index

- `returns`

The child type of the struct type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_union_type_member_count` Returns the number of members that the union type has.

Syntax

```
idx_t duckdb_union_type_member_count(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type (union) object

- `returns`

The number of members of a union type.

`duckdb_union_type_member_name` Retrieves the name of the union member.

The result must be freed with `duckdb_free`

Syntax

```
char *duckdb_union_type_member_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`

The logical type object

- `index`

The child index

- `returns`

The name of the union member. Must be freed with `duckdb_free`.

`duckdb_union_type_member_type` Retrieves the child type of the given union member at the specified index.

The result must be freed with `duckdb_destroy_logical_type`

Syntax

```
duckdb_logical_type duckdb_union_type_member_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`

The logical type object

- `index`

The child index

- `returns`

The child type of the union member. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_destroy_logical_type` Destroys the logical type and de-allocates all memory allocated for that type.

Syntax

```
void duckdb_destroy_logical_type(
  duckdb_logical_type *type
);
```

Parameters

- `type`

The logical type to destroy.

`duckdb_create_data_chunk` Creates an empty DataChunk with the specified set of types.

Syntax

```
duckdb_data_chunk duckdb_create_data_chunk(
  duckdb_logical_type *types,
  idx_t column_count
);
```

Parameters

- `types`

An array of types of the data chunk.

- `column_count`

The number of columns.

- `returns`

The data chunk.

`duckdb_destroy_data_chunk` Destroys the data chunk and de-allocates all memory allocated for that chunk.

Syntax

```
void duckdb_destroy_data_chunk(
  duckdb_data_chunk *chunk
);
```

Parameters

- `chunk`

The data chunk to destroy.

`duckdb_data_chunk_reset` Resets a data chunk, clearing the validity masks and setting the cardinality of the data chunk to 0.

Syntax

```
void duckdb_data_chunk_reset(
  duckdb_data_chunk chunk
);
```

Parameters

- `chunk`

The data chunk to reset.

`duckdb_data_chunk_get_column_count` Retrieves the number of columns in a data chunk.

Syntax

```
idx_t duckdb_data_chunk_get_column_count(
  duckdb_data_chunk chunk
);
```

Parameters

- `chunk`

The data chunk to get the data from

- `returns`

The number of columns in the data chunk

`duckdb_data_chunk_get_vector` Retrieves the vector at the specified column index in the data chunk.

The pointer to the vector is valid for as long as the chunk is alive. It does NOT need to be destroyed.

Syntax

```
duckdb_vector duckdb_data_chunk_get_vector(
  duckdb_data_chunk chunk,
  idx_t col_idx
);
```

Parameters

- `chunk`

The data chunk to get the data from

- `returns`

The vector

`duckdb_data_chunk_get_size` Retrieves the current number of tuples in a data chunk.

Syntax

```
idx_t duckdb_data_chunk_get_size(
  duckdb_data_chunk chunk
);
```

Parameters

- `chunk`

The data chunk to get the data from

- `returns`

The number of tuples in the data chunk

`duckdb_data_chunk_set_size` Sets the current number of tuples in a data chunk.

Syntax

```
void duckdb_data_chunk_set_size(
  duckdb_data_chunk chunk,
  idx_t size
);
```

Parameters

- `chunk`

The data chunk to set the size in

- `size`

The number of tuples in the data chunk

`duckdb_vector_get_column_type` Retrieves the column type of the specified vector.

The result must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_vector_get_column_type(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector get the data from

- `returns`

The type of the vector

`duckdb_vector_get_data` Retrieves the data pointer of the vector.

The data pointer can be used to read or write values from the vector. How to read or write values depends on the type of the vector.

Syntax

```
void *duckdb_vector_get_data(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector to get the data from

- `returns`

The data pointer

`duckdb_vector_get_validity` Retrieves the validity mask pointer of the specified vector.

If all values are valid, this function MIGHT return NULL!

The validity mask is a bitset that signifies null-ness within the data chunk. It is a series of uint64_t values, where each uint64_t value contains validity for 64 tuples. The bit is set to 1 if the value is valid (i.e., not NULL) or 0 if the value is invalid (i.e., NULL).

Validity of a specific value can be obtained like this:

idx_t entry_idx = row_idx / 64; idx_t idx_in_entry = row_idx % 64; bool is_valid = validity_mask[entry_idx] & (1 « idx_in_entry);

Alternatively, the (slower) duckdb_validity_row_is_valid function can be used.

Syntax

```
uint64_t *duckdb_vector_get_validity(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector to get the data from

- `returns`

The pointer to the validity mask, or NULL if no validity mask is present

`duckdb_vector_ensure_validity_writable` Ensures the validity mask is writable by allocating it.

After this function is called, `duckdb_vector_get_validity` will ALWAYS return non-NULL. This allows null values to be written to the vector, regardless of whether a validity mask was present before.

Syntax

```
void duckdb_vector_ensure_validity_writable(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector to alter

`duckdb_vector_assign_string_element` Assigns a string element in the vector at the specified location.

Syntax

```
void duckdb_vector_assign_string_element(
  duckdb_vector vector,
  idx_t index,
  const char *str
);
```

Parameters

- `vector`

The vector to alter

- `index`

The row position in the vector to assign the string to

- `str`

The null-terminated string

`duckdb_vector_assign_string_element_len` Assigns a string element in the vector at the specified location.

Syntax

```
void duckdb_vector_assign_string_element_len(
  duckdb_vector vector,
  idx_t index,
  const char *str,
  idx_t str_len
);
```

Parameters

- `vector`

The vector to alter

- `index`

The row position in the vector to assign the string to

- `str`

The string

- `str_len`

The length of the string (in bytes)

`duckdb_list_vector_get_child` Retrieves the child vector of a list vector.

The resulting vector is valid as long as the parent vector is valid.

Syntax

```
duckdb_vector duckdb_list_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector

- `returns`

The child vector

`duckdb_list_vector_get_size` Returns the size of the child vector of the list

Syntax

```
idx_t duckdb_list_vector_get_size(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector

- `returns`

The size of the child list

`duckdb_list_vector_set_size` Sets the total size of the underlying child-vector of a list vector.

Syntax

```
duckdb_state duckdb_list_vector_set_size(
  duckdb_vector vector,
  idx_t size
);
```

Parameters

- `vector`

The list vector.

- `size`

The size of the child list.

- `returns`

The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_list_vector_reserve` Sets the total capacity of the underlying child-vector of a list.

Syntax

```
duckdb_state duckdb_list_vector_reserve(
  duckdb_vector vector,
  idx_t required_capacity
);
```

Parameters

- `vector`

The list vector.

- `required_capacity`

the total capacity to reserve.

- `return`

The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_struct_vector_get_child` Retrieves the child vector of a struct vector.

The resulting vector is valid as long as the parent vector is valid.

Syntax

```
duckdb_vector duckdb_struct_vector_get_child(
  duckdb_vector vector,
  idx_t index
);
```

Parameters

- `vector`

The vector

- `index`

The child index

- `returns`

The child vector

`duckdb_validity_row_is_valid` Returns whether or not a row is valid (i.e., not NULL) in the given validity mask.

Syntax

```
bool duckdb_validity_row_is_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`

The validity mask, as obtained through `duckdb_vector_get_validity`

- `row`

The row index

- `returns`

true if the row is valid, false otherwise

`duckdb_validity_set_row_validity` In a validity mask, sets a specific row to either valid or invalid.

Note that `duckdb_vector_ensure_validity_writable` should be called before calling `duckdb_vector_get_validity`, to ensure that there is a validity mask to write to.

Syntax

```
void duckdb_validity_set_row_validity(
  uint64_t *validity,
  idx_t row,
  bool valid
);
```

Parameters

- `validity`

The validity mask, as obtained through `duckdb_vector_get_validity`.

- `row`

The row index

- `valid`

Whether or not to set the row to valid, or invalid

`duckdb_validity_set_row_invalid` In a validity mask, sets a specific row to invalid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to false.

Syntax

```
void duckdb_validity_set_row_invalid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`

The validity mask

- `row`

The row index

`duckdb_validity_set_row_valid` In a validity mask, sets a specific row to valid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to true.

Syntax

```
void duckdb_validity_set_row_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`

The validity mask

- `row`

The row index

`duckdb_create_table_function` Creates a new empty table function.

The return value should be destroyed with `duckdb_destroy_table_function`.

Syntax

```
duckdb_table_function duckdb_create_table_function(

);
```

Parameters

- `returns`

The table function object.

`duckdb_destroy_table_function` Destroys the given table function object.

Syntax

```
void duckdb_destroy_table_function(
  duckdb_table_function *table_function
);
```

Parameters

- `table_function`

The table function to destroy

`duckdb_table_function_set_name` Sets the name of the given table function.

Syntax

```
void duckdb_table_function_set_name(
  duckdb_table_function table_function,
  const char *name
);
```

Parameters

- `table_function`

The table function

- `name`

The name of the table function

`duckdb_table_function_add_parameter` Adds a parameter to the table function.

Syntax

```
void duckdb_table_function_add_parameter(
  duckdb_table_function table_function,
  duckdb_logical_type type
);
```

Parameters

- `table_function`

The table function

- `type`

The type of the parameter to add.

`duckdb_table_function_add_named_parameter` Adds a named parameter to the table function.

Syntax

```
void duckdb_table_function_add_named_parameter(
  duckdb_table_function table_function,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `table_function`

The table function

- `name`

The name of the parameter

- `type`

The type of the parameter to add.

`duckdb_table_function_set_extra_info` Assigns extra information to the table function that can be fetched during binding, etc.

Syntax

```
void duckdb_table_function_set_extra_info(
  duckdb_table_function table_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `table_function`

The table function

- `extra_info`

The extra information

- `destroy`

The callback that will be called to destroy the bind data (if any)

`duckdb_table_function_set_bind` Sets the bind function of the table function

Syntax

```
void duckdb_table_function_set_bind(
  duckdb_table_function table_function,
  duckdb_table_function_bind_t bind
);
```

Parameters

- `table_function`

The table function

- `bind`

The bind function

`duckdb_table_function_set_init` Sets the init function of the table function

Syntax

```
void duckdb_table_function_set_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`

The table function

- `init`

The init function

`duckdb_table_function_set_local_init` Sets the thread-local init function of the table function

Syntax

```
void duckdb_table_function_set_local_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`

The table function

- `init`

The init function

`duckdb_table_function_set_function` Sets the main function of the table function

Syntax

```
void duckdb_table_function_set_function(
  duckdb_table_function table_function,
  duckdb_table_function_t function
);
```

Parameters

- `table_function`

The table function

- `function`

The function

`duckdb_table_function_supports_projection_pushdown` Sets whether or not the given table function supports projection pushdown.

If this is set to true, the system will provide a list of all required columns in the `init` stage through the `duckdb_init_get_column_count` and `duckdb_init_get_column_index` functions. If this is set to false (the default), the system will expect all columns to be projected.

Syntax

```
void duckdb_table_function_supports_projection_pushdown(
  duckdb_table_function table_function,
  bool pushdown
);
```

Parameters

- `table_function`

The table function

- `pushdown`

True if the table function supports projection pushdown, false otherwise.

`duckdb_register_table_function` Register the table function object within the given connection.

The function requires at least a name, a bind function, an init function and a main function.

If the function is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
duckdb_state duckdb_register_table_function(
  duckdb_connection con,
  duckdb_table_function function
);
```

Parameters

- `con`

The connection to register it in.

- `function`

The function pointer

- `returns`

Whether or not the registration was successful.

`duckdb_bind_get_extra_info` Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`

Syntax

```
void *duckdb_bind_get_extra_info(
  duckdb_bind_info info
);
```

Parameters

- `info`

The info object

- `returns`

The extra info

`duckdb_bind_add_result_column` Adds a result column to the output of the table function.

Syntax

```
void duckdb_bind_add_result_column(
  duckdb_bind_info info,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `info`

The info object

- `name`

The name of the column

- `type`

The logical type of the column

`duckdb_bind_get_parameter_count` Retrieves the number of regular (non-named) parameters to the function.

Syntax

```
idx_t duckdb_bind_get_parameter_count(
  duckdb_bind_info info
);
```

Parameters

- `info`

The info object

- `returns`

The number of parameters

`duckdb_bind_get_parameter` Retrieves the parameter at the given index.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_bind_get_parameter(
  duckdb_bind_info info,
  idx_t index
);
```

Parameters

- `info`

The info object

- `index`

The index of the parameter to get

- `returns`

The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_get_named_parameter` Retrieves a named parameter with the given name.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_bind_get_named_parameter(
  duckdb_bind_info info,
  const char *name
);
```

Parameters

- `info`

The info object

- `name`

The name of the parameter

- `returns`

The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_set_bind_data` Sets the user-provided bind data in the bind object. This object can be retrieved again during execution.

Syntax

```
void duckdb_bind_set_bind_data(
  duckdb_bind_info info,
  void *bind_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`

The info object

- `extra_data`

The bind data object.

- `destroy`

The callback that will be called to destroy the bind data (if any)

`duckdb_bind_set_cardinality` Sets the cardinality estimate for the table function, used for optimization.

Syntax

```
void duckdb_bind_set_cardinality(
  duckdb_bind_info info,
  idx_t cardinality,
  bool is_exact
);
```

Parameters

- `info`

The bind data object.

- `is_exact`

Whether or not the cardinality estimate is exact, or an approximation

`duckdb_bind_set_error` Report that an error has occurred while calling bind.

Syntax

```
void duckdb_bind_set_error(
  duckdb_bind_info info,
  const char *error
);
```

Parameters

- `info`

The info object

- `error`

The error message

`duckdb_init_get_extra_info` Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`

Syntax

```
void *duckdb_init_get_extra_info(
  duckdb_init_info info
);
```

Parameters

- `info`

The info object

- `returns`

The extra info

`duckdb_init_get_bind_data` Gets the bind data set by `duckdb_bind_set_bind_data` during the bind.

Note that the bind data should be considered as read-only. For tracking state, use the init data instead.

Syntax

```
void *duckdb_init_get_bind_data(
  duckdb_init_info info
);
```

Parameters

- `info`

The info object

- `returns`

The bind data object

`duckdb_init_set_init_data` Sets the user-provided init data in the init object. This object can be retrieved again during execution.

Syntax

```
void duckdb_init_set_init_data(
  duckdb_init_info info,
  void *init_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`

The info object

- `extra_data`

The init data object.

- `destroy`

The callback that will be called to destroy the init data (if any)

`duckdb_init_get_column_count` Returns the number of projected columns.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
idx_t duckdb_init_get_column_count(
  duckdb_init_info info
);
```

Parameters

- `info`

The info object

- `returns`

The number of projected columns.

`duckdb_init_get_column_index` Returns the column index of the projected column at the specified position.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
idx_t duckdb_init_get_column_index(
  duckdb_init_info info,
  idx_t column_index
);
```

Parameters

- `info`

The info object

- `column_index`

The index at which to get the projected column index, from 0..duckdb_init_get_column_count(info)

- `returns`

The column index of the projected column.

`duckdb_init_set_max_threads` Sets how many threads can process this table function in parallel (default: 1)

Syntax

```
void duckdb_init_set_max_threads(
  duckdb_init_info info,
  idx_t max_threads
);
```

Parameters

- `info`

The info object

- `max_threads`

The maximum amount of threads that can process this table function

`duckdb_init_set_error` Report that an error has occurred while calling init.

Syntax

```
void duckdb_init_set_error(
  duckdb_init_info info,
  const char *error
);
```

Parameters

- `info`

The info object

- `error`

The error message

`duckdb_function_get_extra_info` Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`

Syntax

```
void *duckdb_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`

The info object

- `returns`

The extra info

`duckdb_function_get_bind_data` Gets the bind data set by `duckdb_bind_set_bind_data` during the bind.

Note that the bind data should be considered as read-only. For tracking state, use the init data instead.

Syntax

```
void *duckdb_function_get_bind_data(
  duckdb_function_info info
);
```

Parameters

- `info`

The info object

- `returns`

The bind data object

`duckdb_function_get_init_data` Gets the init data set by `duckdb_init_set_init_data` during the init.

Syntax

```
void *duckdb_function_get_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`

The info object

- `returns`

The init data object

`duckdb_function_get_local_init_data` Gets the thread-local init data set by `duckdb_init_set_init_data` during the local_init.

Syntax

```
void *duckdb_function_get_local_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`

The info object

- `returns`

The init data object

`duckdb_function_set_error` Report that an error has occurred while executing the function.

Syntax

```
void duckdb_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`

The info object

- `error`

The error message

`duckdb_add_replacement_scan` Add a replacement scan definition to the specified database

Syntax

```
void duckdb_add_replacement_scan(
  duckdb_database db,
  duckdb_replacement_callback_t replacement,
  void *extra_data,
  duckdb_delete_callback_t delete_callback
);
```

Parameters

- `db`

The database object to add the replacement scan to

- `replacement`

The replacement scan callback

- `extra_data`

Extra data that is passed back into the specified callback

- `delete_callback`

The delete callback to call on the extra data, if any

`duckdb_replacement_scan_set_function_name` Sets the replacement function name to use. If this function is called in the replacement callback, the replacement scan is performed. If it is not called, the replacement callback is not performed.

Syntax

```
void duckdb_replacement_scan_set_function_name(
  duckdb_replacement_scan_info info,
  const char *function_name
);
```

Parameters

- `info`

The info object

- `function_name`

The function name to substitute.

`duckdb_replacement_scan_add_parameter` Adds a parameter to the replacement scan function.

Syntax

```
void duckdb_replacement_scan_add_parameter(
  duckdb_replacement_scan_info info,
  duckdb_value parameter
);
```

Parameters

- `info`

The info object

- `parameter`

The parameter to add.

`duckdb_replacement_scan_set_error` Report that an error has occurred while executing the replacement scan.

Syntax

```
void duckdb_replacement_scan_set_error(
  duckdb_replacement_scan_info info,
  const char *error
);
```

Parameters

- `info`

The info object

- `error`

The error message

`duckdb_appender_create` Creates an appender object.

Syntax

```
duckdb_state duckdb_appender_create(
  duckdb_connection connection,
  const char *schema,
  const char *table,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`

The connection context to create the appender in.

- `schema`

The schema of the table to append to, or `nullptr` for the default schema.

- `table`

The table name to append to.

- `out_appender`

The resulting appender object.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_error` Returns the error message associated with the given appender. If the appender has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_appender_destroy` is called.

Syntax

```
const char *duckdb_appender_error(
  duckdb_appender appender
);
```

Parameters

- `appender`

The appender to get the error from.

- `returns`

The error message, or `nullptr` if there is none.

`duckdb_appender_flush` Flush the appender to the table, forcing the cache of the appender to be cleared and the data to be appended to the base table.

This should generally not be used unless you know what you are doing. Instead, call `duckdb_appender_destroy` when you are done with the appender.

Syntax

```
duckdb_state duckdb_appender_flush(
  duckdb_appender appender
);
```

Parameters

- `appender`

The appender to flush.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_close` Close the appender, flushing all intermediate state in the appender to the table and closing it for further appends.

This is generally not necessary. Call `duckdb_appender_destroy` instead.

Syntax

```
duckdb_state duckdb_appender_close(
  duckdb_appender appender
);
```

Parameters

- `appender`

The appender to flush and close.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_destroy` Close the appender and destroy it. Flushing all intermediate state in the appender to the table, and de-allocating all memory associated with the appender.

Syntax

```
duckdb_state duckdb_appender_destroy(
  duckdb_appender *appender
);
```

Parameters

- `appender`

The appender to flush, close and destroy.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_begin_row` A nop function, provided for backwards compatibility reasons. Does nothing. Only `duckdb_appender_end_row` is required.

Syntax

```
duckdb_state duckdb_appender_begin_row(
  duckdb_appender appender
);
```

`duckdb_appender_end_row` Finish the current row of appends. After end_row is called, the next row can be appended.

Syntax

```
duckdb_state duckdb_appender_end_row(
  duckdb_appender appender
);
```

Parameters

- `appender`

The appender.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_append_bool` Append a bool value to the appender.

Syntax

```
duckdb_state duckdb_append_bool(
  duckdb_appender appender,
  bool value
);
```

`duckdb_append_int8` Append an int8_t value to the appender.

Syntax

```
duckdb_state duckdb_append_int8(
  duckdb_appender appender,
  int8_t value
);
```

`duckdb_append_int16` Append an int16_t value to the appender.

Syntax

```
duckdb_state duckdb_append_int16(
  duckdb_appender appender,
  int16_t value
);
```

duckdb_append_int32 Append an int32_t value to the appender.

Syntax

```
duckdb_state duckdb_append_int32(
  duckdb_appender appender,
  int32_t value
);
```

duckdb_append_int64 Append an int64_t value to the appender.

Syntax

```
duckdb_state duckdb_append_int64(
  duckdb_appender appender,
  int64_t value
);
```

duckdb_append_hugeint Append a duckdb_hugeint value to the appender.

Syntax

```
duckdb_state duckdb_append_hugeint(
  duckdb_appender appender,
  duckdb_hugeint value
);
```

duckdb_append_uint8 Append a uint8_t value to the appender.

Syntax

```
duckdb_state duckdb_append_uint8(
  duckdb_appender appender,
  uint8_t value
);
```

duckdb_append_uint16 Append a uint16_t value to the appender.

Syntax

```
duckdb_state duckdb_append_uint16(
  duckdb_appender appender,
  uint16_t value
);
```

`duckdb_append_uint32` Append a uint32_t value to the appender.

Syntax

```
duckdb_state duckdb_append_uint32(
  duckdb_appender appender,
  uint32_t value
);
```

`duckdb_append_uint64` Append a uint64_t value to the appender.

Syntax

```
duckdb_state duckdb_append_uint64(
  duckdb_appender appender,
  uint64_t value
);
```

`duckdb_append_float` Append a float value to the appender.

Syntax

```
duckdb_state duckdb_append_float(
  duckdb_appender appender,
  float value
);
```

`duckdb_append_double` Append a double value to the appender.

Syntax

```
duckdb_state duckdb_append_double(
  duckdb_appender appender,
  double value
);
```

`duckdb_append_date` Append a duckdb_date value to the appender.

Syntax

```
duckdb_state duckdb_append_date(
  duckdb_appender appender,
  duckdb_date value
);
```

`duckdb_append_time` Append a duckdb_time value to the appender.

Syntax

```
duckdb_state duckdb_append_time(
  duckdb_appender appender,
  duckdb_time value
);
```

`duckdb_append_timestamp` Append a duckdb_timestamp value to the appender.

Syntax

```
duckdb_state duckdb_append_timestamp(
  duckdb_appender appender,
  duckdb_timestamp value
);
```

`duckdb_append_interval` Append a duckdb_interval value to the appender.

Syntax

```
duckdb_state duckdb_append_interval(
  duckdb_appender appender,
  duckdb_interval value
);
```

`duckdb_append_varchar` Append a varchar value to the appender.

Syntax

```
duckdb_state duckdb_append_varchar(
  duckdb_appender appender,
  const char *val
);
```

`duckdb_append_varchar_length` Append a varchar value to the appender.

Syntax

```
duckdb_state duckdb_append_varchar_length(
  duckdb_appender appender,
  const char *val,
  idx_t length
);
```

`duckdb_append_blob` Append a blob value to the appender.

Syntax

```
duckdb_state duckdb_append_blob(
  duckdb_appender appender,
  const void *data,
  idx_t length
);
```

`duckdb_append_null` Append a NULL value to the appender (of any type).

Syntax

```
duckdb_state duckdb_append_null(
  duckdb_appender appender
);
```

`duckdb_append_data_chunk` Appends a pre-filled data chunk to the specified appender.

The types of the data chunk must exactly match the types of the table, no casting is performed. If the types do not match or the appender is in an invalid state, DuckDBError is returned. If the append is successful, DuckDBSuccess is returned.

Syntax

```
duckdb_state duckdb_append_data_chunk(
  duckdb_appender appender,
  duckdb_data_chunk chunk
);
```

Parameters

- `appender`

The appender to append to.

- `chunk`

The data chunk to append.

- `returns`

The return state.

`duckdb_query_arrow` Executes a SQL query within a connection and stores the full (materialized) result in an arrow structure. If the query fails to execute, DuckDBError is returned and the error message can be retrieved by calling `duckdb_query_arrow_error`.

Note that after running `duckdb_query_arrow`, `duckdb_destroy_arrow` must be called on the result object even if the query fails, otherwise the error stored within the result will not be freed correctly.

Syntax

```
duckdb_state duckdb_query_arrow(
  duckdb_connection connection,
  const char *query,
  duckdb_arrow *out_result
);
```

Parameters

- `connection`

The connection to perform the query in.

- `query`

The SQL query to run.

- `out_result`

The query result.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_query_arrow_schema` Fetch the internal arrow schema from the arrow result.

Syntax

```
duckdb_state duckdb_query_arrow_schema(
  duckdb_arrow result,
  duckdb_arrow_schema *out_schema
);
```

Parameters

- `result`

The result to fetch the schema from.

- `out_schema`

The output schema.

- `returns`

DuckDBSuccess on success or DuckDBError on failure.

`duckdb_prepared_arrow_schema` Fetch the internal arrow schema from the prepared statement.

Syntax

```
duckdb_state duckdb_prepared_arrow_schema(
  duckdb_prepared_statement prepared,
  duckdb_arrow_schema *out_schema
);
```

Parameters

- `result`

The prepared statement to fetch the schema from.

- `out_schema`

The output schema.

- `returns`

DuckDBSuccess on success or DuckDBError on failure.

`duckdb_query_arrow_array` Fetch an internal arrow array from the arrow result.

This function can be called multiple time to get next chunks, which will free the previous out_array. So consume the out_array before calling this function again.

Syntax

```
duckdb_state duckdb_query_arrow_array(
  duckdb_arrow result,
  duckdb_arrow_array *out_array
);
```

Parameters

- `result`

The result to fetch the array from.

- `out_array`

The output array.

- `returns`

`DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_arrow_column_count` Returns the number of columns present in a the arrow result object.

Syntax

```
idx_t duckdb_arrow_column_count(
  duckdb_arrow result
);
```

Parameters

- `result`

The result object.

- `returns`

The number of columns present in the result object.

`duckdb_arrow_row_count` Returns the number of rows present in a the arrow result object.

Syntax

```
idx_t duckdb_arrow_row_count(
  duckdb_arrow result
);
```

Parameters

- `result`

The result object.

- `returns`

The number of rows present in the result object.

`duckdb_arrow_rows_changed` Returns the number of rows changed by the query stored in the arrow result. This is relevant only for INSERT/UPDATE/DELETE queries. For other queries the rows_changed will be 0.

Syntax

```
idx_t duckdb_arrow_rows_changed(
  duckdb_arrow result
);
```

Parameters

- `result`

The result object.

- `returns`

The number of rows changed.

`duckdb_query_arrow_error` Returns the error message contained within the result. The error is only set if `duckdb_query_arrow` returns `DuckDBError`.

The error message should not be freed. It will be de-allocated when `duckdb_destroy_arrow` is called.

Syntax

```
const char *duckdb_query_arrow_error(
  duckdb_arrow result
);
```

Parameters

- `result`

The result object to fetch the nullmask from.

- `returns`

The error of the result.

`duckdb_destroy_arrow` Closes the result and de-allocates all memory allocated for the arrow result.

Syntax

```
void duckdb_destroy_arrow(
  duckdb_arrow *result
);
```

Parameters

- `result`

The result to destroy.

`duckdb_execute_tasks` Execute DuckDB tasks on this thread.

Will return after `max_tasks` have been executed, or if there are no more tasks present.

Syntax

```
void duckdb_execute_tasks(
  duckdb_database database,
  idx_t max_tasks
);
```

Parameters

- `database`

The database object to execute tasks for

- `max_tasks`

The maximum amount of tasks to execute

`duckdb_create_task_state` Creates a task state that can be used with duckdb_execute_tasks_state to execute tasks until duckdb_finish_execution is called on the state.

duckdb_destroy_state should be called on the result in order to free memory.

Syntax

```
duckdb_task_state duckdb_create_task_state(
  duckdb_database database
);
```

Parameters

- `database`

The database object to create the task state for

- `returns`

The task state that can be used with duckdb_execute_tasks_state.

`duckdb_execute_tasks_state` Execute DuckDB tasks on this thread.

The thread will keep on executing tasks forever, until duckdb_finish_execution is called on the state. Multiple threads can share the same duckdb_task_state.

Syntax

```
void duckdb_execute_tasks_state(
  duckdb_task_state state
);
```

Parameters

- `state`

The task state of the executor

`duckdb_execute_n_tasks_state` Execute DuckDB tasks on this thread.

The thread will keep on executing tasks until either duckdb_finish_execution is called on the state, max_tasks tasks have been executed or there are no more tasks to be executed.

Multiple threads can share the same duckdb_task_state.

Syntax

```
idx_t duckdb_execute_n_tasks_state(
  duckdb_task_state state,
  idx_t max_tasks
);
```

Parameters

- `state`

The task state of the executor

- `max_tasks`

The maximum amount of tasks to execute

- `returns`

The amount of tasks that have actually been executed

`duckdb_finish_execution` Finish execution on a specific task.

Syntax

```
void duckdb_finish_execution(
  duckdb_task_state state
);
```

Parameters

- `state`

The task state to finish execution

`duckdb_task_state_is_finished` Check if the provided duckdb_task_state has finished execution

Syntax

```
bool duckdb_task_state_is_finished(
  duckdb_task_state state
);
```

Parameters

- `state`

The task state to inspect

- `returns`

Whether or not duckdb_finish_execution has been called on the task state

`duckdb_destroy_task_state` Destroys the task state returned from duckdb_create_task_state.

Note that this should not be called while there is an active duckdb_execute_tasks_state running on the task state.

Syntax

```
void duckdb_destroy_task_state(
  duckdb_task_state state
);
```

Parameters

- `state`

The task state to clean up

`duckdb_execution_is_finished` Returns true if execution of the current query is finished.

Syntax

```
bool duckdb_execution_is_finished(
  duckdb_connection con
);
```

Parameters

- `con`

The connection on which to check

`duckdb_stream_fetch_chunk` Fetches a data chunk from the (streaming) duckdb_result. This function should be called repeatedly until the result is exhausted.

The result must be destroyed with `duckdb_destroy_data_chunk`.

This function can only be used on duckdb_results created with 'duckdb_pending_prepared_streaming'

If this function is used, none of the other result functions can be used and vice versa (i.e., this function cannot be mixed with the legacy result functions or the materialized result functions).

It is not known beforehand how many chunks will be returned by this result.

Syntax

```
duckdb_data_chunk duckdb_stream_fetch_chunk(
  duckdb_result result
);
```

Parameters

- `result`

The result object to fetch the data chunk from.

- `returns`

The resulting data chunk. Returns `NULL` if the result has an error.

C++ API

Installation

The DuckDB C++ API can be installed as part of the `libduckdb` packages. Please see the installation page for details.

Basic API Usage

DuckDB implements a custom C++ API. This is built around the abstractions of a database instance (`DuckDB` class), multiple `Connections` to the database instance and `QueryResult` instances as the result of queries. The header file for the C++ API is `duckdb.hpp`.

> **Note.** The standard source distribution of `libduckdb` contains an "amalgamation" of the DuckDB sources, which combine all sources into two files `duckdb.hpp` and `duckdb.cpp`. The `duckdb.hpp` header is much larger in this case. Regardless of whether you are using the amalgamation or not, just include `duckdb.hpp`.

Startup & Shutdown To use DuckDB, you must first initialize a `DuckDB` instance using its constructor. `DuckDB()` takes as parameter the database file to read and write from. The special value `nullptr` can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the process). The second parameter to the `DuckDB` constructor is an optional `DBConfig` object. In `DBConfig`, you can set various database parameters, for example the read/write mode or memory limits. The `DuckDB` constructor may throw exceptions, for example if the database file is not usable.

With the `DuckDB` instance, you can create one or many `Connection` instances using the `Connection()` constructor. While connections should be thread-safe, they will be locked during querying. It is therefore recommended that each thread uses its own connection if you are in a multithreaded environment.

```
DuckDB db(nullptr);
Connection con(db);
```

Querying Connections expose the `Query()` method to send a SQL query string to DuckDB from C++. `Query()` fully materializes the query result as a `MaterializedQueryResult` in memory before returning at which point the query result can be consumed. There is also a streaming API for queries, see further below.

```
// create a table
con.Query("CREATE TABLE integers(i INTEGER, j INTEGER)");

// insert three rows into the table
con.Query("INSERT INTO integers VALUES (3, 4), (5, 6), (7, NULL)");

MaterializedQueryResult result = con.Query("SELECT * FROM integers");
if (!result->success) {
    cerr << result->error;
}
```

The `MaterializedQueryResult` instance contains firstly two fields that indicate whether the query was successful. `Query` will not throw exceptions under normal circumstances. Instead, invalid queries or other issues will lead to the `success` boolean field in the query result instance to

be set to `false`. In this case an error message may be available in `error` as a string. If successful, other fields are set: the type of statement that was just executed (e.g., `StatementType::INSERT_STATEMENT`) is contained in `statement_type`. The high-level ("Logical type"/"SQL type") types of the result set columns are in `types`. The names of the result columns are in the `names` string vector. In case multiple result sets are returned, for example because the result set contained multiple statements, the result set can be chained using the `next` field.

DuckDB also supports prepared statements in the C++ API with the `Prepare()` method. This returns an instance of `PreparedStatement`. This instance can be used to execute the prepared statement with parameters. Below is an example:

```
std::unique_ptr<PreparedStatement> prepare = con.Prepare("SELECT COUNT(*)
 ↪  FROM a WHERE i=$1");
std::unique_ptr<QueryResult> result = prepare->Execute(12);
```

> **Note.** Do **not** use prepared statements to insert large amounts of data into DuckDB. See the data import documentation for better options.

UDF API The UDF API allows the definition of user-defined functions. It is exposed in `duckdb:Connection` through the methods: `CreateScalarFunction()`, `CreateVectorizedFunction()`, and variants. These methods created UDFs into the temporary schema (TEMP_SCHEMA) of the owner connection that is the only one allowed to use and change them.

CreateScalarFunction The user can code an ordinary scalar function and invoke the `CreateScalarFunction()` to register and afterward use the UDF in a *SELECT* statement, for instance:

```
bool bigger_than_four(int value) {
    return value > 4;
}

connection.CreateScalarFunction<bool, int>("bigger_than_four", &bigger_
 ↪  than_four);

connection.Query("SELECT bigger_than_four(i) FROM (VALUES(3), (5))
 ↪  tbl(i)")->Print();
```

The `CreateScalarFunction()` methods automatically creates vectorized scalar UDFs so they are as efficient as built-in functions, we have two variants of this method interface as follows:

1.

```
template<typename TR, typename... Args>
void CreateScalarFunction(string name, TR (*udf_func)(Args…))
```

- template parameters:
  - **TR** is the return type of the UDF function;
  - **Args** are the arguments up to 3 for the UDF function (this method only supports until ternary functions);
- **name**: is the name to register the UDF function;
- **udf_func**: is a pointer to the UDF function.

This method automatically discovers from the template typenames the corresponding LogicalTypes:

- `bool` → `LogicalType::BOOLEAN`
- `int8_t` → `LogicalType::TINYINT`
- `int16_t` → `LogicalType::SMALLINT`
- `int32_t` → `LogicalType::INTEGER`
- `int64_t` →`LogicalType::BIGINT`
- `float` → `LogicalType::FLOAT`
- `double` → `LogicalType::DOUBLE`
- `string_t` → `LogicalType::VARCHAR`

*In DuckDB some primitive types, e.g., *int32_t*, are mapped to the same LogicalType: `INTEGER`, `TIME` and `DATE`, then for disambiguation the users can use the following overloaded method.

2.

```
template<typename TR, typename... Args>
void CreateScalarFunction(string name, vector<LogicalType> args, LogicalType
↪  ret_type, TR (*udf_func)(Args…))
```

An example of use would be:

```
int32_t udf_date(int32_t a) {
    return a;
}

con.Query("CREATE TABLE dates (d DATE)");
con.Query("INSERT INTO dates VALUES ('1992-01-01')");

con.CreateScalarFunction<int32_t, int32_t>("udf_date", {LogicalType::DATE},
↪  LogicalType::DATE, &udf_date);
```

```
con.Query("SELECT udf_date(d) FROM dates")->Print();
```

- template parameters:
  - **TR** is the return type of the UDF function;
  - **Args** are the arguments up to 3 for the UDF function (this method only supports until ternary functions);
- **name**: is the name to register the UDF function;
- **args**: are the LogicalType arguments that the function uses, which should match with the template Args types;
- **ret_type**: is the LogicalType of return of the function, which should match with the template TR type;
- **udf_func**: is a pointer to the UDF function.

This function checks the template types against the LogicalTypes passed as arguments and they must match as follow:

- LogicalTypeId::BOOLEAN → bool
- LogicalTypeId::TINYINT → int8_t
- LogicalTypeId::SMALLINT → int16_t
- LogicalTypeId::DATE, LogicalTypeId::TIME, LogicalTypeId::INTEGER → int32_t
- LogicalTypeId::BIGINT, LogicalTypeId::TIMESTAMP → int64_t
- LogicalTypeId::FLOAT, LogicalTypeId::DOUBLE, LogicalTypeId::DECIMAL → double
- LogicalTypeId::VARCHAR, LogicalTypeId::CHAR, LogicalTypeId::BLOB → string_t
- LogicalTypeId::VARBINARY → blob_t

CreateVectorizedFunction The `CreateVectorizedFunction()` methods register a vectorized UDF such as:

```
/*
* This vectorized function copies the input values to the result vector
*/
template<typename TYPE>
static void udf_vectorized(DataChunk &args, ExpressionState &state, Vector
 ↪  &result) {
    // set the result vector type
    result.vector_type = VectorType::FLAT_VECTOR;
    // get a raw array from the result
    auto result_data = FlatVector::GetData<TYPE>(result);
```

```
    // get the solely input vector
    auto &input = args.data[0];
    // now get an orrified vector
    VectorData vdata;
    input.Orrify(args.size(), vdata);

    // get a raw array from the orrified input
    auto input_data = (TYPE *)vdata.data;

    // handling the data
    for (idx_t i = 0; i < args.size(); i++) {
        auto idx = vdata.sel->get_index(i);
        if ((*vdata.nullmask)[idx]) {
            continue;
        }
        result_data[i] = input_data[idx];
    }
}

con.Query("CREATE TABLE integers (i INTEGER)");
con.Query("INSERT INTO integers VALUES (1), (2), (3), (999)");

con.CreateVectorizedFunction<int, int>("udf_vectorized_int", &&udf_
 ↪  vectorized<int>);

con.Query("SELECT udf_vectorized_int(i) FROM integers")->Print();
```

The Vectorized UDF is a pointer of the type *scalar_function_t*:

```
typedef std::function<void(DataChunk &args, ExpressionState &expr, Vector
 ↪  &result)> scalar_function_t;
```

- **args** is a DataChunk that holds a set of input vectors for the UDF that all have the same length;
- **expr** is an ExpressionState that provides information to the query's expression state;
- **result**: is a Vector to store the result values.

There are different vector types to handle in a Vectorized UDF:

- ConstantVector;
- DictionaryVector;
- FlatVector;
- ListVector;
- StringVector;
- StructVector;

- SequenceVector.

The general API of the `CreateVectorizedFunction()` method is as follows:

1.

```
template<typename TR, typename... Args>
void CreateVectorizedFunction(string name, scalar_function_t udf_func,
 ↪  LogicalType varargs = LogicalType::INVALID)
```

- template parameters:
  - **TR** is the return type of the UDF function;
  - **Args** are the arguments up to 3 for the UDF function.
- **name** is the name to register the UDF function;
- **udf_func** is a *vectorized* UDF function;
- **varargs** The type of varargs to support, or LogicalTypeId::INVALID (default value) if the function does not accept variable length arguments.

This method automatically discovers from the template typenames the corresponding LogicalTypes:

- bool → LogicalType::BOOLEAN;
- int8_t → LogicalType::TINYINT;
- int16_t → LogicalType::SMALLINT
- int32_t → LogicalType::INTEGER
- int64_t → LogicalType::BIGINT
- float → LogicalType::FLOAT
- double → LogicalType::DOUBLE
- string_t → LogicalType::VARCHAR

2.

```
template<typename TR, typename... Args>
void CreateVectorizedFunction(string name, vector<LogicalType> args,
 ↪  LogicalType ret_type, scalar_function_t udf_func, LogicalType varargs =
 ↪  LogicalType::INVALID)
```

CLI API

Installation

The DuckDB CLI (Command Line Interface) is a single, dependency-free executable. It is precompiled for Windows, Mac, and Linux for both the stable version and for nightly builds produced by GitHub

Actions. Please see the installation page under the CLI tab for download links.

The DuckDB CLI is based on the SQLite command line shell, so CLI-client-specific functionality is similar to what is described in the SQLite documentation (although DuckDB's SQL syntax follows PostgreSQL conventions).

Getting Started

Once the CLI executable has been downloaded, unzip it and save it to any directory. Navigate to that directory in a terminal and enter the command `duckdb` to run the executable. If in a PowerShell or POSIX shell environment, use the command `./duckdb` instead.

The executable can be configured in many ways when started. Some common configurations include:

- `-csv`, to set the output mode to CSV
- `-json` to set the output mode to JSON
- `-readonly` to open the database in read-only mode

> **Note.** DuckDB has two options for concurrent access: Either one process runs which can both read and write to the database, or multiple processes can read from the database but no processes can write (`-readonly`). See concurrency in DuckDB for more details.

To see additional command line options to use when starting the CLI, use the command `duckdb -help`.

> **Note.** DuckDB has a tldr page. If you have tldr installed, you can display it by running `tldr duckdb`.

Frequently-used configurations can be stored in the file `~/.duckdbrc`. See the Configuring the CLI below for further information on these options.

By default, the CLI will open a temporary in-memory database. To open or create a persistent database, simply include a path as a command line argument like `duckdb path/to/my_database.duckdb`. This path can point to an existing database or to a file that does not yet exist and DuckDB will open or create a database at that location as needed. The file may have any arbitrary extension, but `.db` or `.duckdb` are two common choices. You will see a prompt like the below, with a D on the final line.

```
v0.9.1 401c8061c6
Enter ".help" for usage hints.
Connected to a transient in-memory database.
```

```
Use ".open FILENAME" to reopen on a persistent database.
D
```

Once the CLI has been opened, enter a SQL statement followed by a semicolon, then hit enter and it will be executed. Results will be displayed in a table in the terminal. If a semicolon is omitted, hitting enter will allow for multi-line SQL statements to be entered.

```
SELECT 'quack' AS my_column;
```

my_column
quack

```
SELECT
    'nicely formatted quack' AS my_column,
    'excited quacking' AS another_column;
```

my_column	another_column
nicely formatted quack	excited quacking

The CLI supports all of DuckDB's rich SQL syntax including `SELECT`, `CREATE`, and `ALTER` statements, etc.

To exit the CLI, press `Ctrl-D` if your platform supports it. Otherwise press `Ctrl-C`. If using a persistent database, it will automatically checkpoint (save the latest edits to disk) and close. This will remove the .WAL file (the Write-Ahead-Log) and consolidate all of your data into the single file database.

Special Commands (Dot Commands)

In addition to SQL syntax, special dot commands may be entered that are specific to the CLI client. To use one of these commands, begin the line with a period (`.`) immediately followed by the name of the command you wish to execute. Additional arguments to the command are entered, space separated, after the command. If an argument must contain a space, either single or double quotes may be used to wrap that parameter. Dot commands must be entered on a single line, and no whitespace may occur before the period. No semicolon is required at the end of the line. To see available commands, use the `.help` command:

```
.help
.bail on|off             Stop after hitting an error.  Default OFF
```

```
.binary on|off           Turn binary output on or off.  Default OFF
.cd DIRECTORY            Change the working directory to DIRECTORY
.changes on|off          Show number of rows changed by SQL
.check GLOB              Fail if output since .testcase does not match
.clone NEWDB             Clone data into NEWDB from the existing database
.columns                 Column-wise rendering of query results
.constant ?COLOR?        Sets the syntax highlighting color used for
 ↪  constant values
.constantcode ?CODE?     Sets the syntax highlighting terminal code used for
 ↪  constant values
.databases               List names and files of attached databases
.dump ?TABLE?            Render database content as SQL
.echo on|off             Turn command echo on or off
.excel                   Display the output of next command in spreadsheet
.exit ?CODE?             Exit this program with return-code CODE
.explain ?on|off|auto?   Change the EXPLAIN formatting mode.  Default: auto
.fullschema ?--indent?   Show schema and the content of sqlite_stat tables
.headers on|off          Turn display of headers on or off
.help ?-all? ?PATTERN?   Show help text for PATTERN
.highlight [on|off]      Toggle syntax highlighting in the shell on/off
.import FILE TABLE       Import data from FILE into TABLE
.indexes ?TABLE?         Show names of indexes
.keyword ?COLOR?        Sets the syntax highlighting color used for keywords
.keywordcode ?CODE?      Sets the syntax highlighting terminal code used for
 ↪  keywords
.lint OPTIONS            Report potential schema issues.
.log FILE|off            Turn logging on or off.  FILE can be stderr/stdout
.maxrows COUNT           Sets the maximum number of rows for display. Only
 ↪  for duckbox mode.
.maxwidth COUNT          Sets the maximum width in characters. 0 defaults to
 ↪  terminal width. Only for duckbox mode.
.mode MODE ?TABLE?       Set output mode
.nullvalue STRING        Use STRING in place of NULL values
.once ?OPTIONS? ?FILE?   Output for the next SQL command only to FILE
.open ?OPTIONS? ?FILE?   Close existing database and reopen FILE
.output ?FILE?           Send output to FILE or stdout if FILE is omitted
.parameter CMD ...       Manage SQL parameter bindings
.print STRING...         Print literal STRING
.prompt MAIN CONTINUE    Replace the standard prompts
.quit                    Exit this program
.read FILE               Read input from FILE
.rows                    Row-wise rendering of query results (default)
.schema ?PATTERN?        Show the CREATE statements matching PATTERN
```

```
.separator COL ?ROW?     Change the column and row separators
.sha3sum ...             Compute a SHA3 hash of database content
.shell CMD ARGS...       Run CMD ARGS... in a system shell
.show                    Show the current values for various settings
.system CMD ARGS...      Run CMD ARGS... in a system shell
.tables ?TABLE?          List names of tables matching LIKE pattern TABLE
.testcase NAME           Begin redirecting output to 'testcase-out.txt'
.timer on|off            Turn SQL timer on or off
.width NUM1 NUM2 ...     Set minimum column widths for columnar output
```

Note that the above list of methods is extensive, and DuckDB supports only a subset of the commands that are displayed. Please file a GitHub issue if a command that is central to your workflow is not yet supported.

As an example of passing an argument to a dot command, the `.help` text may be filtered by passing in a text string as the second argument.

```
.help sh
```

```
.sha3sum ...             Compute a SHA3 hash of database content
.shell CMD ARGS...       Run CMD ARGS... in a system shell
.show                    Show the current values for various settings
```

Syntax Highlighting

By default the shell includes support for syntax highlighting. Syntax highlighting can be disabled using the `.highlight off` command.

The colors of the syntax highlighting can also be configured using the following commands.

```
.constant
Error: Expected usage: .constant
 ↪  [red|green|yellow|blue|magenta|cyan|white|brightblack|brightred|brightgreen|brightye

.keyword
Error: Expected usage: .keyword
 ↪  [red|green|yellow|blue|magenta|cyan|white|brightblack|brightred|brightgreen|brightye

.keywordcode
Error: Expected usage: .keywordcode [terminal_code]

.constantcode
Error: Expected usage: .constantcode [terminal_code]
```

Auto-Complete

The shell offers context-aware auto-complete of SQL queries. 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
```

Output Formats

The `.mode` command may be used to change the appearance of the tables returned in the terminal output. In addition to customizing the appearance, these modes have additional benefits. This can be useful for presenting DuckDB output elsewhere by redirecting the terminal output to a file, for example (see "Writing Results to a File" section below). Using the `insert` mode will build a series of SQL statements that can be used to insert the data at a later point. The `markdown` mode is particularly useful for building documentation!

mode	description
ascii	Columns/rows delimited by 0x1F and 0x1E
box	Tables using unicode box-drawing characters
csv	Comma-separated values
column	Output in columns. (See .width)
duckbox	Tables with extensive features
html	HTML `<table>` code
insert	SQL insert statements for TABLE

mode	description
json	Results in a JSON array
jsonlines	Results in a NDJSON
latex	LaTeX tabular environment code
line	One value per line
list	Values delimited by "|"
markdown	Markdown table format
quote	Escape answers as for SQL
table	ASCII-art table
tabs	Tab-separated values
tcl	TCL list elements
trash	No output

```
.mode markdown
SELECT 'quacking intensifies' AS incoming_ducks;
```

```
|    incoming_ducks    |
|----------------------|
| quacking intensifies |
```

The output appearance can also be adjusted with the `.separator` command. If using an export mode that relies on a separator (`csv` or `tabs` for example), the separator will be reset when the mode is changed. For example, `.mode csv` will set the separator to a comma (`,`). Using `.separator "|"` will then convert the output to be pipe separated.

```
.mode csv
SELECT 1 AS col_1, 2 AS col_2
UNION ALL
SELECT 10 AS col1, 20 AS col_2;
```

```
col_1,col_2
1,2
10,20
```

```
.separator "|"
SELECT 1 AS col_1, 2 AS col_2
UNION ALL
SELECT 10 AS col1, 20 AS col_2;
```

```
col_1|col_2
1|2
10|20
```

Prepared Statements

The DuckDB CLI supports executing prepared statements in addition to normal `SELECT` statements. To create a prepared statement, use the `PREPARE` statement

```
PREPARE S1 AS SELECT * FROM my_table WHERE my_column < $1 OR my_column > $2;
```

To run the prepared statement with parameters, use the `EXECUTE` statement

```
EXECUTE S1(42, 101);
```

Querying the Database Schema

All DuckDB clients support querying the database schema with SQL, but the CLI has additional dot commands that can make it easier to understand the contents of a database. The `.tables` command will return a list of tables in the database. It has an optional argument that will filter the results according to a `LIKE` pattern.

```
CREATE TABLE swimmers AS SELECT 'duck' AS animal;
CREATE TABLE fliers AS SELECT 'duck' AS animal;
CREATE TABLE walkers AS SELECT 'duck' AS animal;
.tables
```

```
fliers    swimmers  walkers
```

For example, to filter to only tables that contain an "l", use the `LIKE` pattern `%l%`.

```
.tables %l%
```

```
fliers   walkers
```

The `.schema` command will show all of the SQL statements used to define the schema of the database.

```
.schema
```

```
CREATE TABLE fliers(animal VARCHAR);
CREATE TABLE swimmers(animal VARCHAR);
CREATE TABLE walkers(animal VARCHAR);
```

Opening Database Files

In addition to connecting to a database when opening the CLI, a new database connection can be made by using the `.open` command. If no additional parameters are supplied, a new in-memory database connection is created. This database will not be persisted when the CLI connection is closed.

```
.open
```

The `.open` command optionally accepts several options, but the final parameter can be used to indicate a path to a persistent database (or where one should be created). The special string `:memory:` can also be used to open a temporary in-memory database.

```
.open persistent.duckdb
```

One important option accepted by `.open` is the `--readonly` flag. This disallows any editing of the database. To open in read only mode, the database must already exist. This also means that a new in-memory database can't be opened in read only mode since in-memory databases are created upon connection.

```
.open --readonly preexisting.duckdb
```

Writing Results to a File

By default, the DuckDB CLI sends results to the terminal's standard output. However, this can be modified using either the `.output` or `.once` commands. Pass in the desired output file location as a parameter. The `.once` command will only output the next set of results and then revert to standard out, but `.output` will redirect all subsequent output to that file location. Note that each result will overwrite the entire file at that destination. To revert back to standard output, enter `.output` with no file parameter.

In this example, the output format is changed to markdown, the destination is identified as a markdown file, and then DuckDB will write the output of the SQL statement to that file. Output is then reverted to standard output using `.output` with no parameter.

```
.mode markdown
.output my_results.md
SELECT 'taking flight' AS output_column;
.output
SELECT 'back to the terminal' AS displayed_column;
```

The file `my_results.md` will then contain:

```
| output_column |
|---------------|
| taking flight |
```

The terminal will then display:

```
|   displayed_column   |
|----------------------|
| back to the terminal |
```

A common output format is CSV, or comma separated values. DuckDB supports SQL syntax to export data as CSV or Parquet, but the CLI-specific commands may be used to write a CSV instead if desired.

```
.mode csv
.once my_output_file.csv
SELECT 1 AS col_1, 2 AS col_2
UNION ALL
SELECT 10 AS col1, 20 AS col_2;
```

The file my_output_file.csv will then contain:

```
col_1,col_2
1,2
10,20
```

By passing special options (flags) to the `.once` command, query results can also be sent to a temporary file and automatically opened in the user's default program. Use either the `-e` flag for a text file (opened in the default text editor), or the `-x` flag for a CSV file (opened in the default spreadsheet editor). This is useful for more detailed inspection of query results, especially if there is a relatively large result set. The `.excel` command is equivalent to `.once -x`.

```
.once -e
SELECT 'quack' AS hello;
```

The results then open in the default text file editor of the system, for example:

Import Data from CSV

DuckDB supports SQL syntax to directly query or import CSV files, but the CLI-specific commands may be used to import a CSV instead if desired. The `.import` command takes two arguments and also supports several options. The first argument is the path to the CSV file, and the second is the name of the DuckDB table to create. Since DuckDB requires stricter typing than SQLite (upon which the DuckDB CLI is based), the destination table must be created before using the `.import` command. To

automatically detect the schema and create a table from a CSV, see the `read_csv_auto` examples in the import docs.

In this example, a CSV file is generated by changing to CSV mode and setting an output file location:

```
.mode csv
.output import_example.csv
SELECT 1 AS col_1, 2 AS col_2 UNION ALL SELECT 10 AS col1, 20 AS col_2;
```

Now that the CSV has been written, a table can be created with the desired schema and the CSV can be imported. The output is reset to the terminal to avoid continuing to edit the output file specified above. The `--skip N` option is used to ignore the first row of data since it is a header row and the table has already been created with the correct column names.

```
.mode csv
.output
CREATE TABLE test_table (col_1 INT, col_2 INT);
.import import_example.csv test_table --skip 1
```

Note that the `.import` command utilizes the current `.mode` and `.separator` settings when identifying the structure of the data to import. The `--csv` option can be used to override that behavior.

```
.import import_example.csv test_table --skip 1 --csv
```

Reading SQL from a File

The DuckDB CLI can read both SQL commands and dot commands from an external file instead of the terminal using the `.read` command. This allows for a number of commands to be run in sequence and allows command sequences to be saved and reused.

The `.read` command requires only one argument: the path to the file containing the SQL and/or commands to execute. After running the commands in the file, control will revert back to the terminal. Output from the execution of that file is governed by the same `.output` and `.once` commands that have been discussed previously. This allows the output to be displayed back to the terminal, as in the first example below, or out to another file, as in the second example.

In this example, the file `select_example.sql` is located in the same directory as duckdb.exe and contains the following SQL statement:

```
SELECT
    *
FROM generate_series(5);
```

To execute it from the CLI, the `.read` command is used.

```
.read select_example.sql
```

The output below is returned to the terminal by default. The formatting of the table be adjusted using the `.output` or `.once` commands.

```
| generate_series |
|-----------------|
| 0               |
| 1               |
| 2               |
| 3               |
| 4               |
| 5               |
```

Multiple commands, including both SQL and dot commands, can also be run in a single `.read` command. In this example, the file `write_markdown_to_file.sql` is located in the same directory as duckdb.exe and contains the following commands:

```
.mode markdown
.output series.md
SELECT
    *
FROM generate_series(5);
```

To execute it from the CLI, the `.read` command is used as before.

```
.read write_markdown_to_file.sql
```

In this case, no output is returned to the terminal. Instead, the file `series.md` is created (or replaced if it already existed) with the markdown-formatted results shown here:

```
| generate_series |
|-----------------|
| 0               |
| 1               |
| 2               |
| 3               |
| 4               |
| 5               |
```

Configuring the CLI

The various dot commands above can be used to configure the CLI. On start-up, the CLI reads and executes all commands in the file `~/.duckdbrc`. This allows you to store the configuration state

of the CLI. This file is passed to a `.read` command at startup, so any series of dot commands and SQL commands may be included. You may also point to a different initialization file using the `-init` switch.

As an example, a file in the same directory as the DuckDB CLI named `select_example` will change the DuckDB prompt to be a duck head and run a SQL statement. Note that the duck head is built with unicode characters and does not always work in all terminal environments (like Windows, unless running with WSL and using the Windows Terminal).

```
-- Duck head prompt
.prompt '⚫◗ '
-- Example SQL statement
SELECT 'Begin quacking!' AS "Ready, Set, ...";
```

To invoke that file on initialization, use this command:

```
$ ./duckdb -init select_example
```

This outputs:

```
-- Loading resources from /home/<user>/.duckdbrc
┌──────────────────┐
│ Ready, Set, ...  │
│     varchar      │
├──────────────────┤
│ Begin quacking!  │
└──────────────────┘
v<version> <git hash>
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
⚫◗
```

Non-interactive Usage

To read/process a file and exit immediately, pipe the file contents in to `duckdb`:

```
$ ./duckdb < select_example.sql
```

```
| generate_series |
|-----------------|
| 0               |
| 1               |
| 2               |
```

```
| 3                |
| 4                |
| 5                |
```

To execute a command with SQL text passed in directly from the command line, call `duckdb` with two arguments: the database location (or `:memory:`), and a string with the SQL statement to execute.

```
./duckdb :memory: "SELECT 42 AS the_answer"
```

```
┌────────────┐
│ the_answer │
│   int32    │
├────────────┤
│         42 │
└────────────┘
```

Loading Extensions

The CLI does not use the SQLite shell's `.load` command. Instead, directly execute DuckDB's SQL `install` and `load` commands as you would other SQL statements. See the Extension docs for details.

```
INSTALL fts;
LOAD fts;
```

Reading from stdin and Writing to stdout

When in a Unix environment, it can be useful to pipe data between multiple commands. DuckDB is able to read data from stdin as well as write to stdout using the file location of stdin (`/dev/stdin`) and stdout (`/dev/stdout`) within SQL commands, as pipes act very similarly to file handles.

This command will create an example CSV:

```
COPY (SELECT 42 AS woot UNION ALL SELECT 43 AS woot) TO 'test.csv' (HEADER);
```

First, read a file and pipe it to the `duckdb` CLI executable. As arguments to the DuckDB CLI, pass in the location of the database to open, in this case, an in memory database, and a SQL command that utilizes `/dev/stdin` as a file location.

```
cat test.csv | ./duckdb :memory: "SELECT * FROM read_csv_auto('/dev/stdin')"
```

```
┌───────┐
│ woot  │
│ int32 │
```

```
┌─────────┐
│ column0 │
│  int64  │
├─────────┤
│      42 │
│      43 │
└─────────┘
```

To write back to stdout, the copy command can be used with the `/dev/stdout` file location.

```
cat test.csv | ./duckdb :memory: "COPY (SELECT * FROM read_csv_
 ↪  auto('/dev/stdin')) TO '/dev/stdout' WITH (FORMAT 'csv', HEADER)"
```

```
woot
42
43
```

Java JDBC API

Installation

The DuckDB Java JDBC API can be installed from Maven Central. Please see the installation page for details.

Basic API Usage

DuckDB's JDBC API implements the main parts of the standard Java Database Connectivity (JDBC) API, version 4.1. Describing JDBC is beyond the scope of this page, see the official documentation for details. Below we focus on the DuckDB-specific parts.

Refer to the externally hosted API Reference for more information about our extensions to the JDBC specification, or the below Arrow Methods

Startup & Shutdown In JDBC, database connections are created through the standard `java.sql.DriverManager` class. The driver should auto-register in the DriverManager, if that does not work for some reason, you can enforce registration like so:

```
Class.forName("org.duckdb.DuckDBDriver");
```

To create a DuckDB connection, call `DriverManager` with the `jdbc:duckdb:` JDBC URL prefix, like so:

```
Connection conn = DriverManager.getConnection("jdbc:duckdb:");
```

When using the `jdbc:duckdb:` URL alone, an **in-memory database** is created. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the Java program). If you would like to access or create a persistent database, append its file name after the path. For example, if your database is stored in `/tmp/my_database`, use the JDBC URL `jdbc:duckdb:/tmp/my_database` to create a connection to it.

It is possible to open a DuckDB database file in **read-only** mode. This is for example useful if multiple Java processes want to read the same database file at the same time. To open an existing database file in read-only mode, set the connection property `duckdb.read_only` like so:

```
Properties ro_prop = new Properties();
ro_prop.setProperty("duckdb.read_only", "true");
Connection conn_ro = DriverManager.getConnection("jdbc:duckdb:/tmp/my_
 ↪  database", ro_prop);
```

Additional connections can be created using the `DriverManager`. A more efficient mechanism is to call the `DuckDBConnecttion#duplicate()` method like so:

```
Connection conn2 = ((DuckDBConnection) conn).duplicate();
```

Multiple connections are allowed, but mixing read-write and read-only connections is unsupported.

Querying DuckDB supports the standard JDBC methods to send queries and retrieve result sets. First a `Statement` object has to be created from the `Connection`, this object can then be used to send queries using `execute` and `executeQuery`. `execute()` is meant for queries where no results are expected like `CREATE TABLE` or `UPDATE` etc. and `executeQuery()` is meant to be used for queries that produce results (e.g., `SELECT`). Below two examples. See also the JDBC `Statement` and `ResultSet` documentations.

```
// create a table
Statement stmt = conn.createStatement();
stmt.execute("CREATE TABLE items (item VARCHAR, value DECIMAL(10, 2), count
 ↪  INTEGER)");
// insert two items into the table
stmt.execute("INSERT INTO items VALUES ('jeans', 20.0, 1), ('hammer', 42.2,
 ↪  2)");

try (ResultSet rs = stmt.executeQuery("SELECT * FROM items")) {
    while (rs.next()) {
        System.out.println(rs.getString(1));
        System.out.println(rs.getInt(3));
    }
}
```

```
// jeans
// 1
// hammer
// 2
```

DuckDB also supports prepared statements as per the JDBC API:

```
try (PreparedStatement p_stmt = conn.prepareStatement("INSERT INTO items
 ↪  VALUES (?, ?, ?);")) {
    p_stmt.setString(1, "chainsaw");
    p_stmt.setDouble(2, 500.0);
    p_stmt.setInt(3, 42);
    p_stmt.execute();

    // more calls to execute() possible
}
```

> **Note.** Do *not* use prepared statements to insert large amounts of data into DuckDB. See the data import documentation for better options.

Arrow Methods Refer to the API Reference for type signatures

Arrow Export The following demonstrates exporting an arrow stream and consuming it using the java arrow bindings

```
import org.apache.arrow.memory.RootAllocator;
import org.apache.arrow.vector.ipc.ArrowReader;
import org.duckdb.DuckDBResultSet;

try (var conn = DriverManager.getConnection("jdbc:duckdb:");
     var p_stmt = conn.prepareStatement("SELECT * FROM generate_
      ↪  series(2000)");
     var resultset = (DuckDBResultSet) p_stmt.executeQuery();
     var allocator = new RootAllocator()) {
  try (var reader = (ArrowReader) resultset.arrowExportStream(allocator,
   ↪  256)) {
    while (reader.loadNextBatch()) {
      System.out.println(reader.getVectorSchemaRoot().getVector("generate_
       ↪  series"));
    }
  }
}
```

Arrow Import The following demonstrates consuming an arrow stream from the java arrow bindings

```
import org.apache.arrow.memory.RootAllocator;
import org.apache.arrow.vector.ipc.ArrowReader;
import org.duckdb.DuckDBConnection;

// arrow stuff
try (var allocator = new RootAllocator();
     ArrowStreamReader reader = null; /* should not be null of course */
     var arrow_array_stream = ArrowArrayStream.allocateNew(allocator)) {
  Data.exportArrayStream(allocator, reader, arrow_array_stream);

  // duckdb stuff
  try (var conn = (DuckDBConnection)
   ↪  DriverManager.getConnection("jdbc:duckdb:")) {
    conn.registerArrowStream("adsf", arrow_array_stream);

    // run a query
    try (var stmt = conn.createStatement();
         var rs = (DuckDBResultSet) stmt.executeQuery("SELECT count(*) FROM
          ↪  adsf")) {
      while (rs.next()) {
        System.out.println(rs.getInt(1));
      }
    }
  }
}
```

Streaming results Result streaming is opt-in in the JDBC driver - by setting the `jdbc_stream_results` config to `true` before running a query. The easiest way do that is to pass it in the `Properties` object.

```
Properties props = new Properties();
props.setProperty(DuckDBDriver.JDBC_STREAM_RESULTS, String.valueOf(true));

Connection conn = DriverManager.getConnection("jdbc:duckdb:", props);
```

Julia Package

The DuckDB Julia package provides a high-performance front-end for DuckDB. Much like SQLite, DuckDB runs in-process within the Julia client, and provides a DBInterface front-end.

The package also supports multi-threaded execution. It uses Julia threads/tasks for this purpose. If you wish to run queries in parallel, you must launch Julia with multi-threading support (by e.g., setting the `JULIA_NUM_THREADS` environment variable).

Installation

Install DuckDB as follows:

```
using Pkg
Pkg.add("DuckDB")
```

Alternatively:

```
pkg> add DuckDB
```

Basics

```
using DuckDB

# create a new in-memory database
con = DBInterface.connect(DuckDB.DB, ":memory:")

# create a table
DBInterface.execute(con, "CREATE TABLE integers(i INTEGER)")

# insert data using a prepared statement
stmt = DBInterface.prepare(con, "INSERT INTO integers VALUES(?)")
DBInterface.execute(stmt, [42])

# query the database
results = DBInterface.execute(con, "SELECT 42 a")
print(results)
```

Scanning DataFrames

The DuckDB Julia package also provides support for querying Julia DataFrames. Note that the DataFrames are directly read by DuckDB - they are not inserted or copied into the database itself.

If you wish to load data from a DataFrame into a DuckDB table you can run a `CREATE TABLE AS` or `INSERT INTO` query.

```
using DuckDB
using DataFrames

# create a new in-memory dabase
con = DBInterface.connect(DuckDB.DB)

# create a DataFrame
df = DataFrame(a = [1, 2, 3], b = [42, 84, 42])

# register it as a view in the database
DuckDB.register_data_frame(con, df, "my_df")

# run a SQL query over the DataFrame
results = DBInterface.execute(con, "SELECT * FROM my_df")
print(results)
```

Original Julia Connector

Credits to kimmolinna for the original DuckDB Julia connector.

Node.js

Node.js API

This package provides a Node.js API for DuckDB. The API for this client is somewhat compliant to the SQLite node.js client for easier transition.

Load the package and create a database object:

```
const duckdb = require('duckdb');
const db = new duckdb.Database(':memory:'); // or a file name for a
 ↪  persistent DB
```

All options as described on Database configuration can be (optionally) supplied to the `Database` constructor as second argument. The third argument can be optionally supplied to get feedback on the given options.

```
const db = new duckdb.Database(':memory:', {
    "access_mode": "READ_WRITE",
```

```
    "max_memory": "512MB",
    "threads": "4"
}, (err) => {
  if (err) {
    console.error(err);
  }
});
```

Then you can run a query:

```
db.all('SELECT 42 AS fortytwo', function(err, res) {
  if (err) {
    throw err;
  }
  console.log(res[0].fortytwo)
});
```

Other available methods are `each`, where the callback is invoked for each row, `run` to execute a single statement without results and `exec`, which can execute several SQL commands at once but also does not return results. All those commands can work with prepared statements, taking the values for the parameters as additional arguments. For example like so:

```
db.all('SELECT ?::INTEGER AS fortytwo, ?::STRING AS hello', 42, 'Hello,
 ↪  World', function(err, res) {
  if (err) {
    throw err;
  }
  console.log(res[0].fortytwo)
  console.log(res[0].hello)
});
```

However, these are all shorthands for something much more elegant. A database can have multiple `Connection`s, those are created using `db.connect()`.

```
const con = db.connect();
```

You can create multiple connections, each with their own transaction context.

`Connection` objects also contain shorthands to directly call `run()`, `all()` and `each()` with parameters and callbacks, respectively, for example:

```
con.all('SELECT 42 AS fortytwo', function(err, res) {
  if (err) {
    throw err;
  }
```

```
  console.log(res[0].fortytwo)
});
```

From connections, you can create prepared statements (and only that) using `con.prepare()`:

```
const stmt = con.prepare('select ?::INTEGER as fortytwo');
```

To execute this statement, you can call for example `all()` on the `stmt` object:

```
stmt.all(42, function(err, res) {
  if (err) {
    throw err;
  }
  console.log(res[0].fortytwo)
});
```

You can also execute the prepared statement multiple times. This is for example useful to fill a table with data:

```
con.run('CREATE TABLE a (i INTEGER)');
const stmt = con.prepare('INSERT INTO a VALUES (?)');
for (let i = 0; i < 10; i++) {
  stmt.run(i);
}
stmt.finalize();
con.all('SELECT * FROM a', function(err, res) {
  if (err) {
    throw err;
  }
  console.log(res)
});
```

`prepare()` can also take a callback which gets the prepared statement as an argument:

```
const stmt = con.prepare('select ?::INTEGER as fortytwo', function(err,
 ↪  stmt) {
  stmt.all(42, function(err, res) {
    if (err) {
      throw err;
    }
    console.log(res[0].fortytwo)
  });
});
```

Apache Arrow can be used to insert data into DuckDB without making a copy:

```
const arrow = require('apache-arrow');
const db = new duckdb.Database(':memory:');

const jsonData = [
  {"userId":1,"id":1,"title":"delectus aut autem","completed":false},
  {"userId":1,"id":2,"title":"quis ut nam facilis et officia
 ↪  qui","completed":false}
];

// note; doesn't work on Windows yet
db.exec(` INSTALL arrow; LOAD arrow;`, (err) => {
    if (err) {
        throw err;
    }

    const arrowTable = arrow.tableFromJSON(jsonData);
    db.register_buffer("jsonDataTable", [arrow.tableToIPC(arrowTable)],
 ↪  true, (err, res) => {
        if (err) {
            throw err;
        }

        // `SELECT * FROM jsonDataTable` would return the entries in
         ↪  `jsonData`
    });
});
```

NodeJS API

Modules

Typedefs

duckdb

Summary: DuckDB is an embeddable SQL OLAP Database Management System

- duckdb
  - ~Connection
    - .run(sql, ...params, callback) ⇒ void
    - .all(sql, ...params, callback) ⇒ void

    - .arrowIPCAll(sql, ...params, callback) ⇒ void
    - .arrowIPCStream(sql, ...params, callback) ⇒
    - .each(sql, ...params, callback) ⇒ void
    - .stream(sql, ...params)
    - .register_udf(name, return_type, fun) ⇒ void
    - .prepare(sql, ...params, callback) ⇒ Statement
    - .exec(sql, ...params, callback) ⇒ void
    - .register_udf_bulk(name, return_type, callback) ⇒ void
    - .unregister_udf(name, return_type, callback) ⇒ void
    - .register_buffer(name, array, force, callback) ⇒ void
    - .unregister_buffer(name, callback) ⇒ void
    - .close(callback) ⇒ void
- ~Statement
    - .sql ⇒
    - .get()
    - .run(sql, ...params, callback) ⇒ void
    - .all(sql, ...params, callback) ⇒ void
    - .arrowIPCAll(sql, ...params, callback) ⇒ void
    - .each(sql, ...params, callback) ⇒ void
    - .finalize(sql, ...params, callback) ⇒ void
    - .stream(sql, ...params)
    - .columns() ⇒ Array.<ColumnInfo>
- ~QueryResult
    - .nextChunk() ⇒
    - .nextIpcBuffer() ⇒
    - .asyncIterator()
- ~Database
    - .close(callback) ⇒ void
    - .close_internal(callback) ⇒ void
    - .wait(callback) ⇒ void
    - .serialize(callback) ⇒ void
    - .parallelize(callback) ⇒ void
    - .connect(path) ⇒ Connection
    - .interrupt(callback) ⇒ void
    - .prepare(sql) ⇒ Statement
    - .run(sql, ...params, callback) ⇒ void
    - .scanArrowIpc(sql, ...params, callback) ⇒ void

        * .each(sql, ...params, callback) ⇒ void
        * .all(sql, ...params, callback) ⇒ void
        * .arrowIPCAll(sql, ...params, callback) ⇒ void
        * .arrowIPCStream(sql, ...params, callback) ⇒ void
        * .exec(sql, ...params, callback) ⇒ void
        * .register_udf(name, return_type, fun) ⇒ this
        * .register_buffer(name) ⇒ this
        * .unregister_buffer(name) ⇒ this
        * .unregister_udf(name) ⇒ this
        * .registerReplacementScan(fun) ⇒ this
        * .tokenize(text) ⇒ ScriptTokens
        * .get()
    - ~TokenType
    - ~ERROR : number
    - ~OPEN_READONLY : number
    - ~OPEN_READWRITE : number
    - ~OPEN_CREATE : number
    - ~OPEN_FULLMUTEX : number
    - ~OPEN_SHAREDCACHE : number
    - ~OPEN_PRIVATECACHE : number

duckdb~Connection **Kind**: inner class of duckdb

- ~Connection
    - .run(sql, ...params, callback) ⇒ void
    - .all(sql, ...params, callback) ⇒ void
    - .arrowIPCAll(sql, ...params, callback) ⇒ void
    - .arrowIPCStream(sql, ...params, callback) ⇒
    - .each(sql, ...params, callback) ⇒ void
    - .stream(sql, ...params)
    - .register_udf(name, return_type, fun) ⇒ void
    - .prepare(sql, ...params, callback) ⇒ Statement
    - .exec(sql, ...params, callback) ⇒ void
    - .register_udf_bulk(name, return_type, callback) ⇒ void
    - .unregister_udf(name, return_type, callback) ⇒ void
    - .register_buffer(name, array, force, callback) ⇒ void
    - .unregister_buffer(name, callback) ⇒ void
    - .close(callback) ⇒ void

connection.run(sql, ...params, callback) ⇒ void Run a SQL statement and trigger a callback when done

Kind: instance method of Connection

Param	Type
sql	
...params	*
callback	

connection.all(sql, ...params, callback) ⇒ void Run a SQL query and triggers the callback once for all result rows

Kind: instance method of Connection

Param	Type
sql	
...params	*
callback	

connection.arrowIPCAll(sql, ...params, callback) ⇒ void Run a SQL query and serialize the result into the Apache Arrow IPC format (requires arrow extension to be loaded)

Kind: instance method of Connection

Param	Type
sql	
...params	*
callback	

connection.arrowIPCStream(sql, ...params, callback) ⇒ Run a SQL query, returns a IpcResultStreamIterator that allows streaming the result into the Apache Arrow IPC format (requires arrow extension to be loaded)

Kind: instance method of Connection
Returns: Promise

Param	Type
sql	
...params	*
callback	

connection.each(sql, ...params, callback) ⇒ void Runs a SQL query and triggers the callback for each result row

Kind: instance method of Connection

Param	Type
sql	
...params	*
callback	

connection.stream(sql, ...params) **Kind**: instance method of Connection

Param	Type
sql	
...params	*

connection.register_udf(name, return_type, fun) ⇒ void Register a User Defined Function

Kind: instance method of Connection
Note: this follows the wasm udfs somewhat but is simpler because we can pass data much more cleanly

Param
name
return_type
fun

connection.prepare(sql, ...params, callback) ⇒ Statement Prepare a SQL query for execution

Kind: instance method of Connection

Param	Type
sql	
...params	*
callback	

connection.exec(sql, ...params, callback) ⇒ void Execute a SQL query

Kind: instance method of Connection

Param	Type
sql	
...params	*
callback	

connection.register_udf_bulk(name, return_type, callback) ⇒ void Register a User Defined Function

Kind: instance method of Connection

Param
name
return_type

Param
callback

connection.unregister_udf(name, return_type, callback) ⇒ void Unregister a User Defined Function

Kind: instance method of Connection

Param
name
return_type
callback

connection.register_buffer(name, array, force, callback) ⇒ void Register a Buffer to be scanned using the Apache Arrow IPC scanner (requires arrow extension to be loaded)

Kind: instance method of Connection

Param
name
array
force
callback

connection.unregister_buffer(name, callback) ⇒ void Unregister the Buffer

Kind: instance method of Connection

Param
name
callback

connection.close(callback) ⇒ void Closes connection

Kind: instance method of Connection

Param
callback

duckdb~Statement **Kind**: inner class of duckdb

- ~Statement
  - .sql ⇒
  - .get()
  - .run(sql, ...params, callback) ⇒ void
  - .all(sql, ...params, callback) ⇒ void
  - .arrowIPCAll(sql, ...params, callback) ⇒ void
  - .each(sql, ...params, callback) ⇒ void
  - .finalize(sql, ...params, callback) ⇒ void
  - .stream(sql, ...params)
  - .columns() ⇒ Array.<ColumnInfo>

statement.sql ⇒ **Kind**: instance property of Statement
Returns: sql contained in statement
Field:

statement.get() Not implemented

Kind: instance method of Statement

statement.run(sql, ...params, callback) ⇒ void **Kind**: instance method of Statement

Param	Type
sql	
...params	*

Param	Type
callback	

statement.all(sql, ...params, callback) ⇒ void **Kind**: instance method of Statement

Param	Type
sql	
...params	*
callback	

statement.arrowIPCAll(sql, ...params, callback) ⇒ void **Kind**: instance method of Statement

Param	Type
sql	
...params	*
callback	

statement.each(sql, ...params, callback) ⇒ void **Kind**: instance method of Statement

Param	Type
sql	
...params	*
callback	

statement.finalize(sql, ...params, callback) ⇒ void **Kind**: instance method of Statement

Param	Type
sql	
...params	*
callback	

statement.stream(sql, ...params) **Kind**: instance method of Statement

Param	Type
sql	
...params	*

statement.columns() ⇒ Array.<ColumnInfo> **Kind**: instance method of Statement
Returns: Array.<ColumnInfo> - - Array of column names and types

duckdb~QueryResult **Kind**: inner class of duckdb

- ~QueryResult
  - .nextChunk() ⇒
  - .nextIpcBuffer() ⇒
  - .asyncIterator()

queryResult.nextChunk() ⇒ **Kind**: instance method of QueryResult
Returns: data chunk

queryResult.nextIpcBuffer() ⇒ Function to fetch the next result blob of an Arrow IPC Stream in a zero-copy way. (requires arrow extension to be loaded)

Kind: instance method of QueryResult
Returns: data chunk

queryResult.asyncIterator() **Kind**: instance method of QueryResult

duckdb~Database Main database interface

Kind: inner property of duckdb

Param	Description
path	path to database file or :memory: for in-memory database
access_mode	access mode
config	the configuration object
callback	callback function

- ~Database
  - .close(callback) ⇒ void
  - .close_internal(callback) ⇒ void
  - .wait(callback) ⇒ void
  - .serialize(callback) ⇒ void
  - .parallelize(callback) ⇒ void
  - .connect(path) ⇒ Connection
  - .interrupt(callback) ⇒ void
  - .prepare(sql) ⇒ Statement
  - .run(sql, ...params, callback) ⇒ void
  - .scanArrowIpc(sql, ...params, callback) ⇒ void
  - .each(sql, ...params, callback) ⇒ void
  - .all(sql, ...params, callback) ⇒ void
  - .arrowIPCAll(sql, ...params, callback) ⇒ void
  - .arrowIPCStream(sql, ...params, callback) ⇒ void
  - .exec(sql, ...params, callback) ⇒ void
  - .register_udf(name, return_type, fun) ⇒ this
  - .register_buffer(name) ⇒ this
  - .unregister_buffer(name) ⇒ this
  - .unregister_udf(name) ⇒ this
  - .registerReplacementScan(fun) ⇒ this
  - .tokenize(text) ⇒ ScriptTokens
  - .get()

database.close(callback) ⇒ void Closes database instance

Kind: instance method of Database

Param
callback

database.close_internal(callback) ⇒ void Internal method. Do not use, call Connection#close instead

Kind: instance method of Database

Param
callback

database.wait(callback) ⇒ void Triggers callback when all scheduled database tasks have completed.

Kind: instance method of Database

Param
callback

database.serialize(callback) ⇒ void Currently a no-op. Provided for SQLite compatibility

Kind: instance method of Database

Param
callback

database.parallelize(callback) ⇒ void Currently a no-op. Provided for SQLite compatibility

Kind: instance method of Database

Param
callback

database.connect(path) ⇒ Connection Create a new database connection

Kind: instance method of Database

Param	Description
path	the database to connect to, either a file path, or `:memory:`

database.interrupt(callback) ⇒ void Supposedly interrupt queries, but currently does not do anything.

Kind: instance method of Database

Param
callback

database.prepare(sql) ⇒ Statement Prepare a SQL query for execution

Kind: instance method of Database

Param
sql

database.run(sql, ...params, callback) ⇒ void Convenience method for Connection#run using a built-in default connection

Kind: instance method of Database

Param	Type
sql	

Param	Type
...params	*
callback	

database.scanArrowIpc(sql, ...params, callback) ⇒ void Convenience method for Connection#scanArrowIpc using a built-in default connection

Kind: instance method of Database

Param	Type
sql	
...params	*
callback	

database.each(sql, ...params, callback) ⇒ void **Kind**: instance method of Database

Param	Type
sql	
...params	*
callback	

database.all(sql, ...params, callback) ⇒ void Convenience method for Connection#apply using a built-in default connection

Kind: instance method of Database

Param	Type
sql	
...params	*
callback	

database.arrowIPCAll(sql, ...params, callback) ⇒ void Convenience method for Connection#arrowIPCAll using a built-in default connection

Kind: instance method of Database

Param	Type
sql	
...params	*
callback	

database.arrowIPCStream(sql, ...params, callback) ⇒ void Convenience method for Connection#arrowIPCStream using a built-in default connection

Kind: instance method of Database

Param	Type
sql	
...params	*
callback	

database.exec(sql, ...params, callback) ⇒ void **Kind**: instance method of Database

Param	Type
sql	
...params	*
callback	

database.register_udf(name, return_type, fun) ⇒ this Register a User Defined Function

Convenience method for Connection#register_udf

Kind: instance method of Database

Param
name
return_type
fun

database.register_buffer(name) ⇒ this Register a buffer containing serialized data to be scanned from DuckDB.

Convenience method for Connection#unregister_buffer

Kind: instance method of Database

Param
name

database.unregister_buffer(name) ⇒ this Unregister a Buffer

Convenience method for Connection#unregister_buffer

Kind: instance method of Database

Param
name

database.unregister_udf(name) ⇒ this Unregister a UDF

Convenience method for Connection#unregister_udf

Kind: instance method of Database

Param
name

database.registerReplacementScan(fun) ⇒ this Register a table replace scan function

Kind: instance method of Database

Param	Description
fun	Replacement scan function

database.tokenize(text) ⇒ ScriptTokens Return positions and types of tokens in given text

Kind: instance method of Database

Param
text

database.get() Not implemented

Kind: instance method of Database

duckdb~TokenType Types of tokens return by `tokenize`.

Kind: inner property of duckdb

duckdb~ERROR : number Check that errno attribute equals this to check for a duckdb error

Kind: inner constant of duckdb

duckdb~OPEN_READONLY : number Open database in readonly mode

Kind: inner constant of duckdb

duckdb~OPEN_READWRITE : number Currently ignored

Kind: inner constant of duckdb

duckdb~OPEN_CREATE : number Currently ignored

Kind: inner constant of duckdb

duckdb~OPEN_FULLMUTEX : number Currently ignored

Kind: inner constant of duckdb

duckdb~OPEN_SHAREDCACHE : number Currently ignored

Kind: inner constant of duckdb

duckdb~OPEN_PRIVATECACHE : number Currently ignored

Kind: inner constant of duckdb

ColumnInfo : object

Kind: global typedef
Properties

Name	Type	Description
name	string	Column name
type	TypeInfo	Column type

TypeInfo : object

Kind: global typedef
Properties

Name	Type	Description
id	string	Type ID

Name	Type	Description
[alias]	string	SQL type alias
sql_type	string	SQL type name

DuckDbError : object

Kind: global typedef
Properties

Name	Type	Description
errno	number	-1 for DuckDB errors
message	string	Error message
code	string	'DUCKDB_NODEJS_ERROR' for DuckDB errors
errorType	string	DuckDB error type code (eg, HTTP, IO, Catalog)

HTTPError : object

Kind: global typedef
Extends: DuckDbError
Properties

Name	Type	Description
statusCode	number	HTTP response status code
reason	string	HTTP response reason
response	string	HTTP response body
headers	object	HTTP headers

Python

Python API

Installation

The DuckDB Python API can be installed using pip: `pip install duckdb`. Please see the installation page for details. It is also possible to install DuckDB using conda: `conda install python-duckdb -c conda-forge`.

You must be using Python 3.7 or newer.

Basic API Usage

The most straight-forward manner of running SQL queries using DuckDB is using the `duckdb.sql` command.

```
import duckdb
duckdb.sql('SELECT 42').show()
```

This will run queries using an **in-memory database** that is stored globally inside the Python module. The result of the query is returned as a **Relation**. A relation is a symbolic representation of the query. The query is not executed until the result is fetched or requested to be printed to the screen.

Relations can be referenced in subsequent queries by storing them inside variables, and using them as tables. This way queries can be constructed incrementally.

```
import duckdb
r1 = duckdb.sql('SELECT 42 AS i')
duckdb.sql('SELECT i * 2 AS k FROM r1').show()
```

Data Input

DuckDB can ingest data from a wide variety of formats – both on-disk and in-memory. See the data ingestion page for more information.

```
import duckdb
duckdb.read_csv('example.csv')                # read a CSV file into a
↪ Relation
duckdb.read_parquet('example.parquet')        # read a Parquet file into a
↪ Relation
```

```
duckdb.read_json('example.json')               # read a JSON file into a
 ↪  Relation

duckdb.sql('SELECT * FROM "example.csv"')      # directly query a CSV file
duckdb.sql('SELECT * FROM "example.parquet"') # directly query a Parquet
 ↪  file
duckdb.sql('SELECT * FROM "example.json"')     # directly query a JSON file
```

DataFrames DuckDB can also directly query Pandas DataFrames, Polars DataFrames and Arrow tables.

```
import duckdb

# directly query a Pandas DataFrame
import pandas as pd
pandas_df = pd.DataFrame({'a': [42]})
duckdb.sql('SELECT * FROM pandas_df')

# directly query a Polars DataFrame
import polars as pl
polars_df = pl.DataFrame({'a': [42]})
duckdb.sql('SELECT * FROM polars_df')

# directly query a pyarrow table
import pyarrow as pa
arrow_table = pa.Table.from_pydict({'a':[42]})
duckdb.sql('SELECT * FROM arrow_table')
```

Result Conversion

DuckDB supports converting query results efficiently to a variety of formats. See the result conversion page for more information.

```
import duckdb
duckdb.sql('SELECT 42').fetchall()   # Python objects
duckdb.sql('SELECT 42').df()         # Pandas DataFrame
duckdb.sql('SELECT 42').pl()         # Polars DataFrame
duckdb.sql('SELECT 42').arrow()      # Arrow Table
duckdb.sql('SELECT 42').fetchnumpy() # NumPy Arrays
```

Writing Data to Disk

DuckDB supports writing Relation objects directly to disk in a variety of formats. The COPY statement can be used to write data to disk using SQL as an alternative.

```
import duckdb
duckdb.sql('SELECT 42').write_parquet('out.parquet') # Write to a Parquet
↪  file
duckdb.sql('SELECT 42').write_csv('out.csv')         # Write to a CSV file
duckdb.sql("COPY (SELECT 42) TO 'out.parquet'")      # Copy to a parquet file
```

Using an In-memory Database

When using DuckDB through `duckdb.sql()`, it operates on an **in-memory** database, i.e., no tables are persisted on disk. The `duckdb.connect()` method returns a connection to an in-memory database:

```
import duckdb

con = duckdb.connect()
con.sql('SELECT 42 AS x').show()
```

Persistent Storage

The `duckdb.connect(`*dbname*`)` creates a connection to a **persistent** database. Any data written to that connection will be persisted, and can be reloaded by re-connecting to the same file, both from Python and from other DuckDB clients.

```
import duckdb

# create a connection to a file called 'file.db'
con = duckdb.connect('file.db')
# create a table and load data into it
con.sql('CREATE TABLE test(i INTEGER)')
con.sql('INSERT INTO test VALUES (42)')
# query the table
con.table('test').show()
# explicitly close the connection
con.close()
# Note: connections also closed implicitly when they go out of scope
```

You can also use a context manager to ensure that the connection is closed:

```
import duckdb

with duckdb.connect('file.db') as con:
    con.sql('CREATE TABLE test(i INTEGER)')
    con.sql('INSERT INTO test VALUES (42)')
    con.table('test').show()
    # the context manager closes the connection automatically
```

Connection Object and Module

The connection object and the `duckdb` module can be used interchangeably – they support the same methods. The only difference is that when using the `duckdb` module a global in-memory database is used.

Note that if you are developing a package designed for others to use, and use DuckDB in the package, it is recommend that you create connection objects instead of using the methods on the `duckdb` module. That is because the `duckdb` module uses a shared global database – which can cause hard to debug issues if used from within multiple different packages.

Loading and Installing Extensions

DuckDB's Python API provides functions for installing and loading extensions, which perform the equivalent operations to running the `INSTALL` and `LOAD` SQL commands, respectively. An example that installs and loads the `spatial` extension looks like follows:

```
import duckdb

con = duckdb.connect()
con.install_extension("spatial")
con.load_extension("spatial")
```

Data Ingestion

CSV Files

CSV files can be read using the `read_csv` function, called either from within Python or directly from within SQL. By default, the `read_csv` function attempts to auto-detect the CSV settings by sampling from the provided file.

```python
import duckdb
# read from a file using fully auto-detected settings
duckdb.read_csv('example.csv')
# read multiple CSV files from a folder
duckdb.read_csv('folder/*.csv')
# specify options on how the CSV is formatted internally
duckdb.read_csv('example.csv', header=False, sep=',')
# override types of the first two columns
duckdb.read_csv('example.csv', dtype=['int', 'varchar'])
# use the (experimental) parallel CSV reader
duckdb.read_csv('example.csv', parallel=True)
# directly read a CSV file from within SQL
duckdb.sql("SELECT * FROM 'example.csv'")
# call read_csv from within SQL
duckdb.sql("SELECT * FROM read_csv_auto('example.csv')")
```

See the CSV Import page for more information.

Parquet Files

Parquet files can be read using the `read_parquet` function, called either from within Python or directly from within SQL.

```python
import duckdb
# read from a single Parquet file
duckdb.read_parquet('example.parquet')
# read multiple Parquet files from a folder
duckdb.read_parquet('folder/*.parquet')
# directly read a Parquet file from within SQL
duckdb.sql("SELECT * FROM 'example.parquet'")
# call read_parquet from within SQL
duckdb.sql("SELECT * FROM read_parquet('example.parquet')")
```

See the Parquet Loading page for more information.

JSON Files

JSON files can be read using the `read_json` function, called either from within Python or directly from within SQL. By default, the `read_json` function will automatically detect if a file contains newline-delimited JSON or regular JSON, and will detect the schema of the objects stored within the JSON file.

```
import duckdb
# read from a single JSON file
duckdb.read_json('example.json')
# read multiple JSON files from a folder
duckdb.read_json('folder/*.json')
# directly read a JSON file from within SQL
duckdb.sql("SELECT * FROM 'example.json'")
# call read_json from within SQL
duckdb.sql("SELECT * FROM read_json_auto('example.json')")
```

DataFrames & Arrow Tables

DuckDB is automatically able to query a Pandas DataFrame, Polars DataFrame, or Arrow object that is stored in a Python variable by name. DuckDB supports querying multiple types of Apache Arrow objects including tables, datasets, RecordBatchReaders, and scanners. See the Python guides for more examples.

```
import duckdb
import pandas as pd
test_df = pd.DataFrame.from_dict({"i":[1, 2, 3, 4], "j":["one", "two",
 ↪  "three", "four"]})
duckdb.sql('SELECT * FROM test_df').fetchall()
# [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```

DuckDB also supports "registering" a DataFrame or Arrow object as a virtual table, comparable to a SQL `VIEW`. This is useful when querying a DataFrame/Arrow object that is stored in another way (as a class variable, or a value in a dictionary). Below is a Pandas example:

If your Pandas DataFrame is stored in another location, here is an example of manually registering it:

```
import duckdb
import pandas as pd
my_dictionary = {}
my_dictionary['test_df'] = pd.DataFrame.from_dict({"i":[1, 2, 3, 4],
 ↪  "j":["one", "two", "three", "four"]})
duckdb.register('test_df_view', my_dictionary['test_df'])
duckdb.sql('SELECT * FROM test_df_view').fetchall()
# [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```

You can also create a persistent table in DuckDB from the contents of the DataFrame (or the view):

```
# create a new table from the contents of a DataFrame
con.execute('CREATE TABLE test_df_table AS SELECT * FROM test_df')
```

```
# insert into an existing table from the contents of a DataFrame
con.execute('INSERT INTO test_df_table SELECT * FROM test_df')
```

Pandas DataFrames – `object` Columns `pandas.DataFrame` columns of an `object` dtype require some special care, since this stores values of arbitrary type. To convert these columns to DuckDB, we first go through an analyze phase before converting the values. In this analyze phase a sample of all the rows of the column are analyzed to determine the target type. This sample size is by default set to 1000. If the type picked during the analyze step is incorrect, this will result in a "Failed to cast value:" error, in which case you will need to increase the sample size. The sample size can be changed by setting the `pandas_analyze_sample` config option.

```
# example setting the sample size to 100000
duckdb.default_connection.execute("SET GLOBAL pandas_analyze_sample=100000")
```

Object Conversion

This is a mapping of Python object types to DuckDB Logical Types:

- `None` -> `NULL`
- `bool` -> `BOOLEAN`
- `datetime.timedelta` -> `INTERVAL`
- `str` -> `VARCHAR`
- `bytearray` -> `BLOB`
- `memoryview` -> `BLOB`
- `decimal.Decimal` -> `DECIMAL` / `DOUBLE`
- `uuid.UUID` -> `UUID`

The rest of the conversion rules are as follows.

`int` Since integers can be of arbitrary size in Python, there is not a one-to-one conversion possible for ints. Intead we perform these casts in order until one succeeds:

- `BIGINT`
- `INTEGER`
- `UBIGINT`
- `UINTEGER`
- `DOUBLE`

When using the DuckDB Value class, it's possible to set a target type, which will influence the conversion.

`float` These casts are tried in order until one succeeds:

- DOUBLE
- FLOAT

`datetime.datetime` For `datetime` we will check `pandas.isnull` if it's available and return NULL if it returns true.
We check against `datetime.datetime.min` and `datetime.datetime.max` to convert to `-inf` and `+inf` respectively.

If the `datetime` has tzinfo, we will use `TIMESTAMPTZ`, otherwise it becomes `TIMESTAMP`.

`datetime.time` If the `time` has tzinfo, we will use `TIMETZ`, otherwise it becomes `TIME`.

`datetime.date` `date` converts to the DATE type.
We check against `datetime.date.min` and `datetime.date.max` to convert to `-inf` and `+inf` respectively.

`bytes` `bytes` converts to BLOB by default, when it's used to construct a Value object of type `BITSTRING`, it maps to `BITSTRING` instead.

`list` `list` becomes a `LIST` type of the "most permissive" type of its children, for example:

```
my_list_value = [
    12345,
    'test'
]
```

Will become VARCHAR`[]` because 12345 can convert to VARCHAR but `test` can not convert to `INTEGER`.

```
[12345, test]
```

`dict` The `dict` object can convert to either STRUCT`(...)` or MAP`(..., ...)` depending on its structure. If the dict has a structure similar to:

```
my_map_dict = {
    'key': [
        1, 2, 3
```

```
    ],
    'value': [
        'one', 'two', 'three'
    ]
}
```

Then we'll convert it to a `MAP` of key-value pairs of the two lists zipped together.
The example above becomes a `MAP(INTEGER, VARCHAR)`:

```
{1=one, 2=two, 3=three}
```

> **Note.** The name of the fields matters and the two lists need to have the same size.

Otherwise we'll try to convert it to a `STRUCT`.

```
my_struct_dict = {
    1: 'one',
    '2': 2,
    'three': [1,2,3],
    False: True
}
```

Becomes:

```
{'1': one, '2': 2, 'three': [1, 2, 3], 'False': true}
```

> **Note.** Every `key` of the dictionary is converted to string.

`tuple` `tuple` converts to `LIST` by default, when it's used to construct a Value object of type `STRUCT` it will convert to `STRUCT` instead.

`numpy.ndarray` and `numpy.datetime64` `ndarray` and `datetime64` are converted by calling `tolist()` and converting the result of that.

Result Conversion

DuckDB's Python client provides multiple additional methods that can be used to efficiently retrieve data.

NumPy

- `fetchnumpy()` fetches the data as a dictionary of NumPy arrays

Pandas

- `df()` fetches the data as a Pandas DataFrame
- `fetchdf()` is an alias of `df()`
- `fetch_df()` is an alias of `df()`
- `fetch_df_chunk(vector_multiple)` fetches a portion of the results into a DataFrame. The number of rows returned in each chunk is the vector size (2048 by default) * vector_multiple (1 by default).

Apache Arrow

- `arrow()` fetches the data as an Arrow table
- `fetch_arrow_table()` is an alias of `arrow()`
- `fetch_record_batch(chunk_size)` returns an Arrow record batch reader with `chunk_size` rows per batch

Polars

- `pl()` fetches the data as a Polars DataFrame

Below are some examples using this functionality. See the Python guides for more examples.

```
# fetch as Pandas DataFrame
df = con.execute("SELECT * FROM items").fetchdf()
print(df)
#        item   value  count
# 0     jeans    20.0      1
# 1    hammer    42.2      2
# 2    laptop  2000.0      1
# 3  chainsaw   500.0     10
# 4    iphone   300.0      2

# fetch as dictionary of numpy arrays
arr = con.execute("SELECT * FROM items").fetchnumpy()
print(arr)
# {'item': masked_array(data=['jeans', 'hammer', 'laptop', 'chainsaw',
 ↪  'iphone'],
#              mask=[False, False, False, False, False],
#        fill_value='?',
#             dtype=object), 'value': masked_array(data=[20.0, 42.2, 2000.0,
 ↪  500.0, 300.0],
```

```
#              mask=[False, False, False, False, False],
#        fill_value=1e+20), 'count': masked_array(data=[1, 2, 1, 10, 2],
#              mask=[False, False, False, False, False],
#        fill_value=999999,
#             dtype=int32)}

# fetch as an Arrow table. Converting to Pandas afterwards just for pretty
 ↪  printing
tbl = con.execute("SELECT * FROM items").fetch_arrow_table()
print(tbl.to_pandas())
#        item    value  count
# 0     jeans    20.00      1
# 1    hammer    42.20      2
# 2    laptop  2000.00      1
# 3  chainsaw   500.00     10
# 4    iphone   300.00      2
```

Python DB API

The standard DuckDB Python API provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249 similar to the SQLite Python API.

Connection

To use the module, you must first create a `DuckDBPyConnection` object that represents the database. The connection object takes as a parameter the database file to read and write from. If the database file does not exist, it will be created (the file extension may be `.db`, `.duckdb`, or anything else). The special value `:memory:` (the default) can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the Python process). If you would like to connect to an existing database in read-only mode, you can set the `read_only` flag to `True`. Read-only mode is required if multiple Python processes want to access the same database file at the same time.

By default we create an **in-memory-database** that lives inside the `duckdb` module. Every method of `DuckDBPyConnection` is also available on the `duckdb` module, this connection is what's used by these methods. You can also get a reference to this connection by providing the special value `:default:` to `connect`.

```
import duckdb

duckdb.execute('CREATE TABLE tbl AS SELECT 42 a')
```

```
con = duckdb.connect(':default:')
con.sql('SELECT * FROM tbl')
# ┌─────────┐
# │    a    │
# │  int32  │
# ├─────────┤
# │      42 │
# └─────────┘
```

```
import duckdb
# to start an in-memory database
con = duckdb.connect(database=':memory:')
# to use a database file (not shared between processes)
con = duckdb.connect(database='my-db.duckdb', read_only=False)
# to use a database file (shared between processes)
con = duckdb.connect(database='my-db.duckdb', read_only=True)
# to explicitly get the default connection
con = duckdb.connect(database=':default:')
```

If you want to create a second connection to an existing database, you can use the `cursor()` method. This might be useful for example to allow parallel threads running queries independently. A single connection is thread-safe but is locked for the duration of the queries, effectively serializing database access in this case.

Connections are closed implicitly when they go out of scope or if they are explicitly closed using `close()`. Once the last connection to a database instance is closed, the database instance is closed as well.

Querying

SQL queries can be sent to DuckDB using the `execute()` method of connections. Once a query has been executed, results can be retrieved using the `fetchone` and `fetchall` methods on the connection. `fetchall` will retrieve all results and complete the transaction. `fetchone` will retrieve a single row of results each time that it is invoked until no more results are available. The transaction will only close once `fetchone` is called and there are no more results remaining (the return value will be None). As an example, in the case of a query only returning a single row, `fetchone` should be called once to retrieve the results and a second time to close the transaction. Below are some short examples:

```
# create a table
con.execute("CREATE TABLE items(item VARCHAR, value DECIMAL(10, 2), count
 ↪  INTEGER)")
```

```
# insert two items into the table
con.execute("INSERT INTO items VALUES ('jeans', 20.0, 1), ('hammer', 42.2,
 ↪  2)")

# retrieve the items again
con.execute("SELECT * FROM items")
print(con.fetchall())
# [('jeans', Decimal('20.00'), 1), ('hammer', Decimal('42.20'), 2)]

# retrieve the items one at a time
con.execute("SELECT * FROM items")
print(con.fetchone())
# ('jeans', Decimal('20.00'), 1)
print(con.fetchone())
# ('hammer', Decimal('42.20'), 2)
print(con.fetchone()) # This closes the transaction. Any subsequent calls to
 ↪  .fetchone will return None
# None
```

The `description` property of the connection object contains the column names as per the standard.

DuckDB also supports prepared statements in the API with the `execute` and `executemany` methods. The values may be passed as an additional parameter after a query that contains `?` or `$1` (dollar symbol and a number) placeholders. Using the `?` notation adds the values in the same sequence as passed within the Python parameter. Using the `$` notation allows for values to be reused within the SQL statement based on the number and index of the value found within the Python parameter.

Here are some examples:

```
# insert a row using prepared statements
con.execute("INSERT INTO items VALUES (?, ?, ?)", ['laptop', 2000, 1])

# insert several rows using prepared statements
con.executemany("INSERT INTO items VALUES (?, ?, ?)", [['chainsaw', 500,
 ↪  10], ['iphone', 300, 2]] )

# query the database using a prepared statement
con.execute("SELECT item FROM items WHERE value > ?", [400])
print(con.fetchall())
# [('laptop',), ('chainsaw',)]

# query using $ notation for prepared statement and reused values
con.execute("SELECT $1, $1, $2", ["duck", "goose"])
```

```
print(con.fetchall())
# [('duck', 'duck', 'goose')]
```

Named Parameters

Besides the standard unnamed parameters, like `$1`, `$2` etc, it's also possible to supply named parameters, like `$my_parameter`.
When using named parameters, you have to provide a dictionary mapping of `str` to value in the `parameters` argument
An example use:

```
import duckdb

duckdb.execute("""
  SELECT
    $my_param,
    $other_param,
    $also_param
""",
  {
    'my_param': 5,
    'other_param': 'DuckDB',
    'also_param': [42]
  }
 ).fetchall()
# [(5, 'DuckDB', [42])]
```

> **Note.** Do *not* use `executemany` to insert large amounts of data into DuckDB. See the data ingestion page for better options.

Relational API

The Relational API is an alternative API that can be used to incrementally construct queries. The API is centered around `DuckDBPyRelation` nodes. The relations can be seen as symbolic representations of SQL queries. They do not hold any data - and nothing is executed - until a method that triggers execution is called.

Constructing Relations

Relations can be created from SQL queries using the `duckdb.sql` method. Alternatively, they can be created from the various data ingestion methods (`read_parquet`, `read_csv`, `read_json`).

For example, here we create a relation from a SQL query:

```
import duckdb
rel = duckdb.sql('SELECT * FROM range(10000000000) tbl(id)');
rel.show()
```

```
┌──────────────────────┐
│          id          │
│        int64         │
├──────────────────────┤
│                    0 │
│                    1 │
│                    2 │
│                    3 │
│                    4 │
│                    5 │
│                    6 │
│                    7 │
│                    8 │
│                    9 │
│                    · │
│                    · │
│                    · │
│                 9990 │
│                 9991 │
│                 9992 │
│                 9993 │
│                 9994 │
│                 9995 │
│                 9996 │
│                 9997 │
│                 9998 │
│                 9999 │
├──────────────────────┤
│        ? rows        │
│ (>9999 rows, 20 shown) │
└──────────────────────┘
```

Note how we are constructing a relation that computes an immense amount of data (`10B` rows, or

74GB of data). The relation is constructed instantly - and we can even print the relation instantly.

When printing a relation using `show` or displaying it in the terminal, the first `10K` rows are fetched. If there are more than `10K` rows, the output window will show `>9999 rows` (as the amount of rows in the relation is unknown).

Data Ingestion

Outside of SQL queries, the following methods are provided to construct relation objects from external data.

- `from_arrow`
- `from_df`
- `read_csv`
- `read_json`
- `read_parquet`

SQL Queries

Relation objects can be queried through SQL through so-called **replacement scans**. If you have a relation object stored in a variable, you can refer to that variable as if it was a SQL table (in the FROM clause). This allows you to incrementally build queries using relation objects.

```
import duckdb
rel = duckdb.sql('SELECT * FROM range(1000000) tbl(id)');
duckdb.sql('SELECT SUM(id) FROM rel').show()
```

```
┌──────────────┐
│   sum(id)    │
│    int128    │
├──────────────┤
│ 499999500000 │
└──────────────┘
```

Operations

There are a number of operations that can be performed on relations. These are all short-hand for running the SQL queries - and will return relations again themselves.

`aggregate(expr, groups = {})` Apply an (optionally grouped) aggregate over the relation. The system will automatically group by any columns that are not aggregates.

```
import duckdb
rel = duckdb.sql('SELECT * FROM range(1000000) tbl(id)');
rel.aggregate('id % 2 AS g, sum(id), min(id), max(id)')
```

```
┌─────────┬──────────────┬─────────┬─────────┐
│    g    │    sum(id)   │ min(id) │ max(id) │
│  int64  │    int128    │  int64  │  int64  │
├─────────┼──────────────┼─────────┼─────────┤
│       0 │ 249999500000 │       0 │  999998 │
│       1 │ 250000000000 │       1 │  999999 │
└─────────┴──────────────┴─────────┴─────────┘
```

`except_(rel)` Select all rows in the first relation, that do not occur in the second relation. The relations must have the same number of columns.

```
import duckdb
r1 = duckdb.sql('SELECT * FROM range(10) tbl(id)');
r2 = duckdb.sql('SELECT * FROM range(5) tbl(id)');
r1.except_(r2).show()
```

```
┌─────────┐
│   id    │
│  int64  │
├─────────┤
│       5 │
│       6 │
│       7 │
│       8 │
│       9 │
└─────────┘
```

`filter(condition)` Apply the given condition to the relation, filtering any rows that do not satisfy the condition.

```
import duckdb
rel = duckdb.sql('SELECT * FROM range(1000000) tbl(id)');
rel.filter('id > 5').limit(3).show()
```

```
┌─────────┐
│   id    │
```

```
┌─────────┐
│  int64  │
├─────────┤
│       6 │
│       7 │
│       8 │
└─────────┘
```

`intersect(rel)` Select the intersection of two relations - returning all rows that occur in both relations. The relations must have the same number of columns.

```
import duckdb
r1 = duckdb.sql('SELECT * FROM range(10) tbl(id)');
r2 = duckdb.sql('SELECT * FROM range(5) tbl(id)');
r1.intersect(r2).show()
```

```
┌─────────┐
│   id    │
│  int64  │
├─────────┤
│       0 │
│       1 │
│       2 │
│       3 │
│       4 │
└─────────┘
```

`join(rel, condition, type = 'inner')` Combine two relations, joining them based on the provided condition.

```
import duckdb
r1 = duckdb.sql('SELECT * FROM range(5) tbl(id)').set_alias('r1');
r2 = duckdb.sql('SELECT * FROM range(10, 15) tbl(id)').set_alias('r2');
r1.join(r2, 'r1.id + 10 = r2.id').show()
```

```
┌─────────┬─────────┐
│   id    │   id    │
│  int64  │  int64  │
├─────────┼─────────┤
│       0 │      10 │
│       1 │      11 │
│       2 │      12 │
│       3 │      13 │
└─────────┴─────────┘
```

```
┌─────────┬─────────┐
│    4    │   14    │
└─────────┴─────────┘
```

`limit(n, offset = 0)` Select the first *n* rows, optionally offset by *offset*.

```
import duckdb
rel = duckdb.sql('SELECT * FROM range(1000000) tbl(id)');
rel.limit(3).show()
```

```
┌─────────┐
│   id    │
│  int64  │
├─────────┤
│       0 │
│       1 │
│       2 │
└─────────┘
```

`order(expr)` Sort the relation by the given set of expressions.

```
import duckdb
rel = duckdb.sql('SELECT * FROM range(1000000) tbl(id)');
rel.order('id DESC').limit(3).show()
```

```
┌─────────┐
│   id    │
│  int64  │
├─────────┤
│  999999 │
│  999998 │
│  999997 │
└─────────┘
```

`project(expr)` Apply the given expression to each row in the relation.

```
import duckdb
rel = duckdb.sql('SELECT * FROM range(1000000) tbl(id)');
rel.project('id + 10 AS id_plus_ten').limit(3).show()
```

```
┌─────────────┐
│ id_plus_ten │
│    int64    │
├─────────────┤
```

```
│          10 │
│          11 │
│          12 │
└─────────────┘
```

`union(rel)` Combine two relations, returning all rows in `r1` followed by all rows in `r2`. The relations must have the same number of columns.

```
import duckdb
r1 = duckdb.sql('SELECT * FROM range(5) tbl(id)');
r2 = duckdb.sql('SELECT * FROM range(10, 15) tbl(id)');
r1.union(r2).show()
```

```
┌───────┐
│  id   │
│ int64 │
├───────┤
│     0 │
│     1 │
│     2 │
│     3 │
│     4 │
│    10 │
│    11 │
│    12 │
│    13 │
│    14 │
└───────┘
```

Result Output

The result of relations can be converted to various types of Python structures, see the result conversion page for more information.

The result of relations can also be directly written to files using the below methods.

- `write_csv`
- `write_parquet`

Python Function API

You can create a DuckDB user-defined function (UDF) out of a Python function so it can be used in SQL queries. Similarly to regular functions, they need to have a name, a return type and parameter types.

Here is an example using a Python function that calls a third-party library.

```
import duckdb
from duckdb.typing import *
from faker import Faker

def random_name():
    fake = Faker()
    return fake.name()

duckdb.create_function('random_name', random_name, [], VARCHAR)
res = duckdb.sql('select random_name()').fetchall()
print(res)
# [('Gerald Ashley',)]
```

Creating Functions

To register a Python UDF, simply use the `create_function` method from a DuckDB connection. Here is the syntax:

```
import duckdb
con = duckdb.connect()
con.create_function(name, function, argument_type_list, return_type, type,
 ↪  null_handling)
```

The `create_function` method requires the following parameters:

1. **name**: A string representing the unique name of the UDF within the connection catalog.
2. **function**: The Python function you wish to register as a UDF.
3. **return_type**: Scalar functions return one element per row. This parameter specifies the return type of the function.
4. **parameters**: Scalar functions can operate on one or more columns. This parameter takes a list of column types used as input.
5. **type** (Optional): DuckDB supports both built-in Python types and PyArrow Tables. By default, built-in types are assumed, but you can specify `type='arrow'` to use PyArrow Tables.
6. **null_handling** (Optional): By default, null values are automatically handled as Null-In Null-Out. Users can specify a desired behavior for null values by setting `null_handling='special'`.

7. **exception_handling** (Optional): By default, when an exception is thrown from the Python function, it will be re-thrown in Python. Users can disable this behavior, and instead return `null`, by set this parameter to `'return_null'`
8. **side_effects** (Optional): By default, functions are expected to produce the same result for the same input. If the result of a function is impacted by any type of randomness, `side_effects` must be set to `True`.

To unregister a UDF, you can call the `remove_function` method with the UDF name:

```
con.remove_function(name)
```

Type Annotation

When the function has type annotation it's often possible to leave out all of the optional parameters. Using DuckDBPyType we can implicitly convert many known types to DuckDBs type system.
For example:

```
import duckdb

def my_function(x: int) -> str:
    return x

duckdb.create_function('my_func', my_function)
duckdb.sql('select my_func(42)')
#
# ┌──────────────┐
# │ my_func(42)  │
# │   varchar    │
# ├──────────────┤
# │ 42           │
# └──────────────┘
```

If only the parameter list types can be inferred, you'll need to pass in None as `argument_type_list`.

Null Handling

By default when functions receive a NULL value, this instantly returns NULL, as part of the default null handling.
When this is not desired, you need to explicitly set this parameter to `'special'`.

```
import duckdb
from duckdb.typing import *
```

```
def dont_intercept_null(x):
    return 5

duckdb.create_function('dont_intercept', dont_intercept_null, [BIGINT],
 ↪  BIGINT)
res = duckdb.sql("""
    select dont_intercept(NULL)
""").fetchall()
print(res)
# [(None,)]

duckdb.remove_function('dont_intercept')
duckdb.create_function('dont_intercept', dont_intercept_null, [BIGINT],
 ↪  BIGINT, null_handling='special')
res = duckdb.sql("""
    select dont_intercept(NULL)
""").fetchall()
print(res)
# [(5,)]
```

Exception Handling

By default, when an exception is thrown from the Python function, we'll forward (re-throw) the exception. If you want to disable this behavior, and instead return null, you'll need to set this parameter to `'return_null'`

```
import duckdb
from duckdb.typing import *

def will_throw():
    raise ValueError("ERROR")

duckdb.create_function('throws', will_throw, [], BIGINT)
try:
    res = duckdb.sql("""
        select throws()
    """).fetchall()
except duckdb.InvalidInputException as e:
    print(e)
```

```
duckdb.create_function('doesnt_throw', will_throw, [], BIGINT, exception_
 ↪  handling='return_null')
res = duckdb.sql("""
    select doesnt_throw()
""").fetchall()
print(res)
# [(None,)]
```

Side Effects

By default DuckDB will assume the created function is a *pure* function, meaning it will produce the same output when given the same input. If your function does not follow that rule, for example when your function makes use of randomness, then you will need to mark this function as having `side_effects`.

For example, this function will produce a new count for every invocation

```
def count() -> int:
    old = count.counter;
    count.counter += 1
    return old
count.counter = 0
```

If we create this function without marking it as having side effects, the result will be the following:

```
con = duckdb.connect()
con.create_function('my_counter', count, side_effects=False)
res = con.sql('select my_counter() from range(10)').fetchall()
# [(0,), (0,), (0,), (0,), (0,), (0,), (0,), (0,), (0,), (0,)]
```

Which is obviously not the desired result, when we add `side_effects=True`, the result is as we would expect:

```
con.create_function('my_counter', count, side_effects=True)
res = con.sql('select my_counter() from range(10)').fetchall()
# [(0,), (1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,), (9,)]
```

Python Function Types

Currently two function types are supported, `native` (default) and `arrow`.

Arrow If the function is expected to receive arrow arrays, set the `type` parameter to `'arrow'`.

This will let the system know to provide arrow arrays of up to `STANDARD_VECTOR_SIZE` tuples to the function, and also expect an array of the same amount of tuples to be returned from the function.

Native When the function type is set to `native` the function will be provided with a single tuple at a time, and expect only a single value to be returned.
This can be useful to interact with Python libraries that don't operate on Arrow, such as `faker`:

```
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, type='native')
res = duckdb.sql('select random_date()').fetchall()
print(res)
# [(datetime.date(2019, 5, 15),)]
```

Types API

The `DuckDBPyType` class represents a type instance of our data types.

Converting from Other Types

To make the API as easy to use as possible, we have added implicit conversions from existing type objects to a DuckDBPyType instance. This means that wherever a DuckDBPyType object is expected, it is also possible to provide any of the options listed below.

Python Builtins The table below shows the mapping of Python Builtin type to DuckDB type.

Type	DuckDB Type
str	VARCHAR

Type	DuckDB Type
int	BIGINT
bytearray	BLOB
bytes	BLOB
float	DOUBLE
bool	BOOLEAN

Numpy DTypes The table below shows the mapping of Numpy DType to DuckDB type.

Type	DuckDB Type
bool	BOOLEAN
int8	TINYINT
int16	SMALLINT
int32	INTEGER
int64	BIGINT
uint8	UTINYINT
uint16	USMALLINT
uint32	UINTEGER
uint64	UBIGINT
float32	FLOAT
float64	DOUBLE

Nested Types

list[child_type] list type objects map to a `LIST` type of the child type. Which can also be arbitrarily nested.

```
import duckdb
from typing import Union
```

```
duckdb.typing.DuckDBPyType(list[dict[Union[str, int], str])
# MAP(UNION(u1 VARCHAR, u2 BIGINT), VARCHAR)[]
```

dict[key_type, value_type] dict type objects map to a MAP type of the key type and the value type.

```
import duckdb

duckdb.typing.DuckDBPyType(dict[str, int])
# MAP(VARCHAR, BIGINT)
```

{'a': field_one, 'b': field_two, .., 'n': field_n} dict objects map to a STRUCT composed of the keys and values of the dict.

```
import duckdb

duckdb.typing.DuckDBPyType({'a': str, 'b': int})
# STRUCT(a VARCHAR, b BIGINT)
```

Union[<type_one>, ... <type_n>] typing.Union objects map to a UNION type of the provided types.

```
import duckdb
from typing import Union

duckdb.typing.DuckDBPyType(Union[int, str, bool, bytearray])
# UNION(u1 BIGINT, u2 VARCHAR, u3 BOOLEAN, u4 BLOB)
```

Creation Functions For the builtin types, you can use the constants defined in `duckdb.typing`

DuckDB Type
SQLNULL
BOOLEAN
TINYINT
UTINYINT
SMALLINT
USMALLINT

DuckDB Type
INTEGER
UINTEGER
BIGINT
UBIGINT
HUGEINT
UUID
FLOAT
DOUBLE
DATE
TIMESTAMP
TIMESTAMP_MS
TIMESTAMP_NS
TIMESTAMP_S
TIME
TIME_TZ
TIMESTAMP_TZ
VARCHAR
BLOB
BIT
INTERVAL

For the complex types there are methods available on the `DuckDBPyConnection` object or the `duckdb` module.
Anywhere a `DuckDBPyType` is accepted, we will also accept one of the type objects that can implicitly convert to a `DuckDBPyType`.

`list_type|array_type` Parameters:

- `child_type: DuckDBPyType`

`struct_type|row_type` Parameters:

- `fields: Union[list[DuckDBPyType], dict[str, DuckDBPyType]]`

`map_type` Parameters:

- `key_type: DuckDBPyType`
- `value_type: DuckDBPyType`

`decimal_type` Parameters:

- `width: int`
- `scale: int`

`union_type` Parameters:

- `members: Union[list[DuckDBPyType], dict[str, DuckDBPyType]]`

`string_type` Parameters:

- `collation: Optional[str]`

Expression API

The `Expression` class represents an instance of an expression.

Why would I use the API?

Using this API makes it possible to dynamically build up expressions, which are typically created by the parser from the query string.
This allows you to skip that and have more fine-grained control over the used expressions.

Below is a list of currently supported expressions that can be created through the API.

Column Expression

This expression references a column by name.

```
import duckdb
import pandas as pd

df = pd.DataFrame({'a': [1,2,3,4]})

col = duckdb.ColumnExpression('a')
res = duckdb.df(df).select(col).fetchall()
print(res)
# [(1,), (2,), (3,), (4,)]
```

Star Expression

This expression selects all columns of the input source.

Optionally it's possible to provide an `exclude` list to filter out columns of the table.
This `exclude` list can contain either strings or Expressions.

```
import duckdb
import pandas as pd

df = pd.DataFrame({
    'a': [1,2,3,4],
    'b': [True, None, False,True],
    'c': [42, 21, 13, 14]
})

star = duckdb.StarExpression(exclude=['b'])
res = duckdb.df(df).select(star).fetchall()
print(res)
# [(1, 42), (2, 21), (3, 13), (4, 14)]
```

Constant Expression

This expression contains a single value.

```
import duckdb
import pandas as pd

df = pd.DataFrame({
    'a': [1,2,3,4],
    'b': [True, None, False,True],
    'c': [42, 21, 13, 14]
```

```
})

const = duckdb.ConstantExpression('hello')
res = duckdb.df(df).select(const).fetchall()
print(res)
# [('hello',), ('hello',), ('hello',), ('hello',)]
```

Case Expression

This expression contains a CASE WHEN (...) THEN (...) ELSE (...) END expression.
By default ELSE is NULL, it can be set using `.else(value=...)`
Additional `WHEN (...) THEN (...)` blocks can be added with `.when(condition=..., value=...)`

```
import duckdb
import pandas as pd
from duckdb import (
    ConstantExpression,
    ColumnExpression,
    CaseExpression
)

df = pd.DataFrame({
    'a': [1,2,3,4],
    'b': [True, None, False,True],
    'c': [42, 21, 13, 14]
})

hello = ConstantExpression('hello')
world = ConstantExpression('world')

case = CaseExpression(condition=ColumnExpression('b') == False,
 ↪  value=world).otherwise(hello)
res = duckdb.df(df).select(
    case
).fetchall()
print(res)
# [('hello',), ('hello',), ('world',), ('hello',)]
```

Function Expression

This expression contains a function call.
It can be constructed by providing the function name and an arbitrary amount of Expressions as arguments.

```
import duckdb
import pandas as pd
from duckdb import (
    ConstantExpression,
    ColumnExpression,
    FunctionExpression
)

df = pd.DataFrame({
    'a': [
        'test',
        'pest',
        'text',
        'rest',
    ]
})

ends_with = FunctionExpression('ends_with', ColumnExpression('a'),
 ↪  ConstantExpression('est'))
res = duckdb.df(df).select(
    ends_with
).fetchall()
print(res)
# [(True,), (True,), (False,), (True,)]
```

Common Operations

The Expression class also contains many operations that can be applied to any Expression type.

`.cast(type: DuckDBPyType)`
Applies a cast to the provided type on the expression.

`.alias(name: str)`
Apply an alias to the expression.

`.isin(*exprs: Expression)`
Create a IN expression against the provided expressions as the list.

`.isnotin(*exprs: Expression)`
Create a NOT IN expression against the provided expressions as the list.

Order Operations When expressions are provided to `DuckDBPyRelation.order()` these take effect:

`.asc()`
Indicates that this expression should be sorted in ascending order.

`.desc()`
Indicates that this expression should be sorted in descending order.

`.nulls_first()`
Indicates that the nulls in this expression should preceed the non-null values.

`.nulls_last()`
Indicates that the nulls in this expression should come after the non-null values.

Spark API

The DuckDB Spark API implements the PySpark API, allowing you to use the familiar Spark API to interact with DuckDB. All statements are translated to DuckDB's internal plans using our relational API and executed using DuckDB's query engine.

> **Note.** The DuckDB Spark 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.

Example

```
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')
]
```

Python Client API

Known Python Issues

Unfortunately there are some issues that are either beyond our control or are very elusive / hard to track down.
Below is a list of these issues that you might have to be aware of, depending on your workflow.

Numpy Import Multithreading

When making use of multi threading and fetching results either directly as Numpy arrays or indirectly through a Pandas DataFrame, it might be necessary to ensure that `numpy.core.multiarray` is imported.
If this module has not been imported from the main thread, and a different thread during execution attempts to import it this causes either a deadlock or a crash.

To avoid this, it's recommended to `import numpy.core.multiarray` before starting up threads.

Running EXPLAIN renders newlines in Jupyter and IPython

When DuckDB is run in Jupyter notebooks or in the IPython shell, the output of the `EXPLAIN` statement contains hard line breaks (`\n`):

```
In [1]: import duckdb
   ...: duckdb.sql("EXPLAIN SELECT 42 AS x")
```

```
Out[1]:
┌─────────────┬──────────────
```

Startup & Shutdown To use DuckDB, you must first create a connection object that represents the database. The connection object takes as parameter the database file to read and write from. If the database file does not exist, it will be created (the file extension may be `.db`, `.duckdb`, or anything else). The special value `:memory:` (the default) can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the R process). If you would like to connect to an existing database in read-only mode, set the `read_only` flag to TRUE. Read-only mode is required if multiple R processes want to access the same database file at the same time.

```
library("duckdb")
# to start an in-memory database
con <- dbConnect(duckdb())
# or
con <- dbConnect(duckdb(), dbdir = ":memory:")
# to use a database file (not shared between processes)
con <- dbConnect(duckdb(), dbdir = "my-db.duckdb", read_only = FALSE)
# to use a database file (shared between processes)
con <- dbConnect(duckdb(), dbdir = "my-db.duckdb", read_only = TRUE)
```

Connections are closed implicitly when they go out of scope or if they are explicitly closed using `dbDisconnect()`. To shut down the database instance associated with the connection, use `dbDisconnect(con, shutdown=TRUE)`

Querying DuckDB supports the standard DBI methods to send queries and retrieve result sets. `dbExecute()` is meant for queries where no results are expected like `CREATE TABLE` or `UPDATE` etc. and `dbGetQuery()` is meant to be used for queries that produce results (e.g., `SELECT`). Below an example.

```
# create a table
dbExecute(con, "CREATE TABLE items(item VARCHAR, value DECIMAL(10, 2), count
 ↪  INTEGER)")
# insert two items into the table
dbExecute(con, "INSERT INTO items VALUES ('jeans', 20.0, 1), ('hammer',
 ↪  42.2, 2)")

# retrieve the items again
res <- dbGetQuery(con, "SELECT * FROM items")
print(res)
#     item value count
# 1  jeans  20.0     1
# 2 hammer  42.2     2
```

DuckDB also supports prepared statements in the R API with the `dbExecute` and `dbGetQuery` methods. Here is an example:

```
# prepared statement parameters are given as a list
dbExecute(con, "INSERT INTO items VALUES (?, ?, ?)", list('laptop', 2000, 1))

# if you want to reuse a prepared statement multiple times, use
 ↪  dbSendStatement() and dbBind()
stmt <- dbSendStatement(con, "INSERT INTO items VALUES (?, ?, ?)")
dbBind(stmt, list('iphone', 300, 2))
dbBind(stmt, list('android', 3.5, 1))
dbClearResult(stmt)

# query the database using a prepared statement
res <- dbGetQuery(con, "SELECT item FROM items WHERE value > ?", list(400))
print(res)
#       item
# 1 laptop
```

> **Note.** Do **not** use prepared statements to insert large amounts of data into DuckDB. See below for better options.

Efficient Transfer

To write a R data frame into DuckDB, use the standard DBI function `dbWriteTable()`. This creates a table in DuckDB and populates it with the data frame contents. For example:

```
dbWriteTable(con, "iris_table", iris)
res <- dbGetQuery(con, "SELECT * FROM iris_table LIMIT 1")
print(res)
#   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
# 1          5.1         3.5          1.4         0.2  setosa
```

It is also possible to ”register” a R data frame as a virtual table, comparable to a SQL `VIEW`. This *does not actually transfer data* into DuckDB yet. Below is an example:

```
duckdb_register(con, "iris_view", iris)
res <- dbGetQuery(con, "SELECT * FROM iris_view LIMIT 1")
print(res)
#   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
# 1          5.1         3.5          1.4         0.2  setosa
```

> **Note.** DuckDB keeps a reference to the R data frame after registration. This prevents the data frame from being garbage-collected. The reference is cleared when the connection is closed, but can also be cleared manually using the `duckdb_unregister()` method.

Also refer to the data import documentation for more options of efficiently importing data.

dbplyr

DuckDB also plays well with the dbplyr / dplyr packages for programmatic query construction from R. Here is an example:

```
library("duckdb")
library("dplyr")
con <- dbConnect(duckdb())
duckdb_register(con, "flights", nycflights13::flights)

tbl(con, "flights") |>
  group_by(dest) |>
  summarise(delay = mean(dep_time, na.rm = TRUE)) |>
  collect()
```

When using dbplyr, CSV and Parquet files can be read using the `dplyr::tbl` function.

```
# Establish a csv for the sake of this example
write.csv(mtcars, "mtcars.csv")

# Summarize the dataset in DuckDB to avoid reading the entire csv into R's
 ↪  memory
tbl(con, "mtcars.csv") |>
  group_by(cyl) |>
  summarise(across(disp:wt, .fns = mean)) |>
  collect()

# Establish a set of parquet files
dbExecute(con, "COPY flights TO 'dataset' (FORMAT PARQUET, PARTITION_BY
 ↪  (year, month))")

# Summarize the dataset in DuckDB to avoid reading 12 parquet files into R's
 ↪  memory
tbl(con, "read_parquet('dataset/**/*.parquet', hive_partitioning=1)") |>
  filter(month == "3") |>
  summarise(delay = mean(dep_time, na.rm = TRUE)) |>
  collect()
```

GitHub Repository

GitHub

Rust API

Installation

The DuckDB Rust API can be installed from crates.io. Please see the docs.rs for details.

Basic API Usage

duckdb-rs is an ergonomic wrapper based on the DuckDB C API, please refer to the README for details.

Startup & Shutdown To use duckdb, you must first initialize a `Connection` handle using `Connection::open()`. `Connection::open()` takes as parameter the database file to read and write from. If the database file does not exist, it will be created (the file extension may be `.db`, `.duckdb`, or anything else). You can also use `Connection::open_in_memory()` to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the process).

```
use duckdb::{params, Connection, Result};
let conn = Connection::open_in_memory()?;
```

You can `conn.close()` the `Connection` manually, or just leave it out of scope, we had implement the `Drop` trait which will automatically close the underlining db connection for you.

Querying SQL queries can be sent to DuckDB using the `execute()` method of connections, or we can also prepare the statement and then query on that.

```
conn.execute(
    "INSERT INTO person (name, data) VALUES (?, ?)",
    params![me.name, me.data],
)?;

let mut stmt = conn.prepare("SELECT id, name, data FROM person")?;
let person_iter = stmt.query_map([], |row| {
    Ok(Person {
```

```
        id: row.get(0)?,
        name: row.get(1)?,
        data: row.get(2)?,
    })
})?;

for person in person_iter {
    println!("Found person {:?}", person.unwrap());
}
```

Scala JDBC API

Installation

The DuckDB Java JDBC API can be used in Scala and can be installed from Maven Central. Please see the installation page for details.

Basic API Usage

Scala uses DuckDB's JDBC API implements the main parts of the standard Java Database Connectivity (JDBC) API, version 4.0. Describing JDBC is beyond the scope of this page, see the official documentation for details. Below we focus on the DuckDB-specific parts.

Startup & Shutdown In Scala, database connections are created through the standard `java.sql.DriverManager` class. The driver should auto-register in the DriverManager, if that does not work for some reason, you can enforce registration like so:

```
Class.forName("org.duckdb.DuckDBDriver");
```

To create a DuckDB connection, call `DriverManager` with the `jdbc:duckdb:` JDBC URL prefix, like so:

```
val conn = DriverManager.getConnection("jdbc:duckdb:");
```

When using the `jdbc:duckdb:` URL alone, an **in-memory database** is created. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the Java program). If you would like to access or create a persistent database, append its file name after the path. For example, if your database is stored in `/tmp/my_database`, use the JDBC URL `jdbc:duckdb:/tmp/my_database` to create a connection to it.

It is possible to open a DuckDB database file in **read-only** mode. This is for example useful if multiple Java processes want to read the same database file at the same time. To open an existing database file in read-only mode, set the connection property `duckdb.read_only` like so:

```
val ro_prop = new Properties();
ro_prop.setProperty("duckdb.read_only", "true");
val conn_ro = DriverManager.getConnection("jdbc:duckdb:/tmp/my_database",
 ↪  ro_prop);
```

The JDBC `DriverManager` API is a relatively poor fit for embedded database management systems such as DuckDB. If you would like to create **multiple connections to the same database**, it would be somewhat logical to just create additional connections with the same URL. This is however only supported for read-only connections. If you would like to create multiple read-write connections to the same database file or the same in-memory database instance, you can use the custom `duplicate()` method like so:

```
val conn2 = ((DuckDBConnection) conn).duplicate();
```

Querying DuckDB supports the standard JDBC methods to send queries and retrieve result sets. First a `Statement` object has to be created from the `Connection`, this object can then be used to send queries using `execute` and `executeQuery`. `execute()` is meant for queries where no results are expected like `CREATE TABLE` or `UPDATE` etc. and `executeQuery()` is meant to be used for queries that produce results (e.g., `SELECT`). Below two examples. See also the JDBC `Statement` and `ResultSet` documentations.

```
// create a table
val stmt = conn.createStatement();
stmt.execute("CREATE TABLE items (item VARCHAR, value DECIMAL(10, 2), count
 ↪  INTEGER)");
// insert two items into the table
stmt.execute("INSERT INTO items VALUES ('jeans', 20.0, 1), ('hammer', 42.2,
 ↪  2)");

val rs = stmt.executeQuery("SELECT * FROM items");
while (rs.next()) {
    System.out.println(rs.getString(1));
    System.out.println(rs.getInt(3));
}
rs.close()
// jeans
// 1
// hammer
// 2
```

DuckDB also supports prepared statements as per the JDBC API:

```
val p_stmt = conn.prepareStatement("INSERT INTO test VALUES (?, ?, ?);");

p_stmt.setString(1, "chainsaw");
p_stmt.setDouble(2, 500.0);
p_stmt.setInt(3, 42);
p_stmt.execute();

// more calls to execute() possible
p_stmt.close();
```

> **Note.** Do *not* use prepared statements to insert large amounts of data into DuckDB. See the data import documentation for better options.

Swift API

TL;DR: See the announcement post for more information!

GitHub Repository

GitHub

Wasm

DuckDB Wasm

DuckDB has been compiled to WebAssembly, so it can run inside any browser on any device.

{% include iframe.html src="https://shell.duckdb.org" %}

DuckDB-Wasm offers a layered API, it can be embedded as a JavaScript + WebAssembly library, as a Web shell, or built from source according to your needs.

Getting Started with DuckDB-Wasm

A great starting point is to read the DuckDB-Wasm launch blog post!

Another great resource is the GitHub repository.

For details, see the full DuckDB-Wasm API Documentation.

Instantiation

DuckDB-Wasm has multiple ways to be instantiated depending on the use case.

Instantiation

cdn(jsdelivr)

```
import * as duckdb from '@duckdb/duckdb-wasm';

const JSDELIVR_BUNDLES = duckdb.getJsDelivrBundles();

// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(JSDELIVR_BUNDLES);

const worker_url = URL.createObjectURL(
  new Blob([`importScripts("${bundle.mainWorker!}");`], {type:
  ↪ 'text/javascript'})
);

// Instantiate the asynchronus version of DuckDB-Wasm
const worker = new Worker(worker_url);
const logger = new duckdb.ConsoleLogger();
const db = new duckdb.AsyncDuckDB(logger, worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
URL.revokeObjectURL(worker_url);
```

webpack

```
import * as duckdb from '@duckdb/duckdb-wasm';
import duckdb_wasm from '@duckdb/duckdb-wasm/dist/duckdb-mvp.wasm';
import duckdb_wasm_next from '@duckdb/duckdb-wasm/dist/duckdb-eh.wasm';
const MANUAL_BUNDLES: duckdb.DuckDBBundles = {
    mvp: {
        mainModule: duckdb_wasm,
        mainWorker: new
↪   URL('@duckdb/duckdb-wasm/dist/duckdb-browser-mvp.worker.js',
↪   import.meta.url).toString(),
    },
    eh: {
        mainModule: duckdb_wasm_next,
```

```
        mainWorker: new
 ↪  URL('@duckdb/duckdb-wasm/dist/duckdb-browser-eh.worker.js',
 ↪  import.meta.url).toString(),
    },
};
// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(MANUAL_BUNDLES);
// Instantiate the asynchronus 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);
```

vite

```
import * as duckdb from '@duckdb/duckdb-wasm';
import duckdb_wasm from '@duckdb/duckdb-wasm/dist/duckdb-mvp.wasm?url';
import mvp_worker from
 ↪  '@duckdb/duckdb-wasm/dist/duckdb-browser-mvp.worker.js?url';
import duckdb_wasm_eh from '@duckdb/duckdb-wasm/dist/duckdb-eh.wasm?url';
import eh_worker from
 ↪  '@duckdb/duckdb-wasm/dist/duckdb-browser-eh.worker.js?url';

const MANUAL_BUNDLES: duckdb.DuckDBBundles = {
    mvp: {
        mainModule: duckdb_wasm,
        mainWorker: mvp_worker,
    },
    eh: {
        mainModule: duckdb_wasm_eh,
        mainWorker: eh_worker,
    },
};
// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(MANUAL_BUNDLES);
// Instantiate the asynchronus 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);
```

static served (manually download the files from https://cdn.jsdelivr.net/npm/@duckdb/duckdb-wasm/dist/)

```
import * as duckdb from '@duckdb/duckdb-wasm';

const MANUAL_BUNDLES: duckdb.DuckDBBundles = {
    mvp: {
        mainModule: 'change/me/../duckdb-mvp.wasm',
        mainWorker: 'change/me/../duckdb-browser-mvp.worker.js',
    },
    eh: {
        mainModule: 'change/m/../duckdb-eh.wasm',
        mainWorker: 'change/m/../duckdb-browser-eh.worker.js',
    },
};
// 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);
```

Data Ingestion

DuckDB-Wasm has multiple ways to import data, depending on the format of the data.

There are two steps to import data into DuckDB.

First, the data file is imported into a local file system using register functions (registerEmptyFileBuffer, registerFileBuffer, registerFileHandle, registerFileText, registerFileURL).

Then, the data file is imported into DuckDB using insert functions (insertArrowFromIPCStream, insertArrowTable, insertCSVFromPath, insertJSONFromPath) or directly using FROM SQL query (using extensions like parquet or wasm flavoured httpfs).

Insert statements can also be used to import data.

Data Import

Open & Close connection

```
// Create a new connection
const c = await db.connect();

// ... import data
```

```
// Close the connection to release memory
await c.close();
```

Apache Arrow

```
// Data can be inserted from an existing arrow.Table
// More Example https://arrow.apache.org/docs/js/
import { tableFromArrays } from 'apache-arrow';

const arrowTable = tableFromArrays({
  id: [1, 2, 3],
  name: ['John', 'Jane', 'Jack'],
  age: [20, 21, 22],
});
await c.insertArrowTable(arrowTable, { name: 'arrow_table' });

// ..., from a raw Arrow IPC stream
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);
```

CSV

```
// ..., from CSV files
// (interchangeable: registerFile{Text,Buffer,URL,Handle})
const csvContent = '1|foo\n2|bar\n';
await db.registerFileText(` data.csv`, csvContent);
// ... with typed insert options
await db.insertCSVFromPath('data.csv', {
    schema: 'main',
    name: 'foo',
    detect: false,
    header: false,
    delimiter: '|',
    columns: {
```

```
        col1: new arrow.Int32(),
        col2: new arrow.Utf8(),
    },
});
```

JSON

```
// ..., from JSON documents in row-major format
const jsonRowContent = [
    { "col1": 1, "col2": "foo" },
    { "col1": 2, "col2": "bar" },
];
await db.registerFileText(
    'rows.json',
    JSON.stringify(jsonRowContent),
);
await c.insertJSONFromPath('rows.json', { name: 'rows' });

// ... or column-major format
const jsonColContent = {
    "col1": [1, 2],
    "col2": ["foo", "bar"]
};
await db.registerFileText(
    'columns.json',
    JSON.stringify(jsonColContent),
);
await c.insertJSONFromPath('columns.json', { name: 'columns' });

// From API
const streamResponse = await fetch(` someapi/content.json`);
await db.registerFileBuffer('file.json', new Uint8Array(await
 ↪  streamResponse.arrayBuffer()))
await c.insertJSONFromPath('file.json', { name: 'JSONContent' });
```

Parquet

```
// from Parquet files
// ...Local
const pickedFile: File = letUserPickFile();
await db.registerFileHandle('local.parquet', pickedFile,
 ↪  DuckDBDataProtocol.BROWSER_FILEREADER, true);
// ...Remote
```

```
await db.registerFileURL('remote.parquet', 'https://origin/remote.parquet',
 ↪  DuckDBDataProtocol.HTTP, false);
// ... Using Fetch
const res = await fetch('https://origin/remote.parquet');
await db.registerFileBuffer('buffer.parquet', new Uint8Array(await
 ↪  res.arrayBuffer()));

// ..., 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")`);
```

httpfs (wasm flavoured)

```
// ..., by specifying URLs in the SQL text
await c.query(`
    CREATE TABLE direct AS
        SELECT * FROM "https://origin/remote.parquet"
`);
```

Insert statement

```
// ..., or by executing raw insert statements
await c.query(` INSERT INTO existing_table
    VALUES (1, "foo"), (2, "bar")`);
```

Query

DuckDB-Wasm provides functions for querying data. Queries are run sequentially.

First, a connection need to be created by calling connect. Then, queries can be run by calling query or send.

Query Execution

```
// Create a new connection
const conn = await db.connect();
```

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

// Close the connection to release memory
await conn.close();
```

Prepared Statements

```
// Create a new connection
const conn = await db.connect();
// Prepare query
const stmt = await conn.prepare(` SELECT v + ? FROM generate_series(0,
 ↪  10000) AS 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)) {
    // ...
}
// Close the statement to release memory
await stmt.close();
// Closing the connection will release statements as well
await conn.close();
```

Arrow Table to JSON

```
// Create a new connection
const conn = await db.connect();

// Query
const arrowResult = await conn.query<{ v: arrow.Int }>(`
    SELECT * FROM generate_series(1, 100) t(v)
```

```
`);

// Convert arrow table to json
const result = arrowResult.toArray().map((row) => row.toJSON());

// Close the connection to release memory
await conn.close();
```

Export Parquet

```
// Create a new connection
const conn = await db.connect();

// Export Parquet
conn.send(` COPY (SELECT * FROM tbl) TO 'result-snappy.parquet' (FORMAT
 ↪  'parquet');`);
const parquet_buffer = await this._
 ↪  db.copyFileToBuffer('result-snappy.parquet');

// Generate a download link
const link = URL.createObjectURL(new Blob([parquet_buffer]));

// Close the connection to release memory
await conn.close();
```

Extensions

DuckDB-Wasm's (dynamic) extension loading is modeled after the regular DuckDB's extension loading, with a few relevant differences due to the difference in platform.

Format

Extensions in DuckDB are binaries to be dynamically loaded via `dlopen`. A cryptographical signature is appended to the binary. An extension in DuckDB-Wasm is a regular Wasm file to be dynamically loaded via Emscripten's dlopen. A cryptographical signature is appended to the Wasm file as a WebAssembly custom section called `duckdb_signature`. This ensures the file remains a valid WebAssembly file.

> **Note.** Currently we require this custom section to be the last one, but this can be potentially relaxed in the future.

INSTALL and LOAD

The `INSTALL` semantic in native embeddings of DuckDB is to fetch, decompress from `gzip` and store data in local disk. The `LOAD` semantic in native embeddings of DuckDB is to (optionally) perform signature checks *and* dynamic load the binary with the main DuckDB binary.

In DuckDB-Wasm, `INSTALL` is a no-op given there is no durable cross-session storage. The `LOAD` operation will fetch (and decompress on the fly), perform signature checks *and* dynamically load via the Emscripten implementation of dlopen.

Autoloading

Autoloading, i.e., the possibility for DuckDB to add extension functionality on-the-fly, is enabled by default in DuckDB-Wasm.

List of Officially Available Extensions

Extension name	Description	Aliases
autocomplete	Adds support for autocomplete in the shell	
excel	Adds support for Excel-like format strings	
fts	Adds support for Full-Text Search Indexes	
icu	Adds support for time zones and collations using the ICU library	
inet	Adds support for IP-related data types and functions	
json	Adds support for JSON operations	
parquet	Adds support for reading and writing parquet files	
sqlite_scanner GitHub	Adds support for reading SQLite database files	sqlite, sqlite3
sqlsmith		

Extension name	Description	Aliases
substrait GitHub	Adds support for the Substrait integration	
tpcds	Adds TPC-DS data generation and query support	
tpch	Adds TPC-H data generation and query support	

WebAssembly is basically an additional platform, and there might be platform-specific limitations that make some extensions not able to match their native capabilities or to perform them in a different way. We will document here relevant differences for DuckDB-hosted extensions.

HTTPFS The HTTPFS extension is, at the moment, not available in DuckDB-Wasm. Https protocol capabilities needs to go through an additional layer, the browser, which adds both differences and some restrictions to what is doable from native.

Instead, DuckDB-Wasm has a separate implementation that for most purposes is interchangable, but does not support all use cases (as it must follow security rules imposed by the browser, such as CORS). Due to this CORS restriction, any requests for data made using the HTTPFS extension must be to websites that allow (using CORS headers) the website hosting the DuckDB-Wasm instance to access that data. The MDN website is a great resource for more information regarding CORS.

Extension Signing

As with regular DuckDB extensions, DuckDB-Wasm extension are by default checked on LOAD to verify the signature confirm the extension has not been tampered with. Extension signature verification can be disabled via a configuration option. Signing is a property of the binary itself, so copying a DuckDB extension (say to serve it from a different location) will still keep a valid signature (e.g., for local development).

Fetching DuckDB-Wasm Extensions

Official DuckDB extensions are served at `extensions.duckdb.org`, and this is also the default value for the `default_extension_repository` option. When installing extensions, a relevant URL will be built that will look like `extensions.duckdb.org/$duckdb_version_hash/$duckdb_platform/$name.duckdb_extension.gz`.

DuckDB-Wasm extension are fetched only on load, and the URL will look like: `extensions.duckdb.org/duckdb-wasm/$duckdb_version_hash/$duckdb_platform/$name.duckdb_extension.wasm`.

Note that an additional `duckdb-wasm` is added to the folder structure, and the file is served as a `.wasm` file.

DuckDB-Wasm extensions are served pre-compressed using Brotli compression. While fetched from a browser, extensions will be transparently uncompressed. If you want to fetch the `duckdb-wasm` extension manually, you can use `curl --compress extensions.duckdb.org/<...>/icu.duckdb_extension.wasm`.

Serving Extensions from a Third-Party Repository

As with regular DuckDB, if you use `SET custom_extension_repository = some.url.com`, subsequent loads will be attempted at `some.url.com/duckdb-wasm/$duckdb_version_hash/$duckdb_platform/$name.duckdb_extension.wasm`.

Note that GET requests on the extensions needs to be CORS enabled for a browser to allow the connection.

Tooling

Both DuckDB-Wasm and its extensions have been compiled using latest packaged Emscripten toolchain.

{% include iframe.html src="https://shell.duckdb.org" %}

ADBC API

Arrow Database Connectivity (ADBC), similarly to ODBC and JDBC, is a C-style API that enables code portability between different database systems. This allows developers to effortlessly build applications that communicate with database systems without using code specific to that system. The main difference between ADBC and ODBC/JDBC is that ADBC uses Arrow to transfer data between the database system and the application. DuckDB has an ADBC driver, which takes advantage of the zero-copy integration between DuckDB and Arrow to efficiently transfer data.

DuckDB's ADBC driver currently supports version 0.5.1 of ADBC.

Please refer to the ADBC documentation page for a more extensive discussion on ADBC and a detailed API explanation.

> **Note.** ADBC is not yet supported on Windows, but it is supported on macOS and Linux.

Implemented Functionality

The DuckDB-ADBC driver implements the full ADBC specification, with the exception of the `ConnectionReadPartition` and `StatementExecutePartitions` functions. Both of these functions exist to support systems that internally partition the query results, which does not apply to DuckDB. In this section, we will describe the main functions that exist in ADBC, along with the arguments they take and provide examples for each function.

Database Set of functions that operate on a database.

Function Name	Description	Arguments	Example
`DatabaseNew`	Allocate a new (but uninitialized) database.	`(AdbcDatabase *database, AdbcError *error)`	`AdbcDatabaseNew(&adbc_database, &adbc_error)`
`DatabaseSetOption`	Set a char* option.	`(AdbcDatabase *database, const char *key, const char *value, AdbcError *error)`	`AdbcDatabaseSetOption(&adbc_database, "path", "test.db", &adbc_error)`
`DatabaseInit`	Finish setting options and initialize the database.	`(AdbcDatabase *database, AdbcError *error)`	`AdbcDatabaseInit(&adbc_database, &adbc_error)`
`DatabaseRelease`	Destroy the database.	`(AdbcDatabase *database, AdbcError *error)`	`AdbcDatabaseRelease(&adbc_database, &adbc_error)`

Connection A set of functions that create and destroy a connection to interact with a database.

Function Name	Description	Arguments	Example
`ConnectionNew`	Allocate a new (but uninitialized) connection.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionNew(&adbc_connection, &adbc_error)`
`ConnectionSetOption`	Options may be set before ConnectionInit.	`(AdbcConnection*, const char*, const char*, AdbcError*)`	`AdbcConnectionSetOption(&adbc_connection, ADBC_CONNECTION_OPTION_AUTOCOMMIT, ADBC_OPTION_VALUE_DISABLED, &adbc_error)`
`ConnectionInit`	Finish setting options and initialize the connection.	`(AdbcConnection*, AdbcDatabase*, AdbcError*)`	`AdbcConnectionInit(&adbc_connection, &adbc_database, &adbc_error)`
`ConnectionRelease`	Destroy this connection.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionRelease(&adbc_connection, &adbc_error)`

A set of functions that retrieve metadata about the database. In general, these functions will return Arrow objects, specifically an ArrowArrayStream.

Function Name	Description	Arguments	Example
`ConnectionGetObjects`	Get a hierarchical view of all catalogs, database schemas, tables, and columns.	`(AdbcConnection*, int, const char*, const char*, const char*, const char**, const char*, ArrowArrayStream*, AdbcError*)`	`AdbcDatabaseInit(&adbc_database, &adbc_error)`

Function Name	Description	Arguments	Example
`ConnectionGetTableSchema`	Get the Arrow schema of a table.	`(AdbcConnection*, const char*, const char*, const char*, ArrowSchema*, AdbcError*)`	`AdbcDatabaseRelease(&adbc_database, &adbc_error)`
`ConnectionGetTableTypes`	Get a list of table types in the database.	`(AdbcConnection*, ArrowArrayStream*, AdbcError*)`	`AdbcDatabaseNew(&adbc_database, &adbc_error)`

A set of functions with transaction semantics for the connection. By default, all connections start with auto-commit mode on, but this can be turned off via the ConnectionSetOption function.

Function Name	Description	Arguments	Example
`ConnectionCommit`	Commit any pending transactions.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionCommit(&adbc_connection, &adbc_error)`
`ConnectionRollback`	Rollback any pending transactions.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionRollback(&adbc_connection, &adbc_error)`

Statement Statements hold state related to query execution. They represent both one-off queries and prepared statements. They can be reused; however, doing so will invalidate prior result sets from that statement.

The functions used to create, destroy, and set options for a statement:

Function Name	Description	Arguments	Example
`StatementNew`	Create a new statement for a given connection.	`(AdbcConnection*, AdbcStatement*, AdbcError*)`	`AdbcStatementNew(&adbc_connection, &adbc_statement, &adbc_error)`
`StatementRelease`	Destroy a statement.	`(AdbcStatement*, AdbcError*)`	`AdbcStatementRelease(&adbc_statement, &adbc_error)`
`StatementSetOption`	Set a string option on a statement.	`(AdbcStatement*, const char*, const char*, AdbcError*)`	`StatementSetOption(&adbc_statement, ADBC_INGEST_OPTION_TARGET_TABLE, "TABLE_NAME", &adbc_error)`

Functions related to query execution:

Function Name	Description	Arguments	Example
`StatementSetSqlQuery`	Set the SQL query to execute. The query can then be executed with StatementExecuteQuery.	`(AdbcStatement*, const char*, AdbcError*)`	`AdbcStatementSetSqlQuery(&adbc_statement, "SELECT * FROM TABLE", &adbc_error)`

Function Name	Description	Arguments	Example
`StatementSetSubstraitPlan`	Set a substrait plan to execute. The query can then be executed with StatementExecuteQuery.	`(AdbcStatement*, const uint8_t*, size_t, AdbcError*)`	`AdbcStatementSetSubstraitPlan(&a statement, substrait_ plan, length, &adbc_ error)`
`StatementExecuteQuery`	Execute a statement and get the results.	`(AdbcStatement*, ArrowAr- rayStream*, int64_t*, AdbcError*)`	`AdbcStatementExecuteQuery(&adbc_ statement, &arrow_ stream, &rows_ affected, &adbc_ error)`
`StatementPrepare`	Turn this statement into a prepared statement to be executed multiple times.	`(AdbcStatement*, AdbcError*)`	`AdbcStatementPrepare(&adbc_ statement, &adbc_ error)`

Functions related to binding, used for bulk insertion or in prepared statements.

Function Name	Description	Arguments	Example
`StatementBindStream`	Bind Arrow Stream. This can be used for bulk inserts or prepared statements.	`(AdbcStatement*, ArrowArrayStream*, AdbcError*)`	`StatementBindStream(&adbc_statement, &input_data, &adbc_error)`

Examples

Regardless of the programming language being used, there are two database options which will be required to utilize ADBC with DuckDB. The first one is the `driver`, which takes a path to the DuckDB library. The second option is the `entrypoint`, which is an exported function from the DuckDB-ADBC driver that initializes all the ADBC functions. Once we have configured these two options, we can optionally set the `path` option, providing a path on disk to store our DuckDB database. If not set, an in-memory database is created. After configuring all the necessary options, we can proceed to initialize our database. Below is how you can do so with various different language environments.

C++ We begin our C++ example by declaring the essential variables for querying data through ADBC. These variables include Error, Database, Connection, Statement handling, and an Arrow Stream to transfer data between DuckDB and the application.

```
AdbcError adbc_error;
AdbcDatabase adbc_database;
AdbcConnection adbc_connection;
AdbcStatement adbc_statement;
ArrowArrayStream arrow_stream;
```

We can then initialize our database variable. Before initializing the database, we need to set the `driver` and `entrypoint` options as mentioned above. Then we set the `path` option and initialize the database. With the example below, the string `"path/to/libduckdb.dylib"` should be the path to the dynamic library for DuckDB. This will be `.dylib` on macOS, and `.so` on Linux.

```
AdbcDatabaseNew(&adbc_database, &adbc_error);
AdbcDatabaseSetOption(&adbc_database, "driver", "path/to/libduckdb.dylib",
 ↪  &adbc_error);
```

```
AdbcDatabaseSetOption(&adbc_database, "entrypoint", "duckdb_adbc_init",
 ↪  &adbc_error);
// By default, we start an in-memory database, but you can optionally define
 ↪  a path to store it on disk.
AdbcDatabaseSetOption(&adbc_database, "path", "test.db", &adbc_error);
AdbcDatabaseInit(&adbc_database, &adbc_error);
```

After initializing the database, we must create and initialize a connection to it.

```
AdbcConnectionNew(&adbc_connection, &adbc_error);
AdbcConnectionInit(&adbc_connection, &adbc_database, &adbc_error);
```

We can now initialize our statement and run queries through our connection. After the `AdbcStatementExecuteQuery` the `arrow_stream` is populated with the result.

```
AdbcStatementNew(&adbc_connection, &adbc_statement, &adbc_error);
AdbcStatementSetSqlQuery(&adbc_statement, "SELECT 42", &adbc_error);
int64_t rows_affected;
AdbcStatementExecuteQuery(&adbc_statement, &arrow_stream, &rows_affected,
 ↪  &adbc_error);
arrow_stream.release(arrow_stream)
```

Besides running queries, we can also ingest data via `arrow_streams`. For this we need to set an option with the table name we want to insert to, bind the stream and then execute the query.

```
StatementSetOption(&adbc_statement, ADBC_INGEST_OPTION_TARGET_TABLE,
 ↪  "AnswerToEverything", &adbc_error);
StatementBindStream(&adbc_statement, &arrow_stream, &adbc_error);
StatementExecuteQuery(&adbc_statement, nullptr, nullptr, &adbc_error);
```

Python The first thing to do is to use `pip` and install the ADBC Driver manager. You will also need to install the `pyarrow` to directly access Apache Arrow formatted result sets (such as using `fetch_arrow_table`).

```
pip install adbc_driver_manager pyarrow
```

The full documentation for the `adbc_driver_manager` package can be found here.

As with C++, we need to provide initialization options consisting of the location of the libduckdb shared object and entrypoint function. Notice that the `path` argument for DuckDB is passed in through the `db_kwargs` dictionary.

```
import adbc_driver_duckdb.dbapi

with adbc_driver_duckdb.dbapi.connect("test.db"
```

```
  ) as conn, conn.cursor() as cur:
  cur.execute("SELECT 42")
  # fetch a pyarrow table
  tbl = cur.fetch_arrow_table()
  print(tbl)
```

Alongside `fetch_arrow_table`, other methods from DBApi are also implemented on the cursor, such as `fetchone` and `fetchall`. Data can also be ingested via `arrow_streams`. We just need to set options on the statement to bind the stream of data and execute the query.

```
import adbc_driver_duckdb.dbapi
import pyarrow

data = pyarrow.record_batch(
    [[1, 2, 3, 4], ["a", "b", "c", "d"]],
    names=["ints", "strs"],
)

with adbc_driver_duckdb.dbapi.connect("test.db"
  ) as conn, conn.cursor() as cur:
  cur.adbc_ingest("AnswerToEverything", data)
```

ODBC

ODBC API - Overview

The ODBC (Open Database Connectivity) is a C-style API that provides access to different flavors of Database Management Systems (DBMSs). The ODBC API consists of the Driver Manager (DM) and the ODBC drivers.

The DM is part of the system library, e.g., unixODBC, which manages the communications between the user applications and the ODBC drivers. Typically, applications are linked against the DM, which uses Data Source Name (DSN) to look up the correct ODBC driver.

The ODBC driver is a DBMS implementation of the ODBC API, which handles all the internals of that DBMS.

The DM maps user application calls of ODBC functions to the correct ODBC driver that performs the specified function and returns the proper values.

DuckDB ODBC Driver

DuckDB supports the ODBC version 3.0 according to the Core Interface Conformance.

We release the ODBC driver as assets for Linux and Windows. Users can download them from the Latest Release of DuckDB.

Operating System

ODBC API - Linux

A driver manager is required to manage communication between applications and the ODBC driver. We tested and support `unixODBC` that is a complete ODBC driver manager for Linux. Users can install it from the command line:

Debian SO Flavors

```
sudo apt get install unixodbc
```

Fedora SO Flavors

```
sudo yum install unixodbc
# or
sudo dnf install unixodbc
```

Step 1: Download ODBC Driver

DuckDB releases the ODBC driver as asset. For linux, download it from ODBC Linux Asset that contains the following artifacts:

libduckdb_odbc.so: the DuckDB driver compiled to Ubuntu 16.04.

unixodbc_setup.sh: a setup script to aid the configuration on Linux.

Step 2: Extracting ODBC Artifacts

Run unzip to extract the files to a permanent directory:

```
mkdir duckdb_odbc
unzip duckdb_odbc-linux-amd64.zip -d duckdb_odbc
```

Step 3: Configuring with unixODBC

The `unixodbc_setup.sh` script aids the configuration of the DuckDB ODBC Driver. It is based on the unixODBC package that provides some commands to handle the ODBC setup and test like `odbcinst` and `isql`.

In a terminal window, change to the `duckdb_odbc` permanent directory, and run the following commands with level options `-u` or `-s` either to configure DuckDB ODBC.

User-Level ODBC Setup (-u) The `-u` option based on the user home directory to setup the ODBC init files.

```
unixodbc_setup.sh -u
```

P.S.: The default configuration consists of a database `:memory:`.

System-Level ODBC setup (-s) The **-s** changes the system level files that will be visible for all users, because of that it requires root privileges.

```
sudo unixodbc_setup.sh -s
```

P.S.: The default configuration consists of a database `:memory:`.

Show Usage (--help) The option `--help` shows the usage of `unixodbc_setup.sh` that provides alternative options for a customer configuration, like `-db` and `-D`.

```
unixodbc_setup.sh --help
```

```
Usage: ./unixodbc_setup.sh <level> [options]

Example: ./unixodbc_setup.sh -u -db ~/database_path -D ~/driver_
 ↪  path/libduckdb_odbc.so

Level:
-s: System-level, using 'sudo' to configure DuckDB ODBC at the system-level,
 ↪  changing the files: /etc/odbc[inst].ini
-u: User-level, configuring the DuckDB ODBC at the user-level, changing the
 ↪  files: ~/.odbc[inst].ini.

Options:
```

```
-db database_path>: the DuckDB database file path, the default is ':memory:'
 ↪  if not provided.
-D driver_path: the driver file path (i.e., the path for libduckdb_odbc.so),
 ↪  the default is using the base script directory
```

Step 4 (Optional): Configure the ODBC Driver

The ODBC setup on Linux is based on files, the well-known `.odbc.ini` and `.odbcinst.ini`. These files can be placed at the system `/etc` directory or at the user home directory `/home/<user>` (shortcut as `~/`). The DM prioritizes the user configuration files and then the system files.

The `.odbc.ini` File The `.odbc.ini` contains the DSNs for the drivers, which can have specific knobs.

An example of `.odbc.ini` with DuckDB would be:

```
[DuckDB]
Driver = DuckDB Driver
Database=:memory:
```

[DuckDB]: between the brackets is a DSN for the DuckDB.

Driver: it describes the driver's name, and other configurations will be placed at the **.odbcinst.ini**.

Database: it describes the database name used by DuckDB, and it can also be a file path to a `.db` in the system.

The `.odbcinst.ini` File The `.odbcinst.ini` contains general configurations for the ODBC installed drivers in the system. A driver section starts with the driver name between brackets, and then it follows specific configuration knobs belonging to that driver.

An example of `.odbcinst.ini` with the DuckDB driver would be:

```
[ODBC]
Trace = yes
TraceFile = /tmp/odbctrace

[DuckDB Driver]
Driver = /home/<user>/duckdb_odbc/libduckdb_odbc.so
```

[ODBC]: it is the DM configuration section.

Trace: it enables the ODBC trace file using the option `yes`.

TraceFile: the absolute system file path for the ODBC trace file.

[DuckDB Driver]: the section of the DuckDB installed driver.

Driver: the absolute system file path of the DuckDB driver.

ODBC API - Windows

The Microsoft Windows requires an ODBC Driver Manager to manage communication between applications and the ODBC drivers. The DM on Windows is provided in a DLL file `odbccp32.dll`, and other files and tools. For detailed information checkout out the Common ODBC Component Files.

Step 1: Download ODBC Driver

DuckDB releases the ODBC driver as asset. For Windows, download it from Windows Asset that contains the following artifacts:

duckdb_odbc.dll: the DuckDB driver compiled for Windows.

duckdb_odbc_setup.dll: a setup DLL used by the Windows ODBC Data Source Administrator tool.

odbc_install.exe: an installation script to aid the configuration on Windows.

Step 2: Extracting ODBC artifacts

Unzip the file to a permanent directory (e.g., duckdb_odbc).

An example with `PowerShell` and `unzip` command would be:

```
mkdir duckdb_odbc
unzip duckdb_odbc-linux-amd64.zip -d duckdb_odbc
```

Step 3: ODBC Windows Installer

The `odbc_install.exe` aids the configuration of the DuckDB ODBC Driver on Windows. It depends on the `Odbccp32.dll` that provides functions to configure the ODBC registry entries.

Inside the permanent directory (e.g., `duckdb_odbc`), double-click on the `odbc_install.exe`.

Windows administrator privileges is required, in case of a non-administrator a User Account Control shall display:

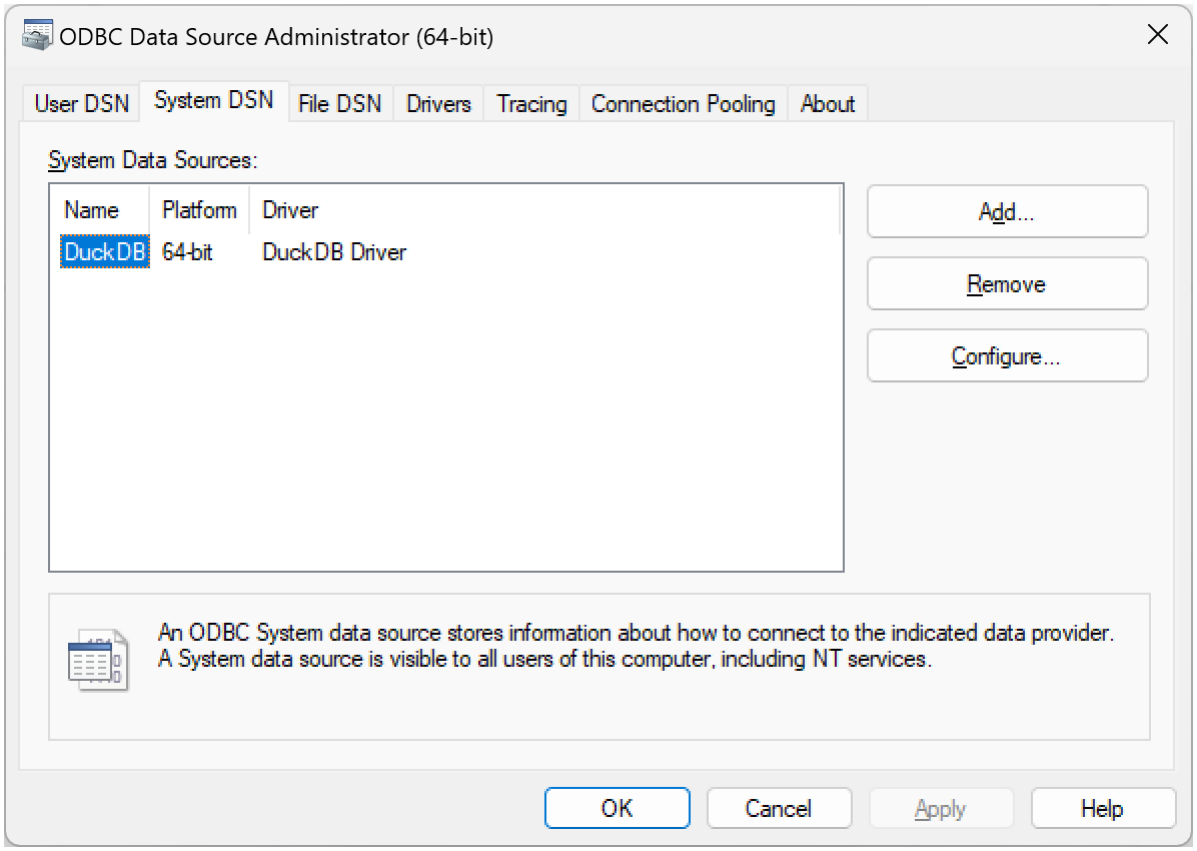
Step 4: Configure the ODBC Driver

The `odbc_install.exe` adds a default DSN configuration into the ODBC registries with a default database `:memory:`.

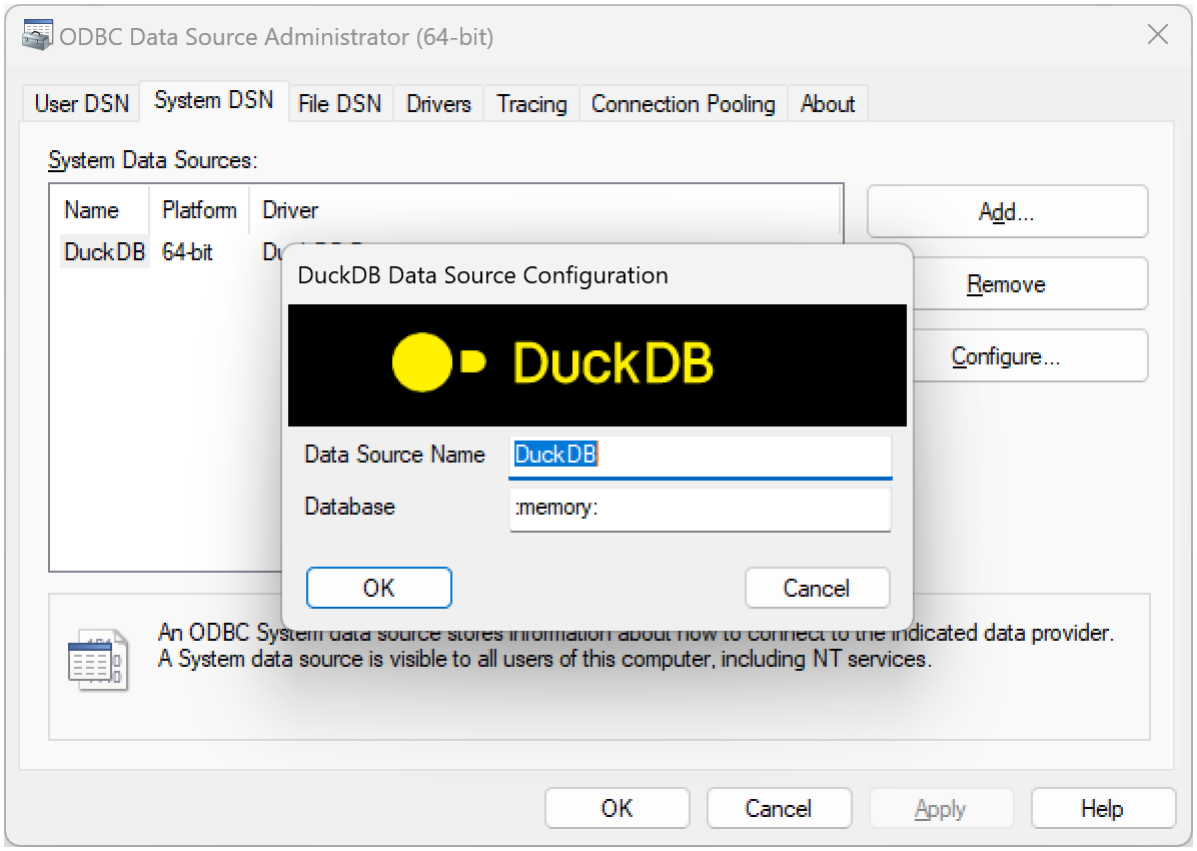
DSN Windows Setup After the installation, it is possible to change the default DSN configuration or add a new one using the Windows ODBC Data Source Administrator tool `odbcad32.exe`.

It also can be launched thought the Windows start:

Default DuckDB DSN The newly installed DSN is visible on the ***System DSN*** in the Windows ODBC Data Source Administrator tool:



Changing DuckDB DSN When selecting the default DSN (i.e., DuckDB) or adding a new configuration, the following setup window will display:



This window allows you to set the DSN and the database file path associated with that DSN.

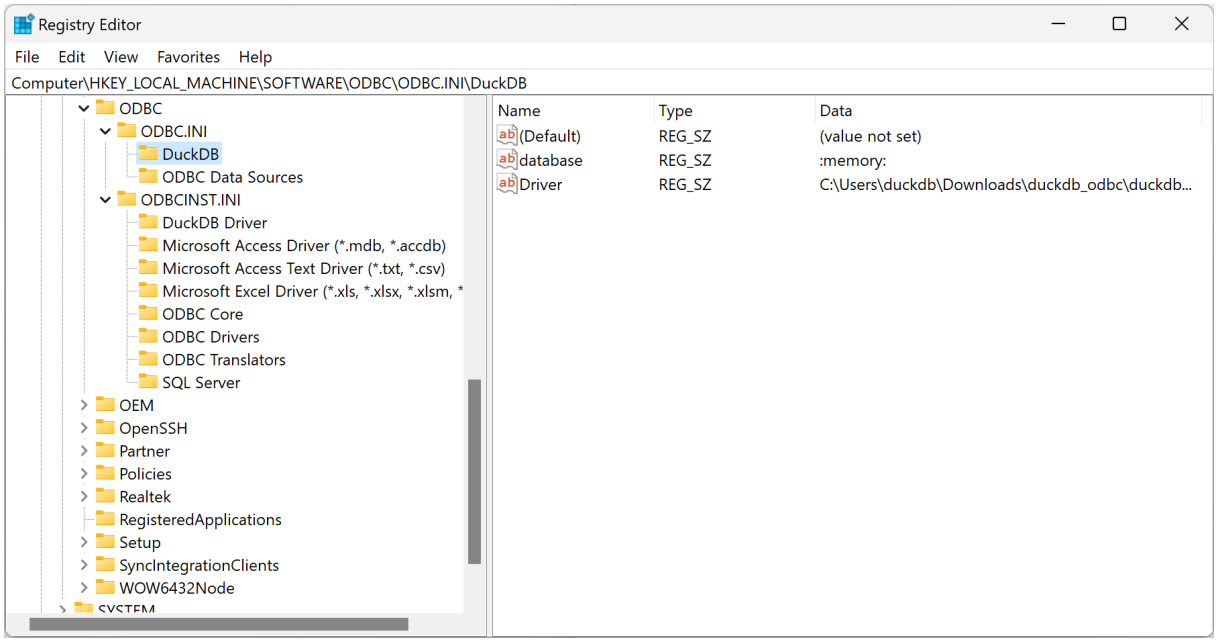
More Detailed Windows Setup

The ODBC setup on Windows is based on registry keys (see Registry Entries for ODBC Components). The ODBC entries can be placed at the current user registry key (`HKCU`) or the system registry key (`HKLM`).

We have tested and used the system entries based on `HKLM->SOFTWARE->ODBC`. The `odbc_install.exe` changes this entry that has two subkeys: `ODBC.INI` and `ODBCINST.INI`.

The `ODBC.INI` is where users usually insert DSN registry entries for the drivers.

For example, the DSN registry for DuckDB would look like this:



The `ODBCINST.INI` contains one entry for each ODBC driver and other keys predefined for Windows ODBC configuration.

ODBC API - MacOS

A driver manager is required to manage communication between applications and the ODBC driver. We tested and support `unixODBC` that is a complete ODBC driver manager for MacOS (and Linux). Users can install it from the command line:

Brew

```
brew install unixodbc
```

Step 1: Download ODBC Driver

DuckDB releases the ODBC driver as asset. For MacOS, download it from ODBC Linux Asset that contains the following artifacts:

libduckdb_odbc.dylib: the DuckDB driver compiled to MacOS (with Intel and Apple M1 support).

Step 2: Extracting ODBC Artifacts

Run unzip to extract the files to a permanent directory:

```
mkdir duckdb_odbc
unzip duckdb_odbc-osx-universal.zip -d duckdb_odbc
```

Step 3: Configure the ODBC Driver

The `odbc.ini` or `.odbc.ini` File The `.odbc.ini` contains the DSNs for the drivers, which can have specific knobs.

An example of `.odbc.ini` with DuckDB would be:

```
[DuckDB]
Driver = DuckDB Driver
Database=:memory:
```

[DuckDB]: between the brackets is a DSN for the DuckDB.

Driver: it describes the driver's name, and other configurations will be placed at the **.odbcinst.ini**.

Database: it describes the database name used by DuckDB, and it can also be a file path to a `.db` in the system.

The `.odbcinst.ini` File The `.odbcinst.ini` contains general configurations for the ODBC installed drivers in the system. A driver section starts with the driver name between brackets, and then it follows specific configuration knobs belonging to that driver.

An example of `.odbcinst.ini` with the DuckDB driver would be:

```
[ODBC]
Trace = yes
TraceFile = /tmp/odbctrace

[DuckDB Driver]
Driver = /User/<user>/duckdb_odbc/libduckdb_odbc.dylib
```

[ODBC]: it is the DM configuration section.

Trace: it enables the ODBC trace file using the option `yes`.

TraceFile: the absolute system file path for the ODBC trace file.

[DuckDB Driver]: the section of the DuckDB installed driver.

Driver: the absolute system file path of the DuckDB driver.

Step 4 (Optional): Test the ODBC Driver

After the configuration, for validate the installation, it is possible to use an odbc client. unixODBC use a command line tool called `isql`.

Use the DSN defined in `odbc.ini` as a parameter of `isql`.

```
isql DuckDB
+---------------------------------------+
| Connected!                            |
|                                       |
| sql-statement                         |
| help [tablename]                      |
| echo [string]                         |
| quit                                  |
|                                       |
+---------------------------------------+
SQL> SELECT 42;
+------------+
| 42         |
+------------+
| 42         |
+------------+

SQLRowCount returns -1
1 rows fetched
```

SQL

SQL Introduction

Here we provide an overview of how to perform simple operations in SQL. This tutorial is only intended to give you an introduction and is in no way a complete tutorial on SQL. This tutorial is adapted from the PostgreSQL tutorial.

In the examples that follow, we assume that you have installed the DuckDB Command Line Interface (CLI) shell. See the installation page for information on how to install the CLI. Launching the shell should give you the following prompt:

```
v0.9.1 401c8061c6
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
D
```

> **Note.** By launching the database like this, an **in-memory database is launched**. That means that no data is persisted on disk. To persist data on disk you should also pass a database path to the shell. The database will then be stored at that path and can be reloaded from disk later.

Concepts

DuckDB is a relational database management system (RDBMS). That means it is a system for managing data stored in relations. A relation is essentially a mathematical term for a table.

Each table is a named collection of rows. Each row of a given table has the same set of named columns, and each column is of a specific data type. Tables themselves are stored inside schemas, and a collection of schemas constitutes the entire database that you can access.

Creating a New Table

You can create a new table by specifying the table name, along with all column names and their types:

```
CREATE TABLE weather (
    city    VARCHAR,
    temp_lo INTEGER, -- minimum temperature on a day
    temp_hi INTEGER, -- maximum temperature on a day
    prcp    REAL,
    date    DATE
);
```

You can enter this into the shell with the line breaks. The command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) can be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. Two dash characters (`--`) introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers.

In the SQL command, we first specify the type of command that we want to perform: `CREATE TABLE`. After that follows the parameters for the command. First, the table name, `weather`, is given. Then the column names and column types follow.

`city VARCHAR` specifies that the table has a column called `city` that is of type `VARCHAR`. `VARCHAR` specifies a data type that can store text of arbitrary length. The temperature fields are stored in an `INTEGER` type, a type that stores integer numbers (i.e., whole numbers without a decimal point). `REAL` columns store single precision floating-point numbers (i.e., numbers with a decimal point). `DATE` stores a date (i.e., year, month, day combination). `DATE` only stores the specific day, not a time associated with that day.

DuckDB supports the standard SQL types `INTEGER`, `SMALLINT`, `REAL`, `DOUBLE`, `DECIMAL`, `CHAR(n)`, `VARCHAR(n)`, `DATE`, `TIME` and `TIMESTAMP`.

The second example will store cities and their associated geographical location:

```
CREATE TABLE cities (
    name VARCHAR,
    lat  DECIMAL,
    lon  DECIMAL
);
```

Finally, it should be mentioned that if you don't need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE [tablename];
```

Populating a Table with Rows

The insert statement is used to populate a table with rows:

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Constants that are not numeric values (e.g., text and dates) must be surrounded by single quotes (`''`), as in the example. Input dates for the date type must be formatted as `'YYYY-MM-DD'`.

We can insert into the `cities` table in the same manner.

```
INSERT INTO cities
VALUES ('San Francisco', -194.0, 53.0);
```

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the `prcp` is unknown:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Many developers consider explicitly listing the columns better style than relying on the order implicitly.

Please enter all the commands shown above so you have some data to work with in the following sections.

You could also have used COPY to load large amounts of data from CSV files. This is usually faster because the COPY command is optimized for this application while allowing less flexibility than `INSERT`. An example with `weather.csv` would be:

```
COPY weather
FROM 'weather.csv';
```

Where the file name for the source file must be available on the machine running the process. There are many other ways of loading data into DuckDB, see the corresponding documentation section for more information.

Querying a Table

To retrieve data from a table, the table is queried. A SQL `SELECT` statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the

part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). For example, to retrieve all the rows of table weather, type:

```
SELECT *
FROM weather;
```

Here * is a shorthand for "all columns". So the same result would be had with:

```
SELECT city, temp_lo, temp_hi, prcp, date
FROM weather;
```

The output should be:

city varchar	temp_lo int32	temp_hi int32	prcp float	date date
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0.0	1994-11-29
Hayward	37	54		1994-11-29

You can write expressions, not just simple column references, in the select list. For example, you can do:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date
FROM weather;
```

This should give:

city varchar	temp_avg double	date date
San Francisco	48.0	1994-11-27
San Francisco	50.0	1994-11-29
Hayward	45.5	1994-11-29

Notice how the AS clause is used to relabel the output column. (The AS clause is optional.)

A query can be "qualified" by adding a WHERE clause that specifies which rows are wanted. The WHERE clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (AND, OR, and NOT) are allowed in the qualification. For example, the following retrieves the weather of San Francisco on rainy days:

```
SELECT *
FROM weather
WHERE city = 'San Francisco' AND prcp > 0.0;
```

Result:

city varchar	temp_lo int32	temp_hi int32	prcp float	date date
San Francisco	46	50	0.25	1994-11-27

You can request that the results of a query be returned in sorted order:

```
SELECT *
FROM weather
ORDER BY city;
```

city varchar	temp_lo int32	temp_hi int32	prcp float	date date
Hayward	37	54		1994-11-29
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0.0	1994-11-29

In this example, the sort order isn't fully specified, and so you might get the San Francisco rows in either order. But you'd always get the results shown above if you do:

```
SELECT *
FROM weather
ORDER BY city, temp_lo;
```

You can request that duplicate rows be removed from the result of a query:

```
SELECT DISTINCT city
FROM weather;
```

city varchar
Hayward
San Francisco

Here again, the result row ordering might vary. You can ensure consistent results by using `DISTINCT` and `ORDER BY` together:

```
SELECT DISTINCT city
FROM weather
ORDER BY city;
```

Joins between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a join query. As an example, say you wish to list all the weather records together with the location of the associated city. To do that, we need to compare the city column of each row of the `weather` table with the name column of all rows in the `cities` table, and select the pairs of rows where these values match.

This would be accomplished by the following query:

```
SELECT *
FROM weather, cities
WHERE city = name;
```

```
┌───────────────┬─────────┬─────────┬───────┬────────────┬───────────────┬──────────────┬──────────────┐
│     city      │ temp_lo │ temp_hi │ prcp  │    date    │     name      │     lat      │     lon      │
│    varchar    │  int32  │  int32  │ float │    date    │    varchar    │ decimal(18,3) │ decimal(18,3) │
├───────────────┼─────────┼─────────┼───────┼────────────┼───────────────┼──────────────┼──────────────┤
│ San Francisco │      46 │      50 │  0.25 │ 1994-11-27 │ San Francisco │      -194.000 │       53.000 │
│ San Francisco │      43 │      57 │   0.0 │ 1994-11-29 │ San Francisco │      -194.000 │       53.000 │
└───────────────┴─────────┴─────────┴───────┴────────────┴───────────────┴──────────────┴──────────────┘
```

Observe two things about the result set:

- There is no result row for the city of Hayward. This is because there is no matching entry in the `cities` table for Hayward, so the join ignores the unmatched rows in the `weather` table. We will see shortly how this can be fixed.
- There are two columns containing the city name. This is correct because the lists of columns from the `weather` and `cities` tables are concatenated. In practice this is undesirable, though, so you will probably want to list the output columns explicitly rather than using `*`:

```
SELECT city, temp_lo, temp_hi, prcp, date, lon, lat
FROM weather, cities
WHERE city = name;
```

```
┌───────────────┬─────────┬─────────┬───────┬────────────┬──────────────┬──────────────┐
│     city      │ temp_lo │ temp_hi │ prcp  │    date    │     lon      │     lat      │
│    varchar    │  int32  │  int32  │ float │    date    │ decimal(18,3) │ decimal(18,3) │
├───────────────┼─────────┼─────────┼───────┼────────────┼──────────────┼──────────────┤
│ San Francisco │      46 │      50 │  0.25 │ 1994-11-27 │       53.000 │     -194.000 │
│ San Francisco │      43 │      57 │   0.0 │ 1994-11-29 │       53.000 │     -194.000 │
└───────────────┴─────────┴─────────┴───────┴────────────┴──────────────┴──────────────┘
```

Since the columns all had different names, the parser automatically found which table they belong to. If there were duplicate column names in the two tables you'd need to qualify the column names to show which one you meant, as in:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.lon, cities.lat
FROM weather, cities
WHERE cities.name = weather.city;
```

It is widely considered good style to qualify all column names in a join query, so that the query won't fail if a duplicate column name is later added to one of the tables.

Join queries of the kind seen thus far can also be written in this alternative form:

```
SELECT *
FROM weather
INNER JOIN cities ON weather.city = cities.name;
```

This syntax is not as commonly used as the one above, but we show it here to help you understand the following topics.

Now we will figure out how we can get the Hayward records back in. What we want the query to do is to scan the `weather` table and for each row to find the matching cities row(s). If no matching row is found we want some ”empty values” to be substituted for the `cities` table's columns. This kind of query is called an outer join. (The joins we have seen so far are inner joins.) The command looks like this:

```
SELECT *
FROM weather
LEFT OUTER JOIN cities ON weather.city = cities.name;
```

```
┌───────────────┬─────────┬─────────┬───────┬────────────┬───────────────┬
│     city      │ temp_lo │ temp_hi │ prcp  │    date    │     name      │
↪  lat      │      lon      │
│    varchar    │  int32  │  int32  │ float │    date    │    varchar    │
↪  decimal(18,3) │ decimal(18,3) │
├───────────────┼─────────┼─────────┼───────┼────────────┼───────────────┼
│ San Francisco │      46 │      50 │  0.25 │ 1994-11-27 │ San Francisco │
↪  -194.000 │        53.000 │
│ San Francisco │      43 │      57 │   0.0 │ 1994-11-29 │ San Francisco │
↪  -194.000 │        53.000 │
│ Hayward       │      37 │      54 │       │ 1994-11-29 │               │
↪  │               │
└───────────────┴─────────┴─────────┴───────┴────────────┴───────────────┴
```

This query is called a left outer join because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have those rows output that match some row of the left table. When outputting a left-table row for which there is no right-table match, empty (null) values are substituted for the right-table columns.

Aggregate Functions

Like most other relational database products, DuckDB supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the `count`, `sum`, `avg` (average), `max` (maximum) and `min` (minimum) over a set of rows.

As an example, we can find the highest low-temperature reading anywhere with:

```
SELECT max(temp_lo)
FROM weather;
```

```
┌──────────────┐
│ max(temp_lo) │
│    int32     │
├──────────────┤
│           46 │
└──────────────┘
```

If we wanted to know what city (or cities) that reading occurred in, we might try:

```
SELECT city
FROM weather
WHERE temp_lo = max(temp_lo);      -- WRONG
```

but this will not work since the aggregate max cannot be used in the WHERE clause. (This restriction exists because the WHERE clause determines which rows will be included in the aggregate calculation; so obviously it has to be evaluated before aggregate functions are computed.) However, as is often the case the query can be restated to accomplish the desired result, here by using a subquery:

```
SELECT city
FROM weather
WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

```
┌──────────────┐
│     city     │
│   varchar    │
├──────────────┤
│ San Francisco │
└──────────────┘
```

This is OK because the subquery is an independent computation that computes its own aggregate separately from what is happening in the outer query.

Aggregates are also very useful in combination with GROUP BY clauses. For example, we can get the maximum low temperature observed in each city with:

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city;
```

```
┌──────────────┬──────────────┐
│     city     │ max(temp_lo) │
│   varchar    │    int32     │
├──────────────┼──────────────┤
│ San Francisco │           46 │
│ Hayward      │           37 │
└──────────────┴──────────────┘
```

Which gives us one output row per city. Each aggregate result is computed over the table rows matching that city. We can filter these grouped rows using HAVING:

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city
HAVING max(temp_lo) < 40;
```

```
┌──────────┬──────────────┐
│   city   │ max(temp_lo) │
│ varchar  │    int32     │
```

```
┌──────────┬──────────────┐
│ Hayward  │           37 │
└──────────┴──────────────┘
```

which gives us the same results for only the cities that have all `temp_lo` values below 40. Finally, if we only care about cities whose names begin with "S", we can use the `LIKE` operator:

```
SELECT city, max(temp_lo)
FROM weather
WHERE city LIKE 'S%'            -- (1)
GROUP BY city
HAVING max(temp_lo) < 40;
```

More information about the `LIKE` operator can be found here.

It is important to understand the interaction between aggregates and SQL's `WHERE` and `HAVING` clauses. The fundamental difference between `WHERE` and `HAVING` is this: `WHERE` selects input rows before groups and aggregates are computed (thus, it controls which rows go into the aggregate computation), whereas `HAVING` selects group rows after groups and aggregates are computed. Thus, the `WHERE` clause must not contain aggregate functions; it makes no sense to try to use an aggregate to determine which rows will be inputs to the aggregates. On the other hand, the `HAVING` clause always contains aggregate functions.

In the previous example, we can apply the city name restriction in `WHERE`, since it needs no aggregate. This is more efficient than adding the restriction to `HAVING`, because we avoid doing the grouping and aggregate calculations for all rows that fail the `WHERE` check.

Updates

You can update existing rows using the `UPDATE` command. Suppose you discover the temperature readings are all off by 2 degrees after November 28. You can correct the data as follows:

```
UPDATE weather
SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
WHERE date > '1994-11-28';
```

Look at the new state of the data:

```
SELECT *
FROM weather;
```

```
┌──────────────┬─────────┬─────────┬─────────┬──────────────┐
│     city     │ temp_lo │ temp_hi │  prcp   │     date     │
│   varchar    │  int32  │  int32  │  float  │     date     │
```

San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0.0	1994-11-29
Hayward	35	52		1994-11-29

Deletions

Rows can be removed from a table using the DELETE command. Suppose you are no longer interested in the weather of Hayward. Then you can do the following to delete those rows from the table:

```
DELETE FROM weather
WHERE city = 'Hayward';
```

All weather records belonging to Hayward are removed.

```
SELECT *
FROM weather;
```

city varchar	temp_lo int32	temp_hi int32	prcp float	date date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0.0	1994-11-29

One should be wary of statements of the form

```
DELETE FROM tablename;
```

Without a qualification, DELETE will remove all rows from the given table, leaving it empty. The system will not request confirmation before doing this!

Statements

Statements Overview

Alter Table

The ALTER TABLE statement changes the schema of an existing table in the catalog.

Examples

```
-- add a new column with name "k" to the table "integers", it will be filled
 ↪  with the default value NULL
ALTER TABLE integers ADD COLUMN k INTEGER;
-- add a new column with name "l" to the table integers, it will be filled
 ↪  with the default value 10
ALTER TABLE integers ADD COLUMN l INTEGER DEFAULT 10;

-- drop the column "k" from the table integers
ALTER TABLE integers DROP k;

-- change the type of the column "i" to the type "VARCHAR" using a standard
 ↪  cast
ALTER TABLE integers ALTER i TYPE VARCHAR;
-- change the type of the column "i" to the type "VARCHAR", using the
 ↪  specified expression to convert the data for each row
ALTER TABLE integers ALTER i SET DATA TYPE VARCHAR USING CONCAT(i, '_', j);

-- set the default value of a column
ALTER TABLE integers ALTER COLUMN i SET DEFAULT 10;
-- drop the default value of a column
ALTER TABLE integers ALTER COLUMN i DROP DEFAULT;

-- make a column not nullable
ALTER TABLE t ALTER COLUMN x SET NOT NULL;
-- drop the not null constraint
ALTER TABLE t ALTER COLUMN x DROP NOT NULL;

-- rename a table
ALTER TABLE integers RENAME TO integers_old;

-- rename a column of a table
ALTER TABLE integers RENAME i TO j;
```

Syntax

`ALTER TABLE` changes the schema of an existing table. All the changes made by `ALTER TABLE` fully respect the transactional semantics, i.e., they will not be visible to other transactions until committed, and can be fully reverted through a rollback.

RENAME TABLE

```
-- rename a table
ALTER TABLE integers RENAME TO integers_old;
```

The `RENAME TO` clause renames an entire table, changing its name in the schema. Note that any views that rely on the table are **not** automatically updated.

RENAME COLUMN

```
-- rename a column of a table
ALTER TABLE integers RENAME i TO j;
ALTER TABLE integers RENAME COLUMN j TO k;
```

The `RENAME COLUMN` clause renames a single column within a table. Any constraints that rely on this name (e.g., `CHECK` constraints) are automatically updated. However, note that any views that rely on this column name are **not** automatically updated.

ADD COLUMN

```
-- add a new column with name "k" to the table "integers", it will be filled
 ↪  with the default value NULL
ALTER TABLE integers ADD COLUMN k INTEGER;
-- add a new column with name "l" to the table integers, it will be filled
 ↪  with the default value 10
ALTER TABLE integers ADD COLUMN l INTEGER DEFAULT 10;
```

The `ADD COLUMN` clause can be used to add a new column of a specified type to a table. The new column will be filled with the specified default value, or `NULL` if none is specified.

DROP COLUMN

```
-- drop the column "k" from the table integers
ALTER TABLE integers DROP k;
```

The `DROP COLUMN` clause can be used to remove a column from a table. Note that columns can only be removed if they do not have any indexes that rely on them. This includes any indexes created as part of a `PRIMARY KEY` or `UNIQUE` constraint. Columns that are part of multi-column check constraints cannot be dropped either.

ALTER TYPE

```
-- change the type of the column "i" to the type "VARCHAR" using a standard
 ↪  cast
ALTER TABLE integers ALTER i TYPE VARCHAR;
-- change the type of the column "i" to the type "VARCHAR", using the
 ↪  specified expression to convert the data for each row
ALTER TABLE integers ALTER i SET DATA TYPE VARCHAR USING CONCAT(i, '_', j);
```

The `SET DATA TYPE` clause changes the type of a column in a table. Any data present in the column is converted according to the provided expression in the `USING` clause, or, if the `USING` clause is absent, cast to the new data type. Note that columns can only have their type changed if they do not have any indexes that rely on them and are not part of any `CHECK` constraints.

SET/DROP DEFAULT

```
-- set the default value of a column
ALTER TABLE integers ALTER COLUMN i SET DEFAULT 10;
-- drop the default value of a column
ALTER TABLE integers ALTER COLUMN i DROP DEFAULT;
```

The `SET/DROP DEFAULT` clause modifies the `DEFAULT` value of an existing column. Note that this does not modify any existing data in the column. Dropping the default is equivalent to setting the default value to `NULL`.

> **Note.** At the moment DuckDB will not allow you to alter a table if there are any dependencies. That means that if you have an index on a column you will first need to drop the index, alter the table, and then recreate the index. Otherwise you will get a "Dependency Error."

ADD/DROP CONSTRAINT

> **Note.** The `ADD CONSTRAINT` and `DROP CONSTRAINT` clauses are not yet supported in DuckDB.

Alter View

The `ALTER VIEW` statement changes the schema of an existing view in the catalog.

Examples

```
-- rename a view
ALTER VIEW v1 RENAME TO v2;
```

`ALTER VIEW` changes the schema of an existing table. All the changes made by `ALTER VIEW` fully respect the transactional semantics, i.e., they will not be visible to other transactions until committed, and can be fully reverted through a rollback. Note that other views that rely on the table are **not** automatically updated.

Attach/Detach

The ATTACH statement adds a new database file to the catalog that can be read from and written to.

Examples

```
-- attach the database "file.db" with the alias inferred from the name
 ↪  ("file")
ATTACH 'file.db';
-- attach the database "file.db" with an explicit alias ("file_db")
ATTACH 'file.db' AS file_db;
-- attach the database "file.db" in read only mode
ATTACH 'file.db' (READ_ONLY);
-- attach a SQLite database for reading and writing (see sqlite extension
 ↪  for more information)
ATTACH 'sqlite_file.db' AS sqlite (TYPE SQLITE);
-- create a table in the attached database with alias "file"
CREATE TABLE file.new_table(i INTEGER);
-- detach the database with alias "file"
DETACH file;
-- show a list of all attached databases
SHOW databases;
-- change the default database that is used to the database "file"
USE file;
```

Syntax

ATTACH allows DuckDB to operate on multiple database files, and allows for transfer of data between different database files.

Detach

The DETACH statement allows previously attached database files to be closed and detached, releasing any locks held on the database file.

Name Qualification

The fully qualified name of catalog objects contains the *catalog*, the *schema* and the *name* of the object. For example:

```
-- attach the database "new_db"
ATTACH 'new_db.db';
-- create the schema "my_schema" in the database "new_db"
CREATE SCHEMA new_db.my_schema;
-- create the table "my_table" in the schema "my_schema"
CREATE TABLE new_db.my_schema.my_table(col INTEGER);
-- refer to the column "col" inside the table "my_table"
SELECT new_db.my_schema.my_table.col FROM new_db.my_schema.my_table;
```

Note that often the fully qualified name is not required. When a name is not fully qualified, the system looks for which entries to reference using the *catalog search path*. The default catalog search path includes the system catalog, the temporary catalog and the initially attached database together with the `main` schema.

Default Database and Schema When a table is created without any qualifications, the table is created in the default schema of the default database. The default database is the database that is launched when the system is created - and the default schema is `main`.

```
-- create the table "my_table" in the default database
CREATE TABLE my_table(col INTEGER);
```

Changing the Default Database and Schema The default database and schema can be changed using the USE command.

```
-- set the default database schema to `new_db.main`
USE new_db;
-- set the default database schema to `new_db.my_schema`
USE new_db.my_schema;
```

Resolving Conflicts When providing only a single qualification, the system can interpret this as *either* a catalog *or* a schema, as long as there are no conflicts. For example:

```
ATTACH 'new_db.db';
CREATE SCHEMA my_schema;
-- creates the table "new_db.main.tbl"
CREATE TABLE new_db.tbl(i INTEGER);
-- creates the table "default_db.my_schema.tbl"
CREATE TABLE my_schema.tbl(i INTEGER);
```

If we create a conflict (i.e., we have both a schema and a catalog with the same name) the system requests that a fully qualified path is used instead:

```
CREATE SCHEMA new_db;
CREATE TABLE new_db.tbl(i INTEGER);
-- Error: Binder Error: Ambiguous reference to catalog or schema "new_db" -
 ↪  use a fully qualified path like "memory.new_db"
```

Changing the Catalog Search Path The catalog search path can be adjusted by setting the `search_path` configuration option, which uses a comma-separated list of values that will be on the search path. The following example demonstrates searching in two databases:

```
ATTACH ':memory:' AS db1;
ATTACH ':memory:' AS db2;
CREATE table db1.tbl1 (i INTEGER);
CREATE table db2.tbl2 (j INTEGER);
-- reference the tables using their fully qualified name
SELECT * FROM db1.tbl1;
SELECT * FROM db2.tbl2;
-- or set the search path and reference the tables using their name
SET search_path='db1,db2';
SELECT * FROM tbl1;
SELECT * FROM tbl2;
```

Transactional Semantics

When running queries on multiple databases, the system opens separate transactions per database. The transactions are started *lazily* by default - when a given database is referenced for the first time in a query, a transaction for that database will be started. `SET immediate_transaction_mode=true` can be toggled to change this behavior to eagerly start transactions in all attached databases instead.

While multiple transactions can be active at a time - the system only supports *writing* to a single attached database in a single transaction. If you try to write to multiple attached databases in a single transaction the following error will be thrown:

```
Attempting to write to database "db2" in a transaction that has already
 ↪  modified database "db1" - a single transaction can only write to a
 ↪  single attached database.
```

The reason for this restriction is that the system does not maintain atomicity for transactions across attached databases. Transactions are only atomic *within* each database file. By restricting the global transaction to write to only a single database file the atomicity guarantees are maintained.

Call

The `CALL` statement invokes the given table function and returns the results.

Examples

```
-- Invoke the 'duckdb_functions' table function.
CALL duckdb_functions();
-- Invoke the 'pragma_table_info' table function.
CALL pragma_table_info('pg_am');
```

Syntax

Checkpoint

The `CHECKPOINT` statement synchronizes data in the write-ahead log (WAL) to the database data file. For in-memory databases this statement will succeed with no effect.

Examples

```
-- Synchronize data in the default database
CHECKPOINT;
-- Synchronize data in the specified database
CHECKPOINT file_db;
-- Abort any in-progress transactions to synchronize the data
FORCE CHECKPOINT;
```

Syntax

Checkpoint operations happen automatically based on the WAL size (see Configuration). This statement is for manual checkpoint actions.

Behavior

The default `CHECKPOINT` command will fail if there are any running transactions. Including `FORCE` will abort any transactions and execute the checkpoint operation.

Also see the related pragma for further behavior modification.

Copy

Examples

```
-- read a CSV file into the lineitem table - using auto-detected options
COPY lineitem FROM 'lineitem.csv' (AUTO_DETECT true);
-- read a parquet file into the lineitem table
COPY lineitem FROM 'lineitem.pq' (FORMAT PARQUET);
-- read a json file into the lineitem table - using auto-detected options
COPY lineitem FROM 'lineitem.json' (FORMAT JSON, AUTO_DETECT true);

-- write a table to a CSV file
COPY lineitem TO 'lineitem.csv' (FORMAT CSV, DELIMITER '|', HEADER);
-- write the result of a query to a Parquet file
COPY (SELECT l_orderkey, l_partkey FROM lineitem) TO 'lineitem.parquet'
 ↪  (COMPRESSION ZSTD);
```

COPY Statements

`COPY` moves data between DuckDB and external files. `COPY ... FROM` imports data into DuckDB from an external file. `COPY ... TO` writes data from DuckDB to an external file. The `COPY` command can be used for `CSV`, `PARQUET` and `JSON` files.

COPY FROM `COPY ... FROM` imports data from an external file into an existing table. The data is appended to whatever data is in the table already. The amount of columns inside the file must match the amount of columns in the table `table_name`, and the contents of the columns must be convertible to the column types of the table. In case this is not possible, an error will be thrown.

If a list of columns is specified, COPY will only copy the data in the specified columns from the file. If there are any columns in the table that are not in the column list, COPY ... FROM will insert the default values for those columns

```
-- Copy the contents of a comma-separated file 'test.csv' without a header
 ↪ into the table 'test'
COPY test FROM 'test.csv';
-- Copy the contents of a comma-separated file with a header into the
 ↪ 'category' table
COPY category FROM 'categories.csv' (HEADER);
-- Copy the contents of 'lineitem.tbl' into the 'lineitem' table, where the
 ↪ contents are delimited by a pipe character ('|')
COPY lineitem FROM 'lineitem.tbl' (DELIMITER '|');
-- Copy the contents of 'lineitem.tbl' into the 'lineitem' table, where the
 ↪ delimiter, quote character, and presence of a header are automatically
 ↪ detected
COPY lineitem FROM 'lineitem.tbl' (AUTO_DETECT true);
-- Read the contents of a comma-separated file 'names.csv' into the 'name'
 ↪ column of the 'category' table. Any other columns of this table are
 ↪ filled with their default value.
COPY category(name) FROM 'names.csv';
-- Read the contents of a parquet file 'lineitem.parquet' into the lineitem
 ↪ table
COPY lineitem FROM 'lineitem.parquet' (FORMAT PARQUET);
-- Read the contents of a newline-delimited json file 'lineitem.ndjson' into
 ↪ the lineitem table
COPY lineitem FROM 'lineitem.ndjson' (FORMAT JSON);
-- Read the contents of a json file 'lineitem.json' into the lineitem table
COPY lineitem FROM 'lineitem.json' (FORMAT JSON, ARRAY true);
```

Syntax

COPY TO COPY ... TO exports data from DuckDB to an external CSV or Parquet file. It has mostly the same set of options as COPY ... FROM, however, in the case of COPY ... TO the options specify how the file should be written to disk. Any file created by COPY ... TO can be copied back into the database by using COPY ... FROM with a similar set of options.

The COPY ... TO function can be called specifying either a table name, or a query. When a table name is specified, the contents of the entire table will be written into the resulting file. When a query is specified, the query is executed and the result of the query is written to the resulting file.

```
-- Copy the contents of the 'lineitem' table to the file 'lineitem.tbl',
 ↪  where the columns are delimited by a pipe character ('|'), including a
 ↪  header line.
COPY lineitem TO 'lineitem.tbl' (DELIMITER '|', HEADER);
-- Copy the l_orderkey column of the 'lineitem' table to the file
 ↪  'orderkey.tbl'
COPY lineitem(l_orderkey) TO 'orderkey.tbl' (DELIMITER '|');
-- Copy the result of a query to the file 'query.csv', including a header
 ↪  with column names
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.csv' WITH (HEADER 1, DELIMITER
 ↪  ',');
-- Copy the result of a query to the Parquet file 'query.parquet'
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.parquet' (FORMAT PARQUET);
-- Copy the result of a query to the newline-delimited JSON file
 ↪  'query.ndjson'
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.ndjson' (FORMAT JSON);
-- Copy the result of a query to the JSON file 'query.json'
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.json' (FORMAT JSON, ARRAY
 ↪  true);
```

Syntax

COPY Options Zero or more copy options may be provided as a part of the copy operation. The `WITH` specifier is optional, but if any options are specified, the parentheses are required. Parameter values can be passed in with or without wrapping in single quotes.

Any option that is a Boolean can be enabled or disabled in multiple ways. You can write `true`, `ON`, or `1` to enable the option, and `false`, `OFF`, or `0` to disable it. The Boolean value can also be omitted (e.g., by only passing `(HEADER)`), in which case `true` is assumed.

The below options are applicable to all formats written with `COPY`.

Name	Description	Type	Default
`allow_overwrite`	Whether or not to allow overwriting a directory if one already exists. Only has an effect when used with `partition_by`.	`BOOL`	`false`

Name	Description	Type	Default
`format`	Specifies the copy function to use. The default is selected from the file extension (e.g., `.parquet` results in a Parquet file being written/read). If the file extension is unknown CSV is selected. Available options are CSV, PARQUET and JSON.	VARCHAR	auto
`partition_by`	The columns to partition by using a hive partitioning scheme, see the partitioned writes section.	VARCHAR[]	(empty)
`per_thread_output`	Generate one file per thread, rather than one file in total. This allows for faster parallel writing.	BOOL	`false`
`use_tmp_file`	Whether or not to write to a temporary file first if the original file exists (`target.csv.tmp`). This prevents overwriting an existing file with a broken file in case the writing is cancelled.	BOOL	`auto`

CSV Options The below options are applicable when writing CSV files.

Name	Description	Type	Default
`compression`	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `file.csv.gz` will use gzip, `file.csv` will use none). Options are none, `gzip`, `zstd`.	VARCHAR	`auto`
`force_quote`	The list of columns to always add quotes to, even if not required.	VARCHAR[]	`[]`
`dateformat`	Specifies the date format to use when writing dates. See Date Format	VARCHAR	(empty)

Name	Description	Type	Default
`delim` or `sep`	The character that is written to separate columns within each row.	`VARCHAR`	`,`
`escape`	The character that should appear before a character that matches the `quote` value.	`VARCHAR`	`"`
`header`	Whether or not to write a header for the CSV file.	`BOOL`	`true`
`nullstr`	The string that is written to represent a NULL value.	`VARCHAR`	(empty)
`quote`	The quoting character to be used when a data value is quoted.	`VARCHAR`	`"`
`timestampformat`	Specifies the date format to use when writing timestamps. See Date Format	`VARCHAR`	(empty)

Parquet Options The below options are applicable when writing `Parquet` files.

Name	Description	Type	Default
`compression`	The compression format to use (`uncompressed`, `snappy`, `gzip` or `zstd`).	`VARCHAR`	`snappy`
`row_group_size`	The target size, i.e., number of rows, of each row-group.	`BIGINT`	122880
`row_group_size_bytes`	The target size of each row-group. You can pass either a human-readable string, e.g., '2MB', or an integer, i.e., the number of bytes. This option is only used when you have issued `SET preserve_insertion_order=false;`, otherwise it is ignored.	`BIGINT`	`row_group_size * 1024`
`field_ids`	The `field_id` for each column. Pass `auto` to attempt to infer automatically.	`STRUCT`	(empty)

Some examples of `FIELD_IDS` are:

```
-- Assign field_ids automatically
COPY (SELECT 128 AS i)
TO 'my.parquet' (FIELD_IDS 'auto');
-- Sets the field_id of column 'i' to 42
COPY (SELECT 128 AS i)
TO 'my.parquet' (FIELD_IDS {i: 42});
-- Sets the field_id of column 'i' to 42, and column 'j' to 43
COPY (SELECT 128 AS i, 256 AS j)
TO 'my.parquet' (FIELD_IDS {i: 42, j: 43});
-- Sets the field_id of column 'my_struct' to 43,
-- and column 'i' (nested inside 'my_struct') to 43
COPY (SELECT {i: 128} AS my_struct)
TO 'my.parquet' (FIELD_IDS {my_struct: {__duckdb_field_id: 42, i: 43}});
-- Sets the field_id of column 'my_list' to 42,
-- and column 'element' (default name of list child) to 43
COPY (SELECT [128, 256] AS my_list)
TO 'my.parquet' (FIELD_IDS {my_list: {__duckdb_field_id: 42, element: 43}});
-- Sets the field_id of colum 'my_map' to 42,
-- and columns 'key' and 'value' (default names of map children) to 43 and 44
COPY (SELECT map {'key1' : 128, 'key2': 256} my_map)
TO 'my.parquet' (FIELD_IDS {my_map: {__duckdb_field_id: 42, key: 43, value:
↪  44}});
```

JSON Options The below options are applicable when writing JSON files.

Name	Description	Type	Default
`compression`	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `file.csv.gz` will use gzip, `file.csv` will use none). Options are none, `gzip`, `zstd`.	VARCHAR	`auto`
`dateformat`	Specifies the date format to use when writing dates. See Date Format	VARCHAR	(empty)
`timestampformat`	Specifies the date format to use when writing timestamps. See Date Format	VARCHAR	(empty)

Name	Description	Type	Default
`array`	Whether to write a JSON array. If `true`, a JSON array of records is written, if `false`, newline-delimited JSON is written	`BOOL`	`false`

Create Macro

The `CREATE MACRO` statement can create a scalar or table macro (function) in the catalog. A macro may only be a single `SELECT` statement (similar to a `VIEW`), but it has the benefit of accepting parameters. For a scalar macro, `CREATE MACRO` is followed by the name of the macro, and optionally parameters within a set of parentheses. The keyword `AS` is next, followed by the text of the macro. By design, a scalar macro may only return a single value. For a table macro, the syntax is similar to a scalar macro except `AS` is replaced with `AS TABLE`. A table macro may return a table of arbitrary size and shape.

If a `MACRO` is temporary, it is only usable within the same database connection and is deleted when the connection is closed.

Examples

```
-- create a macro that adds two expressions (a and b)
CREATE MACRO add(a, b) AS a + b;
-- create a macro for a case expression
CREATE MACRO ifelse(a, b, c) AS CASE WHEN a THEN b ELSE c END;
-- create a macro that does a subquery
CREATE MACRO one() AS (SELECT 1);
-- create a macro with a common table expression
-- (parameter names get priority over column names: disambiguate using the
 ↪  table name)
CREATE MACRO plus_one(a) AS (WITH cte AS (SELECT 1 AS a) SELECT cte.a + a
 ↪  FROM cte);
-- macro's are schema-dependent, and have an alias: FUNCTION
CREATE FUNCTION main.myavg(x) AS SUM(x) / COUNT(x);
-- create a macro with default constant parameters
CREATE MACRO add_default(a, b := 5) AS a + b;
-- create a macro arr_append (with a functionality equivalent to array_
 ↪  append)
CREATE MACRO arr_append(l, e) AS list_concat(l, list_value(e));
```

```
-- TABLE MACROS
-- create a table macro without parameters
CREATE MACRO static_table() AS TABLE SELECT 'Hello' AS column1, 'World' AS
 ↪  column2;
-- create a table macro with parameters (that can be of any type)
CREATE MACRO dynamic_table(col1_value, col2_value) AS TABLE SELECT col1_
 ↪  value AS column1, col2_value AS column2;
-- create a table macro that returns multiple rows.
-- It will be replaced if it already exists, and it is temporary (will be
 ↪  automatically deleted when the connection ends)
CREATE OR REPLACE TEMP MACRO dynamic_table(col1_value, col2_value) AS TABLE
    SELECT col1_value AS column1, col2_value AS column2
    UNION ALL
    SELECT 'Hello' AS col1_value, 456 AS col2_value;
```

Syntax

Macros allow you to create shortcuts for combinations of expressions.

```
-- failure! cannot find column "b"
CREATE MACRO add(a) AS a + b;
-- this works
CREATE MACRO add(a, b) AS a + b;
-- error! cannot bind +(VARCHAR, INTEGER)
SELECT add('hello', 3);
-- success!
SELECT add(1, 2);
-- 3
```

Macro's can have default parameters. Unlike some languages, default parameters must be named when the macro is invoked.

```
-- b is a default parameter
CREATE MACRO add_default(a, b := 5) AS a + b;
-- the following will result in 42
SELECT add_default(37);
-- error! add_default only has one positional parameter
SELECT add_default(40, 2);
-- success! default parameters are used by assigning them like so
SELECT add_default(40, b:=2);
-- error! default parameters must come after positional parameters
SELECT add_default(b=2, 40);
```

```
-- the order of default parameters does not matter
CREATE MACRO triple_add(a, b := 5, c := 10) AS a + b + c;
-- success!
SELECT triple_add(40, c := 1, b := 1);
-- 42
```

When macro's are used, they are expanded (i.e., replaced with the original expression), and the parameters within the expanded expression are replaced with the supplied arguments. Step by step:

```
-- the 'add' macro we defined above is used in a query
SELECT add(40, 2);
-- internally, add is replaced with its definition of a + b
SELECT a + b;
-- then, the parameters are replaced by the supplied arguments
SELECT 40 + 2;
-- 42
```

Create Schema

The `CREATE SCHEMA` statement creates a schema in the catalog. The default schema is `main`.

Examples

```
-- create a schema
CREATE SCHEMA s1;
-- create a schema if it does not exist yet
CREATE SCHEMA IF NOT EXISTS s2;
-- create table in the schemas
CREATE TABLE s1.t(id INTEGER PRIMARY KEY, other_id INTEGER);
CREATE TABLE s2.t(id INTEGER PRIMARY KEY, j VARCHAR);
-- compute a join between tables from two schemas
SELECT * FROM s1.t s1t, s2.t s2t WHERE s1t.other_id = s2t.id;
```

Syntax

Create Sequence

The `CREATE SEQUENCE` statement creates a new sequence number generator.

Examples

```
-- generate an ascending sequence starting from 1
CREATE SEQUENCE serial;
-- generate sequence from a given start number
CREATE SEQUENCE serial START 101;
-- generate odd numbers using INCREMENT BY
CREATE SEQUENCE serial START WITH 1 INCREMENT BY 2;
-- generate a descending sequqnce starting from 99
CREATE SEQUENCE serial START WITH 99 INCREMENT BY -1 MAXVALUE 99;
-- by default, cycles are not allowed and will result in a Serialization
 ↪  Error, e.g.:
-- reached maximum value of sequence "serial" (10)
CREATE SEQUENCE serial START WITH 1 MAXVALUE 10;
-- CYCLE allows cycling through the same sequence repeatedly
CREATE SEQUENCE serial START WITH 1 MAXVALUE 10 CYCLE;
```

Sequences can be created and dropped similarly to other catalogue items:

```
-- overwrite an existing sequence
CREATE OR REPLACE SEQUENCE serial;
-- only create sequence if no such sequence exists yet
CREATE SEQUENCE IF NOT EXISTS serial;
-- remove sequence
DROP SEQUENCE serial;
-- remove sequence if exists
DROP SEQUENCE IF EXISTS serial;
```

Selecting the Next Value Select the next number from a sequence:

```
SELECT nextval('serial') AS nextval;
```

nextval
int64
1

Using this sequence in an INSERT command:

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

Selecting the Current Value You may also view the current number from the sequence. Note that the `nextval` function must have already been called before calling `currval`, otherwise a Serialization Error ("sequence is not yet defined in this session") will be thrown.

```
SELECT currval('serial') AS currval;
```

currval
int64
1

Syntax `CREATE SEQUENCE` creates a new sequence number generator.

If a schema name is given then the sequence is created in the specified schema. Otherwise it is created in the current schema. Temporary sequences exist in a special schema, so a schema name may not be given when creating a temporary sequence. The sequence name must be distinct from the name of any other sequence in the same schema.

After a sequence is created, you use the function `nextval` to operate on the sequence.

Parameters

Name	Description
`TEMPORARY` or `TEMP`	If specified, the sequence object is created only for this session, and is automatically dropped on session exit. Existing permanent sequences with the same name are not visible (in this session) while the temporary sequence exists, unless they are referenced with schema-qualified names.
name	The name (optionally schema-qualified) of the sequence to be created.
`increment`	The optional clause `INCREMENT BY increment` specifies which value is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

Name	Description
`minvalue`	The optional clause `MINVALUE minvalue` determines the minimum value a sequence can generate. If this clause is not supplied or `NO MINVALUE` is specified, then defaults will be used. The defaults are 1 and -(2^63 - 1) for ascending and descending sequences, respectively.
`maxvalue`	The optional clause `MAXVALUE maxvalue` determines the maximum value for the sequence. If this clause is not supplied or `NO MAXVALUE` is specified, then default values will be used. The defaults are 2^63 - 1 and -1 for ascending and descending sequences, respectively.
`start`	The optional clause `START WITH start` allows the sequence to begin anywhere. The default starting value is `minvalue` for ascending sequences and `maxvalue` for descending ones.
`CYCLE` or `NO CYCLE`	The `CYCLE` option allows the sequence to wrap around when the `maxvalue` or `minvalue` has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the `minvalue` or `maxvalue`, respectively.

If `NO CYCLE` is specified, any calls to nextval after the sequence has reached its maximum value will return an error. If neither `CYCLE` or `NO CYCLE` are specified, `NO CYCLE` is the default.

> **Note.** Sequences are based on `BIGINT` arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807).

Create Table

The `CREATE TABLE` statement creates a table in the catalog.

Examples

```
-- create a table with two integer columns (i and j)
CREATE TABLE t1(i INTEGER, j INTEGER);
-- create a table with a primary key
CREATE TABLE t1(id INTEGER PRIMARY KEY, j VARCHAR);
```

```
-- create a table with a composite primary key
CREATE TABLE t1(id INTEGER, j VARCHAR, PRIMARY KEY(id, j));
-- create a table with various different types and constraints
CREATE TABLE t1(i INTEGER NOT NULL, decimalnr DOUBLE CHECK(decimalnr<10),
 ↪  date DATE UNIQUE, time TIMESTAMP);
-- create a table from the result of a query
CREATE TABLE t1 AS SELECT 42 AS i, 84 AS j;
-- create a table from a CSV file using AUTO-DETECT (i.e., automatically
 ↪  detecting column names and types)
CREATE TABLE t1 AS SELECT * FROM read_csv_auto ('path/file.csv');
-- we can use the FROM-first syntax to omit 'SELECT *'
CREATE TABLE t1 AS FROM read_csv_auto ('path/file.csv');
```

Temporary Tables

Temporary tables can be created using a `CREATE TEMP TABLE` statement (see diagram below). Temporary tables are session scoped (similar to PostgreSQL for example), meaning that only the specific connection that created them can access them, and once the connection to DuckDB is closed they will be automatically dropped. Temporary tables reside in memory rather than on disk (even when connecting to a persistent DuckDB), but if the `temp_directory` configuration is set when connecting or with a `SET` command, data will be spilled to disk if memory becomes constrained.

```
-- create a temporary table from a CSV file using AUTO-DETECT (i.e.,
 ↪  automatically detecting column names and types)
CREATE TEMP TABLE t1 AS SELECT * FROM read_csv_auto ('path/file.csv');

-- allow temporary tables to off-load excess memory to disk
SET temp_directory='/path/to/directory/';
```

Create or Replace

The `CREATE OR REPLACE` syntax allows a new table to be created or for an existing table to be overwritten by the new table. This is shorthand for dropping the existing table and then creating the new one.

```
-- create a table with two integer columns (i and j) even if t1 already
 ↪  exists
CREATE OR REPLACE TABLE t1(i INTEGER, j INTEGER);
```

If Not Exists

The `IF NOT EXISTS` syntax will only proceed with the creation of the table if it does not already exist. If the table already exists, no action will be taken and the existing table will remain in the database.

```
-- create a table with two integer columns (i and j) only if t1 does not
 ↪  exist yet.
CREATE TABLE IF NOT EXISTS t1(i INTEGER, j INTEGER);
```

Check Constraints

A CHECK constraint is an expression that must be satisfied by the values of every row in the table.

```
CREATE TABLE t1(
    id INTEGER PRIMARY KEY,
    percentage INTEGER CHECK(0 <= percentage AND percentage <= 100)
);
INSERT INTO t1 VALUES (1, 5);
INSERT INTO t1 VALUES (2, -1);
-- Error: Constraint Error: CHECK constraint failed: t1
INSERT INTO t1 VALUES (3, 101);
-- Error: Constraint Error: CHECK constraint failed: t1

CREATE TABLE t2(id INTEGER PRIMARY KEY, x INTEGER, y INTEGER CHECK(x < y));
INSERT INTO t2 VALUES (1, 5, 10);
INSERT INTO t2 VALUES (2, 5, 3);
-- Error: Constraint Error: CHECK constraint failed: t2
```

CHECK constraints can also be added as part of the `CONSTRAINTS` clause:

```
CREATE TABLE t3(
    id INTEGER PRIMARY KEY,
    x INTEGER,
    y INTEGER,
    CONSTRAINT x_smaller_than_y CHECK(x < y)
);
INSERT INTO t3 VALUES (1, 5, 10);
INSERT INTO t3 VALUES (2, 5, 3);
-- Error: Constraint Error: CHECK constraint failed: t3
```

Foreign Key Constraints

A `FOREIGN KEY` is a column (or set of columns) that references another table's primary key. Foreign keys check referential integrity, i.e., the referred primary key must exist in the other table upon insertion.

```
CREATE TABLE t1(id INTEGER PRIMARY KEY, j VARCHAR);
CREATE TABLE t2(
    id INTEGER PRIMARY KEY,
    t1_id INTEGER,
    FOREIGN KEY (t1_id) REFERENCES t1(id)
);

-- example
INSERT INTO t1 VALUES (1, 'a');
INSERT INTO t2 VALUES (1, 1);
INSERT INTO t2 VALUES (2, 2);
-- Error: Constraint Error: Violates foreign key constraint because key "id:
 ↪  2" does not exist in the referenced table
```

Foreign keys can be defined on composite primary keys:

```
CREATE TABLE t3(id INTEGER, j VARCHAR, PRIMARY KEY(id, j));
CREATE TABLE t4(
    id INTEGER PRIMARY KEY, t3_id INTEGER, t3_j VARCHAR,
    FOREIGN KEY (t3_id, t3_j) REFERENCES t3(id, j)
);

-- example
INSERT INTO t3 VALUES (1, 'a');
INSERT INTO t4 VALUES (1, 1, 'a');
INSERT INTO t4 VALUES (2, 1, 'b');
-- Error: Constraint Error: Violates foreign key constraint because key "id:
 ↪  1, j: b" does not exist in the referenced table
```

Foreign keys can also be defined on unique columns:

```
CREATE TABLE t5(id INTEGER UNIQUE, j VARCHAR);
CREATE TABLE t6(id INTEGER PRIMARY KEY, t5_id INTEGER, FOREIGN KEY (t5_id)
 ↪  REFERENCES t5(id));
```

> **Note.** Foreign keys with cascading deletes (`FOREIGN KEY ... REFERENCES ... ON DELETE CASCADE`) are not supported.

Generated Columns

The `[type] [GENERATED ALWAYS] AS (expr) [VIRTUAL|STORED]` syntax will create a generated column. The data in this kind of column is generated from its expression, which can reference other (regular or generated) columns of the table. Since they are produced by calculations, these columns can not be inserted into directly.

DuckDB can infer the type of the generated column based on the expression's return type. This allows you to leave out the type when declaring a generated column. It is possible to explicitly set a type, but insertions into the referenced columns might fail if the type can not be cast to the type of the generated column.

Generated columns come in two varieties: `VIRTUAL` and `STORED`.
The data of virtual generated columns is not stored on disk, instead it is computed from the expression every time the column is referenced (through a select statement).

The data of stored generated columns is stored on disk and is computed every time the data of their dependencies change (through an insert/update/drop statement).

Currently only the `VIRTUAL` kind is supported, and it is also the default option if the last field is left blank.

```
-- The simplest syntax for a generated column.
-- The type is derived from the expression, and the variant defaults to
 ↪  VIRTUAL
CREATE TABLE t1(x FLOAT, two_x AS (2 * x));

-- Fully specifying the same generated column for completeness
CREATE TABLE t1(x FLOAT, two_x FLOAT GENERATED ALWAYS AS (2 * x) VIRTUAL);
```

Syntax

Create View

The `CREATE VIEW` statement defines a new view in the catalog.

Examples

```
-- create a simple view
CREATE VIEW v1 AS SELECT * FROM tbl;
-- create a view or replace it if a view with that name already exists
CREATE OR REPLACE VIEW v1 AS SELECT 42;
```

```
-- create a view and replace the column names
CREATE VIEW v1(a) AS SELECT 42;
```

Syntax

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

`CREATE OR REPLACE VIEW` is similar, but if a view of the same name already exists, it is replaced.

If a schema name is given then the view is created in the specified schema. Otherwise it is created in the current schema. Temporary views exist in a special schema, so a schema name cannot be given when creating a temporary view. The name of the view must be distinct from the name of any other view or table in the same schema.

Create Type

The `CREATE TYPE` statement defines a new type in the catalog.

Examples

```
-- create a simple enum type
CREATE TYPE mood AS ENUM ('happy', 'sad', 'curious');
-- create a simple struct type
CREATE TYPE many_things AS STRUCT(k INTEGER, l VARCHAR);
-- create a simple union type
CREATE TYPE one_thing AS UNION(number INTEGER, string VARCHAR);
-- create a type alias
CREATE TYPE x_index AS INTEGER;
```

Syntax

`CREATE TYPE` defines a new data type available to this duckdb instance. These new types can then be inspected in the `duckdb_types` table.

Extending these custom types to support custom operators (such as the PostgreSQL && operator) would require C++ development. To do this, create an extension.

Delete Statement

The DELETE statement removes rows from the table identified by the table-name.

Examples

```
-- remove the rows matching the condition "i=2" from the database
DELETE FROM tbl WHERE i=2;
-- delete all rows in the table "tbl"
DELETE FROM tbl;
```

Syntax

The DELETE statement removes rows from the table identified by the table-name.

If the WHERE clause is not present, all records in the table are deleted. If a WHERE clause is supplied, then only those rows for which the WHERE clause results in true are deleted. Rows for which the expression is false or NULL are retained.

The USING clause allows deleting based on the content of other tables or subqueries.

Drop Statement

The DROP statement removes a catalog entry added previously with the CREATE command.

Examples

```
-- delete the table with the name "tbl"
DROP TABLE tbl;
-- drop the view with the name "v1"; do not throw an error if the view does
 ↪  not exist
DROP VIEW IF EXISTS v1;
```

Syntax

The optional IF EXISTS clause suppresses the error that would normally result if the table does not exist.

By default (or if the RESTRICT clause is provided), the entry will not be dropped if there are any other objects that depend on it. If the CASCADE clause is provided then all the objects that are dependent on the object will be dropped as well.

```
CREATE SCHEMA myschema;
CREATE TABLE myschema.t1(i INTEGER);
-- ERROR: Cannot drop myschema because the table myschema.t1 depends on it.
DROP SCHEMA myschema;
-- Cascade drops both myschema and myschema.t1
DROP SCHEMA myschema CASCADE;
```

Export & Import Database

The EXPORT DATABASE command allows you to export the contents of the database to a specific directory. The IMPORT DATABASE command allows you to then read the contents again.

Examples

```
-- export the database to the target directory
EXPORT DATABASE 'target_directory';
-- export the table contents with the given options
EXPORT DATABASE 'target_directory' (FORMAT CSV, DELIMITER '|');
-- export the table contents as parquet
EXPORT DATABASE 'target_directory' (FORMAT PARQUET);
-- export as parquet, compressed with ZSTD, with a row_group_size of 100000
EXPORT DATABASE 'target_directory' (FORMAT PARQUET, COMPRESSION ZSTD, ROW_
 ↪  GROUP_SIZE 100000);
--reload the database again
IMPORT DATABASE 'target_directory';
```

For details regarding the writing of Parquet files, see the Parquet Files page in the Data Import section, and the COPY Statement page.

Syntax

The EXPORT DATABASE command exports the full contents of the database - including schema information, tables, views and sequences - to a specific directory that can then be loaded again. The created directory will be structured as follows:

```
target_directory/schema.sql
target_directory/load.sql
```

```
target_directory/t_1.csv
...
target_directory/t_n.csv
```

The `schema.sql` file contains the schema statements that are found in the database. It contains any `CREATE SCHEMA`, `CREATE TABLE`, `CREATE VIEW` and `CREATE SEQUENCE` commands that are necessary to re-construct the database.

The `load.sql` file contains a set of `COPY` statements that can be used to read the data from the CSV files again. The file contains a single `COPY` statement for every table found in the schema.

The database can be reloaded by using the `IMPORT DATABASE` command again, or manually by running `schema.sql` followed by `load.sql` to re-load the data.

Insert Statement

The `INSERT` statement inserts new data into a table.

Examples

```
-- insert the values (1), (2), (3) into "tbl"
INSERT INTO tbl VALUES (1), (2), (3);
-- insert the result of a query into a table
INSERT INTO tbl SELECT * FROM other_tbl;
-- insert values into the "i" column, inserting the default value into other
 ↪  columns
INSERT INTO tbl(i) VALUES (1), (2), (3);
-- explicitly insert the default value into a column
INSERT INTO tbl(i) VALUES (1), (DEFAULT), (3);
-- assuming tbl has a primary key/unique constraint, do nothing on conflict
INSERT OR IGNORE INTO tbl(i) VALUES(1);
-- or update the table with the new values instead
INSERT OR REPLACE INTO tbl(i) VALUES(1);
```

Syntax `INSERT INTO` inserts new rows into a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query.

Insert Column Order

It's possible to provide an optional insert column order, this can either be `BY POSITION` (the default) or `BY NAME`. Each column not present in the explicit or implicit column list will be filled with a default

value, either its declared default value or NULL if there is none.

If the expression for any column is not of the correct data type, automatic type conversion will be attempted.

BY POSITION The order that values are inserted into the columns of the table is determined by the order that the columns were declared in. This can be overridden by providing column names as part of the target, for example:

```
CREATE TABLE tbl(a INTEGER, b INTEGER);
INSERT INTO tbl(b, a) VALUES (5, 42);
```

This will insert 5 into b and 42 into a. The values supplied by the VALUES clause or query are associated with the column list left-to-right.

BY NAME The names of the column list of the SELECT statement are matched against the column names of the table to determine the order that values should be inserted into the table, even if the order of the columns in the table differs from the order of the values in the SELECT statement. For example:

```
CREATE TABLE tbl(a INTEGER, b INTEGER);
INSERT INTO tbl BY NAME (SELECT 42 AS b);
```

This will insert 42 into b and insert NULL (or its default value) into a.

It's important to note that when using `INSERT INTO <table> BY NAME`, the column names specified in the SELECT statement must match the column names in the table.
If a column name is misspelled or does not exist in the table, an error will occur.

This is not a problem however if columns are missing from the SELECT statement, as those will be filled with the default value.

ON CONFLICT Clause

An `ON CONFLICT` clause can be used to perform a certain action on conflicts that arise from `UNIQUE` or `PRIMARY KEY` constraints.

A `conflict_target` may also be provided, which is a group of columns that an Index indexes on, or if left out, all `UNIQUE` or `PRIMARY KEY` constraint(s) on the table are targeted. The `conflict_target` is optional unless using a `DO UPDATE` (see below) and there are multiple unique/primary key constraints on the table.

When a conflict target is provided, you can further filter this with a `WHERE` clause, that should be met by all conflicts. If a conflict does not meet this condition, an error will be thrown instead, and the entire operation is aborted.

Because we need a way to refer to both the **to-be-inserted** tuple and the **existing** tuple, we introduce the special `excluded` qualifier. When the `excluded` qualifier is provided, the reference refers to the **to-be-inserted** tuple, otherwise it refers to the **existing** tuple This special qualifier can be used within the `WHERE` clauses and `SET` expressions of the `ON CONFLICT` clause.

There are two supported actions:

1. `DO NOTHING`
   Causes the error(s) to be ignored, and the values are not inserted or updated.
2. `DO UPDATE`
   Causes the `INSERT` to turn into an `UPDATE` on the conflicting row(s) instead.
   The `SET` expressions that follow determine how these rows are updated.
   Optionally you can provide an additional `WHERE` clause that can exclude certain rows from the update.
   The conflicts that don't meet this condition are ignored instead.

`INSERT OR REPLACE` is a shorter syntax alternative to `ON CONFLICT DO UPDATE SET (c1 = excluded.c1, c2 = excluded.c2, ..)`.
It updates every column of the **existing** row to the new values of the **to-be-inserted** row.

`INSERT OR IGNORE` is a shorter syntax alternative to `ON CONFLICT DO NOTHING`.

RETURNING Clause The `RETURNING` clause may be used to return the contents of the rows that were inserted. This can be useful if some columns are calculated upon insert. For example, if the table contains an automatically incrementing primary key, then the `RETURNING` clause will include the automatically created primary key. This is also useful in the case of generated columns.

Some or all columns can be explicitly chosen to be returned and they may optionally be renamed using aliases. Arbitrary non-aggregating expressions may also be returned instead of simply returning a column. All columns can be returned using the `*` expression, and columns or expressions can be returned in addition to all columns returned by the `*`.

```
-- A basic example to show the syntax
CREATE TABLE t1(i INT);
INSERT INTO t1
    SELECT 1
    RETURNING *;
```

i
1

```
-- A more complex example that includes an expression in the RETURNING clause
CREATE TABLE t2(i INT, j INT);
INSERT INTO t2
    SELECT 2 AS i, 3 AS j
    RETURNING *, i * j AS i_times_j;
```

i	j	i_times_j
2	3	6

This example shows a situation where the RETURNING clause is more helpful. First, a table is created with a primary key column. Then a sequence is created to allow for that primary key to be incremented as new rows are inserted. When we insert into the table, we do not already know the values generated by the sequence, so it is valuable to return them. For additional information, see the CREATE SEQUENCE documentation.

```
CREATE TABLE t3(i INT PRIMARY KEY, j INT);
CREATE SEQUENCE 't3_key';
INSERT INTO t3
    SELECT nextval('t3_key') AS i, 42 AS j
    UNION ALL
    SELECT nextval('t3_key') AS i, 43 AS j
    RETURNING *;
```

i	j
1	42
2	43

Pivot Statement

The PIVOT statement allows distinct values within a column to be separated into their own columns. The values within those new columns are calculated using an aggregate function on the subset of rows that match each distinct value.

DuckDB implements both the SQL Standard `PIVOT` syntax and a simplified `PIVOT` syntax that automatically detects the columns to create while pivoting. `PIVOT_WIDER` may also be used in place of the `PIVOT` keyword.

> **Note.** The UNPIVOT statement is the inverse of the `PIVOT` statement.

Simplified Pivot Syntax

The full syntax diagram is below, but the simplified `PIVOT` syntax can be summarized using spreadsheet pivot table naming conventions as:

```
PIVOT [dataset]
ON [column(s)]
USING [value(s)]
GROUP BY [row(s)]
ORDER BY [column(s)-with-order-direction(s)]
LIMIT [number-of-rows];
```

The `ON`, `USING`, and `GROUP BY` clauses are each optional, but they may not all be omitted.

Example Data All examples use the dataset produced by the queries below:

```
CREATE TABLE Cities(Country VARCHAR, Name VARCHAR, Year INT, Population
↪  INT);
INSERT INTO Cities VALUES ('NL', 'Amsterdam', 2000, 1005);
INSERT INTO Cities VALUES ('NL', 'Amsterdam', 2010, 1065);
INSERT INTO Cities VALUES ('NL', 'Amsterdam', 2020, 1158);
INSERT INTO Cities VALUES ('US', 'Seattle', 2000, 564);
INSERT INTO Cities VALUES ('US', 'Seattle', 2010, 608);
INSERT INTO Cities VALUES ('US', 'Seattle', 2020, 738);
INSERT INTO Cities VALUES ('US', 'New York City', 2000, 8015);
INSERT INTO Cities VALUES ('US', 'New York City', 2010, 8175);
INSERT INTO Cities VALUES ('US', 'New York City', 2020, 8772);

FROM Cities;
```

Country	Name	Year	Population
NL	Amsterdam	2000	1005
NL	Amsterdam	2010	1065
NL	Amsterdam	2020	1158

Country	Name	Year	Population
US	Seattle	2000	564
US	Seattle	2010	608
US	Seattle	2020	738
US	New York City	2000	8015
US	New York City	2010	8175
US	New York City	2020	8772

PIVOT ON and USING Use the `PIVOT` statement below to create a separate column for each year and calculate the total population in each. The `ON` clause specifies which column(s) to split into separate columns. It is equivalent to the columns parameter in a spreadsheet pivot table.

The `USING` clause determines how to aggregate the values that are split into separate columns. This is equivalent to the values parameter in a spreadsheet pivot table. If the `USING` clause is not included, it defaults to `COUNT(*)`.

```
PIVOT Cities ON Year USING SUM(Population);
```

Country	Name	2000	2010	2020
NL	Amsterdam	1005	1065	1158
US	Seattle	564	608	738
US	New York City	8015	8175	8772

In the above example, the `SUM` aggregate is always operating on a single value. If we only want to change the orientation of how the data is displayed without aggregating, use the `FIRST` aggregate function. In this example, we are pivoting numeric values, but the `FIRST` function works very well for pivoting out a text column. (This is something that is difficult to do in a spreadsheet pivot table, but easy in DuckDB!)

This query produces a result that is identical to the one above:

```
PIVOT Cities ON Year USING FIRST(Population);
```

PIVOT ON, USING, and GROUP BY By default, the PIVOT statement retains all columns not specified in the ON or USING clauses. To include only certain columns and further aggregate, specify columns in the GROUP BY clause. This is equivalent to the rows parameter of a spreadsheet pivot table.

In the below example, the Name column is no longer included in the output, and the data is aggregated up to the Country level.

```
PIVOT Cities ON Year USING SUM(Population) GROUP BY Country;
```

Country	2000	2010	2020
NL	1005	1065	1158
US	8579	8783	9510

IN Filter for ON Clause To only create a separate column for specific values within a column in the ON clause, use an optional IN expression. Let's say for example that we wanted to forget about the year 2020 for no particular reason...

```
PIVOT Cities ON Year IN (2000, 2010) USING SUM(Population) GROUP BY Country;
```

Country	2000	2010
NL	1005	1065
US	8579	8783

Multiple Expressions per Clause Multiple columns can be specified in the ON and GROUP BY clauses, and multiple aggregate expressions can be included in the USING clause.

Multiple ON Columns and ON Expressions Multiple columns can be pivoted out into their own columns. DuckDB will find the distinct values in each ON clause column and create one new column for all combinations of those values (a cartesian product).

In the below example, all combinations of unique countries and unique cities receive their own column. Some combinations may not be present in the underlying data, so those columns are populated with NULL values.

```
PIVOT Cities ON Country, Name USING SUM(Population);
```

Year	NL_ Amsterdam	NL_New York City	NL_ Seattle	US_ Amsterdam	US_New York City	US_ Seattle
2000	1005	NULL	NULL	NULL	8015	564
2010	1065	NULL	NULL	NULL	8175	608
2020	1158	NULL	NULL	NULL	8772	738

To pivot only the combinations of values that are present in the underlying data, use an expression in the ON clause. Multiple expressions and/or columns may be provided.

Here, Country and Name are concatenated together and the resulting concatenations each receive their own column. Any arbitrary non-aggregating expression may be used. In this case, concatenating with an underscore is used to imitate the naming convention the PIVOT clause uses when multiple ON columns are provided (like in the prior example).

```
PIVOT Cities ON Country || '_' || Name USING SUM(Population);
```

Year	NL_Amsterdam	US_New York City	US_Seattle
2000	1005	8015	564
2010	1065	8175	608
2020	1158	8772	738

Multiple USING Expressions An alias may also be included for each expression in the USING clause. It will be appended to the generated column names after an underscore (_). This makes the column naming convention much cleaner when multiple expressions are included in the USING clause.

In this example, both the SUM and MAX of the Population column are calculated for each year and are split into separate columns.

```
PIVOT Cities ON Year USING SUM(Population) AS total, MAX(Population) AS max
↪  GROUP BY Country;
```

Country	2000_total	2000_max	2010_total	2010_max	2020_total	2020_max
NL	1005	1005	1065	1065	1158	1158
US	8579	8015	8783	8175	9510	8772

Multiple GROUP BY Columns Multiple GROUP BY columns may also be provided. Note that column names must be used rather than column positions (1, 2, etc.), and that expressions are not supported in the GROUP BY clause.

```
PIVOT Cities ON Year USING SUM(Population) GROUP BY Country, Name;
```

Country	Name	2000	2010	2020
NL	Amsterdam	1005	1065	1158
US	Seattle	564	608	738
US	New York City	8015	8175	8772

Using PIVOT within a SELECT Statement The PIVOT statement may be included within a SELECT statement as a CTE (a Common Table Expression, or WITH clause), or a subquery. This allows for a PIVOT to be used alongside other SQL logic, as well as for multiple PIVOTs to be used in one query.

No SELECT is needed within the CTE, the PIVOT keyword can be thought of as taking its place.

```
WITH pivot_alias AS (
    PIVOT Cities ON Year USING SUM(Population) GROUP BY Country
)
SELECT * FROM pivot_alias;
```

A PIVOT may be used in a subquery and must be wrapped in parentheses. Note that this behavior is different than the SQL Standard Pivot, as illustrated in subsequent examples.

```
SELECT
    *
FROM (
    PIVOT Cities ON Year USING SUM(Population) GROUP BY Country
) pivot_alias;
```

Multiple Pivots Each PIVOT can be treated as if it were a SELECT node, so they can be joined together or manipulated in other ways.

For example, if two PIVOT statements share the same GROUP BY expression, they can be joined together using the columns in the GROUP BY clause into a wider pivot.

```
FROM
    (PIVOT Cities ON Year USING SUM(Population) GROUP BY Country) year_pivot
JOIN
```

```
    (PIVOT Cities ON Name USING SUM(Population) GROUP BY Country) name_pivot
USING (Country);
```

Country	2000	2010	2020	Amsterdam	New York City	Seattle
NL	1005	1065	1158	3228	NULL	NULL
US	8579	8783	9510	NULL	24962	1910

Internals

Pivoting is implemented as a combination of SQL query re-writing and a dedicated `PhysicalPivot` operator for higher performance. Each `PIVOT` is implemented as set of aggregations into lists and then the dedicated `PhysicalPivot` operator converts those lists into column names and values. Additional pre-processing steps are required if the columns to be created when pivoting are detected dynamically (which occurs when the `IN` clause is not in use).

DuckDB, like most SQL engines, requires that all column names and types be known at the start of a query. In order to automatically detect the columns that should be created as a result of a `PIVOT` statement, it must be translated into multiple queries. ENUM types are used to find the distinct values that should become columns. Each `ENUM` is then injected into one of the `PIVOT` statement's `IN` clauses.

After the `IN` clauses have been populated with `ENUM`s, the query is re-written again into a set of aggregations into lists.

For example:

```
PIVOT Cities ON Year USING SUM(Population);
```

is initially translated into:

```
CREATE TEMPORARY TYPE __pivot_enum_0_0 AS ENUM (
    SELECT DISTINCT
        Year::VARCHAR
    FROM Cities
    ORDER BY
        Year
    );
PIVOT Cities ON Year IN __pivot_enum_0_0 USING SUM(Population);
```

and finally translated into:

```
SELECT Country, Name, LIST(Year), LIST(population_sum)
FROM (
    SELECT Country, Name, Year, SUM(population) AS population_sum
    FROM Cities
    GROUP BY ALL
)
GROUP BY ALL;
```

This produces the result:

Country	Name	list("YEAR")	list(population_sum)
NL	Amsterdam	[2000, 2010, 2020]	[1005, 1065, 1158]
US	Seattle	[2000, 2010, 2020]	[564, 608, 738]
US	New York City	[2000, 2010, 2020]	[8015, 8175, 8772]

The `PhysicalPivot` operator converts those lists into column names and values to return this result:

Country	Name	2000	2010	2020
NL	Amsterdam	1005	1065	1158
US	Seattle	564	608	738
US	New York City	8015	8175	8772

Simplified Pivot Full Syntax Diagram

Below is the full syntax diagram of the `PIVOT` statement.

SQL Standard Pivot Syntax

The full syntax diagram is below, but the SQL Standard `PIVOT` syntax can be summarized as:

```
FROM [dataset]
PIVOT (
    [values(s)]
    FOR
        [column_1] IN ([in_list])
```

```
        [column_2] IN ([in_list])
        ...
    GROUP BY [rows(s)]
);
```

Unlike the simplified syntax, the `IN` clause must be specified for each column to be pivoted. If you are interested in dynamic pivoting, the simplified syntax is recommended.

Note that no commas separate the expressions in the `FOR` clause, but that `value` and `GROUP BY` expressions must be comma-separated!

Examples

This example uses a single value expression, a single column expression, and a single row expression:

```
FROM Cities
PIVOT (
    SUM(Population)
    FOR
        Year IN (2000, 2010, 2020)
    GROUP BY Country
);
```

Country	2000	2010	2020
NL	1005	1065	1158
US	8579	8783	9510

This example is somewhat contrived, but serves as an example of using multiple value expressions and multiple columns in the `FOR` clause.

```
FROM Cities
PIVOT (
    SUM(Population) AS total,
    COUNT(Population) AS count
    FOR
        Year IN (2000, 2010)
        Country in ('NL', 'US')
);
```

Name	2000_NL_total	2000_NL_count	2000_US_total	2000_US_count	2010_NL_total	2010_NL_count	2010_US_total	2010_US_count
Amsterdam	1005	1	NULL	0	1065	1	NULL	0
Seattle	NULL	0	564	1	NULL	0	608	1
New York City	NULL	0	8015	1	NULL	0	8175	1

SQL Standard Pivot Full Syntax Diagram Below is the full syntax diagram of the SQL Standard version of the PIVOT statement.

Select Statement

The SELECT statement retrieves rows from the database.

Examples

```
-- select all columns from the table "tbl"
SELECT * FROM tbl;
-- select the rows from tbl
SELECT j FROM tbl WHERE i=3;
-- perform an aggregate grouped by the column "i"
SELECT i, SUM(j) FROM tbl GROUP BY i;
-- select only the top 3 rows from the tbl
SELECT * FROM tbl ORDER BY i DESC LIMIT 3;
-- join two tables together using the USING clause
SELECT * FROM t1 JOIN t2 USING(a, b);
-- use column indexes to select the first and third column from the table
 ↪  "tbl"
SELECT #1, #3 FROM tbl;
-- select all unique cities from the addresses table
SELECT DISTINCT city FROM addresses;
```

Syntax The SELECT statement retrieves rows from the database. The canonical order of a select statement is as follows, with less common clauses being indented:

```
SELECT select_list
FROM tables
    USING SAMPLE sample_expr
```

```
WHERE condition
GROUP BY groups
HAVING group_filter
    WINDOW window_expr
    QUALIFY qualify_filter
ORDER BY order_expr
LIMIT n;
```

Optionally, the SELECT statement can be prefixed with a WITH clause.

As the SELECT statement is so complex, we have split up the syntax diagrams into several parts. The full syntax diagram can be found at the bottom of the page.

SELECT Clause

The SELECT clause specifies the list of columns that will be returned by the query. While it appears first in the clause, *logically* the expressions here are executed only at the end. The SELECT clause can contain arbitrary expressions that transform the output, as well as aggregates and window functions. The DISTINCT keyword ensures that only unique tuples are returned.

> **Note.** Column names are case-insensitive. See the Rules for Case Sensitivity for more details.

FROM Clause

The FROM clause specifies the *source* of the data on which the remainder of the query should operate. Logically, the FROM clause is where the query starts execution. The FROM clause can contain a single table, a combination of multiple tables that are joined together, or another SELECT query inside a subquery node.

SAMPLE Clause

The SAMPLE clause allows you to run the query on a sample from the base table. This can significantly speed up processing of queries, at the expense of accuracy in the result. Samples can also be used to quickly see a snapshot of the data when exploring a data set. The sample clause is applied right after anything in the `from` clause (i.e., after any joins, but before the where clause or any aggregates). See the sample page for more information.

WHERE Clause

The WHERE clause specifies any filters to apply to the data. This allows you to select only a subset of the data in which you are interested. Logically the WHERE clause is applied immediately after the FROM clause.

GROUP BY/HAVING Clause

The GROUP BY clause specifies which grouping columns should be used to perform any aggregations in the SELECT clause. If the GROUP BY clause is specified, the query is always an aggregate query, even if no aggregations are present in the SELECT clause.

WINDOW Clause

The WINDOW clause allows you to specify named windows that can be used within window functions. These are useful when you have multiple window functions, as they allow you to avoid repeating the same window clause.

QUALIFY Clause

The QUALIFY clause is used to filter the result of WINDOW functions.

ORDER BY/LIMIT Clause

ORDER BY and LIMIT are output modifiers. Logically they are applied at the very end of the query. The LIMIT clause restricts the amount of rows fetched, and the ORDER BY clause sorts the rows on the sorting criteria in either ascending or descending order.

VALUES List

A VALUES list is a set of values that is supplied instead of a SELECT statement.

Row IDs

For each table, the rowid pseudocolumn returns the row identifiers based on the physical storage.

```
CREATE TABLE t(id int, content string);
INSERT INTO t VALUES (42, 'hello'), (43, 'world');
SELECT rowid, id, content FROM t;
```

rowid	id	content
0	42	hello
1	43	world

In the current storage, these identifiers are contiguous unsigned integers (0, 1, ...) if no rows were deleted. Deletions introduce gaps in the rowids which may be reclaimed later. Therefore, it is strongly recommended *not to use rowids as identifiers*.

> **Note.** The `rowid` values are stable within a transaction.

> **Note.** If there is a user-defined column named `rowid`, it shadows the `rowid` pseudocolumn.

Common Table Expressions

Full Syntax Diagram

Below is the full syntax diagram of the SELECT statement:

Set/Reset

The SET statement modifies the provided DuckDB configuration option at the specified scope.

Examples

```
-- Update the `memory_limit` configuration value.
set memory_limit='10GB';
-- configure the system to use 1 thread
SET threads TO 1;
-- Change configuration option to default value
RESET threads;
```

Syntax

SET updates a DuckDB configuration option to the provided value.

Reset

The RESET statement changes the given DuckDB configuration option to the default value.

Scopes

- local - Not yet implemented.
- session - Configuration value is used (or reset) only for the current session attached to a DuckDB instance.
- global - Configuration value is used (or reset) across the entire DuckDB instance.

When not specified, the default scope for the configuration option is used. For most options this is global.

Configuration

See the Configuration page for the full list of configuration options.

Unpivot Statement

The UNPIVOT statement allows multiple columns to be stacked into fewer columns. In the basic case, multiple columns are stacked into two columns: a NAME column (which contains the name of the source column) and a VALUE column (which contains the value from the source column).

DuckDB implements both the SQL Standard UNPIVOT syntax and a simplified UNPIVOT syntax. Both can utilize a COLUMNS expression to automatically detect the columns to unpivot. PIVOT_LONGER may also be used in place of the UNPIVOT keyword.

> **Note.** The PIVOT statement is the inverse of the UNPIVOT statement.

Simplified UNPIVOT Syntax

The full syntax diagram is below, but the simplified UNPIVOT syntax can be summarized using spreadsheet pivot table naming conventions as:

```
UNPIVOT [dataset]
ON [column(s)]
INTO
    NAME [name-column-name]
    VALUE [value-column-name(s)]
```

```
ORDER BY [column(s)-with-order-direction(s)]
LIMIT [number-of-rows];
```

Example Data All examples use the dataset produced by the queries below:

```
CREATE OR REPLACE TABLE monthly_sales(empid INT, dept TEXT, Jan INT, Feb
 ↪  INT, Mar INT, Apr INT, May INT, Jun INT);
INSERT INTO monthly_sales VALUES
    (1, 'electronics', 1, 2, 3, 4, 5, 6),
    (2, 'clothes', 10, 20, 30, 40, 50, 60),
    (3, 'cars', 100, 200, 300, 400, 500, 600);

FROM monthly_sales;
```

empid	dept	Jan	Feb	Mar	Apr	May	Jun
1	electronics	1	2	3	4	5	6
2	clothes	10	20	30	40	50	60
3	cars	100	200	300	400	500	600

UNPIVOT Manually The most typical `UNPIVOT` transformation is to take already pivoted data and re-stack it into a column each for the name and value. In this case, all months will be stacked into a `month` column and a `sales` column.

```
UNPIVOT monthly_sales
ON jan, feb, mar, apr, may, jun
INTO
    NAME month
    VALUE sales;
```

empid	dept	month	sales
1	electronics	Jan	1
1	electronics	Feb	2
1	electronics	Mar	3
1	electronics	Apr	4
1	electronics	May	5

empid	dept	month	sales
1	electronics	Jun	6
2	clothes	Jan	10
2	clothes	Feb	20
2	clothes	Mar	30
2	clothes	Apr	40
2	clothes	May	50
2	clothes	Jun	60
3	cars	Jan	100
3	cars	Feb	200
3	cars	Mar	300
3	cars	Apr	400
3	cars	May	500
3	cars	Jun	600

UNPIVOT Dynamically using Columns Expression In many cases, the number of columns to unpivot is not easy to predetermine ahead of time. In the case of this dataset, the query above would have to change each time a new month is added. The COLUMNS expression can be used to select all columns that are not `empid` or `dept`. This enables dynamic unpivoting that will work regardless of how many months are added. The query below returns identical results to the one above.

```
UNPIVOT monthly_sales
ON COLUMNS(* EXCLUDE (empid, dept))
INTO
    NAME month
    VALUE sales;
```

empid	dept	month	sales
1	electronics	Jan	1
1	electronics	Feb	2
1	electronics	Mar	3

empid	dept	month	sales
1	electronics	Apr	4
1	electronics	May	5
1	electronics	Jun	6
2	clothes	Jan	10
2	clothes	Feb	20
2	clothes	Mar	30
2	clothes	Apr	40
2	clothes	May	50
2	clothes	Jun	60
3	cars	Jan	100
3	cars	Feb	200
3	cars	Mar	300
3	cars	Apr	400
3	cars	May	500
3	cars	Jun	600

UNPIVOT into multiple value columns The `UNPIVOT` statement has additional flexibility: more than 2 destination columns are supported. This can be useful when the goal is to reduce the extent to which a dataset is pivoted, but not completely stack all pivoted columns. To demonstrate this, the query below will generate a dataset with a separate column for the number of each month within the quarter (month 1, 2, or 3), and a separate row for each quarter. Since there are fewer quarters than months, this does make the dataset longer, but not as long as the above.

To accomplish this, multiple sets of columns are included in the `ON` clause. The `q1` and `q2` aliases are optional. The number of columns in each set of columns in the `ON` clause must match the number of columns in the `VALUE` clause.

```
UNPIVOT monthly_sales
    ON (jan, feb, mar) AS q1, (apr, may, jun) AS q2
    INTO
        NAME quarter
        VALUE month_1_sales, month_2_sales, month_3_sales;
```

empid	dept	quarter	month_1_sales	month_2_sales	month_3_sales
1	electronics	q1	1	2	3
1	electronics	q2	4	5	6
2	clothes	q1	10	20	30
2	clothes	q2	40	50	60
3	cars	q1	100	200	300
3	cars	q2	400	500	600

Using UNPIVOT within a SELECT statement The UNPIVOT statement may be included within a SELECT statement as a CTE (a Common Table Expression, or WITH clause), or a subquery. This allows for an UNPIVOT to be used alongside other SQL logic, as well as for multiple UNPIVOTs to be used in one query.

No SELECT is needed within the CTE, the UNPIVOT keyword can be thought of as taking its place.

```
WITH unpivot_alias AS (
    UNPIVOT monthly_sales
    ON COLUMNS(* EXCLUDE (empid, dept))
    INTO
        NAME month
        VALUE sales
)
SELECT * FROM unpivot_alias;
```

An UNPIVOT may be used in a subquery and must be wrapped in parentheses. Note that this behavior is different than the SQL Standard Unpivot, as illustrated in subsequent examples.

```
SELECT
    *
FROM (
    UNPIVOT monthly_sales
    ON COLUMNS(* EXCLUDE (empid, dept))
    INTO
        NAME month
        VALUE sales
) unpivot_alias;
```

Internals Unpivoting is implemented entirely as rewrites into SQL queries. Each `UNPIVOT` is implemented as set of `UNNEST` functions, operating on a list of the column names and a list of the column values. If dynamically unpivoting, the `COLUMNS` expression is evaluated first to calculate the column list.

For example:

```
UNPIVOT monthly_sales
ON jan, feb, mar, apr, may, jun
INTO
    NAME month
    VALUE sales;
```

is translated into:

```
SELECT
    empid,
    dept,
    UNNEST(['jan', 'feb', 'mar', 'apr', 'may', 'jun']) AS month,
    UNNEST(["jan", "feb", "mar", "apr", "may", "jun"]) AS sales
FROM monthly_sales;
```

Note the single quotes to build a list of text strings to populate `month`, and the double quotes to pull the column values for use in `sales`. This produces the same result as the initial example:

empid	dept	month	sales
1	electronics	jan	1
1	electronics	feb	2
1	electronics	mar	3
1	electronics	apr	4
1	electronics	may	5
1	electronics	jun	6
2	clothes	jan	10
2	clothes	feb	20
2	clothes	mar	30
2	clothes	apr	40
2	clothes	may	50

empid	dept	month	sales
2	clothes	jun	60
3	cars	jan	100
3	cars	feb	200
3	cars	mar	300
3	cars	apr	400
3	cars	may	500
3	cars	jun	600

Simplified Unpivot Full Syntax Diagram Below is the full syntax diagram of the `UNPIVOT` statement.

SQL Standard Unpivot Syntax

The full syntax diagram is below, but the SQL Standard `UNPIVOT` syntax can be summarized as:

```
FROM [dataset]
UNPIVOT [INCLUDE NULLS] (
    [value-column-name(s)]
    FOR [name-column-name] IN [column(s)]
);
```

Note that only one column can be included in the `name-column-name` expression.

SQL Standard Unpivot manually To complete the basic `UNPIVOT` operation using the SQL standard syntax, only a few additions are needed.

```
FROM monthly_sales UNPIVOT (
    sales
    FOR month IN (jan, feb, mar, apr, may, jun)
);
```

empid	dept	month	sales
1	electronics	Jan	1

empid	dept	month	sales
1	electronics	Feb	2
1	electronics	Mar	3
1	electronics	Apr	4
1	electronics	May	5
1	electronics	Jun	6
2	clothes	Jan	10
2	clothes	Feb	20
2	clothes	Mar	30
2	clothes	Apr	40
2	clothes	May	50
2	clothes	Jun	60
3	cars	Jan	100
3	cars	Feb	200
3	cars	Mar	300
3	cars	Apr	400
3	cars	May	500
3	cars	Jun	600

SQL Standard Unpivot Dynamically using Columns Expression The COLUMNS expression can be used to determine the `IN` list of columns dynamically. This will continue to work even if additional `month` columns are added to the dataset. It produces the same result as the query above.

```
FROM monthly_sales UNPIVOT (
    sales
    FOR month IN (columns(* EXCLUDE (empid, dept)))
);
```

SQL Standard UNPIVOT into multiple value columns The `UNPIVOT` statement has additional flexibility: more than 2 destination columns are supported. This can be useful when the goal is to reduce the extent to which a dataset is pivoted, but not completely stack all pivoted columns. To demonstrate

this, the query below will generate a dataset with a separate column for the number of each month within the quarter (month 1, 2, or 3), and a separate row for each quarter. Since there are fewer quarters than months, this does make the dataset longer, but not as long as the above.

To accomplish this, multiple columns are included in the `value-column-name` portion of the `UNPIVOT` statement. Multiple sets of columns are included in the `IN` clause. The `q1` and `q2` aliases are optional. The number of columns in each set of columns in the `IN` clause must match the number of columns in the `value-column-name` portion.

```
FROM monthly_sales
UNPIVOT (
    (month_1_sales, month_2_sales, month_3_sales)
    FOR quarter IN (
        (jan, feb, mar) AS q1,
        (apr, may, jun) AS q2
    )
);
```

empid	dept	quarter	month_1_sales	month_2_sales	month_3_sales
1	electronics	q1	1	2	3
1	electronics	q2	4	5	6
2	clothes	q1	10	20	30
2	clothes	q2	40	50	60
3	cars	q1	100	200	300
3	cars	q2	400	500	600

SQL Standard Unpivot Full Syntax Diagram Below is the full syntax diagram of the SQL Standard version of the `UNPIVOT` statement.

Update Statement

The `UPDATE` statement modifies the values of rows in a table.

Examples

```
-- for every row where "i" is NULL, set the value to 0 instead
UPDATE tbl SET i=0 WHERE i IS NULL;
```

```
-- set all values of "i" to 1 and all values of "j" to 2
UPDATE tbl SET i=1, j = 2;
```

Syntax

UPDATE changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the SET clause; columns not explicitly modified retain their previous values.

Update from Other Table

A table can be updated based upon values from another table. This can be done by specifying a table in a FROM clause, or using a sub-select statement. Both approaches have the benefit of completing the UPDATE operation in bulk for increased performance.

```
CREATE OR REPLACE TABLE original AS
    SELECT 1 AS key, 'original value' AS value
    UNION ALL
    SELECT 2 AS key, 'original value 2' AS value;
CREATE OR REPLACE TABLE new AS
    SELECT 1 AS key, 'new value' AS value
    UNION ALL
    SELECT 2 AS key, 'new value 2' AS value;

SELECT * FROM original;
```

key	value
1	original value
2	original value 2

```
UPDATE original
    SET value = new.value
    FROM new
    WHERE original.key = new.key;
-- OR
UPDATE original
    SET value = (
        SELECT
            new.value
```

```
        FROM new
        WHERE original.key = new.key
    );
SELECT * FROM original;
```

key	value
1	new value
2	new value 2

Update from Same Table

The only difference between this case and the above is that a different table alias must be specified on both the target table and the source table. In this example `as true_original` and `as new` are both required.

```
UPDATE original as true_original
    SET value = (
        SELECT
            new.value || ' a change!' as value
        FROM original as new
        WHERE true_original.key = new.key
    );
```

Upsert (Insert or Update)

See the Insert documentation for details.

Use

The USE statement selects a database and optional schema to use as the default.

Examples

```
--- Sets the 'memory' database as the default
USE memory;
--- Sets the 'duck.main' database and schema as the default
USE duck.main;
```

Syntax

The USE statement sets a default database or database/schema combination to use for future operations. For instance, tables created without providing a fully qualified table name will be created in the default database.

Vacuum

The VACUUM statement alone does nothing and is at present provided for PostgreSQL-compatibility. The VACUUM ANALYZE statement recomputes table statistics if they have become stale due to table updates or deletions.

Examples

```
-- No-op
VACUUM;
-- Rebuild database statistics
VACUUM ANALYZE;
-- Rebuild statistics for the table & column
VACUUM ANALYZE memory.main.my_table(my_column);
-- Not supported
VACUUM FULL; -- error
```

Syntax

Query Syntax

SELECT Clause

The SELECT clause specifies the list of columns that will be returned by the query. While it appears first in the clause, *logically* the expressions here are executed only at the end. The SELECT clause can contain arbitrary expressions that transform the output, as well as aggregates and window functions.

Examples

```
-- select all columns from the table called "table_name"
SELECT * FROM table_name;
```

```
-- perform arithmetic on columns in a table, and provide an alias
SELECT col1 + col2 AS res, sqrt(col1) AS root FROM table_name;
-- select all unique cities from the addresses table
SELECT DISTINCT city FROM addresses;
-- return the total number of rows in the addresses table
SELECT COUNT(*) FROM addresses;
-- select all columns except the city column from the addresses table
SELECT * EXCLUDE (city) FROM addresses;
-- select all columns from the addresses table, but replace city with
 ↪  LOWER(city)
SELECT * REPLACE (LOWER(city) AS city) FROM addresses;
-- select all columns matching the given regex from the table
SELECT COLUMNS('number\d+') FROM addresses;
-- compute a function on all given columns of a table
SELECT MIN(COLUMNS(*)) FROM addresses;
```

Syntax

Select List

The SELECT clause contains a list of expressions that specify the result of a query. The select list can refer to any columns in the FROM clause, and combine them using expressions. As the output of a SQL query is a table - every expression in the SELECT clause also has a name. The expressions can be explicitly named using the AS clause (e.g., `expr AS name`). If a name is not provided by the user the expressions are named automatically by the system.

> **Note.** Column names are case-insensitive. See the Rules for Case Sensitivity for more details.

Star Expressions

```
-- select all columns from the table called "table_name"
SELECT * FROM table_name;
-- select all columns matching the given regex from the table
SELECT COLUMNS('number\d+') FROM addresses;
```

The star expression is a special expression that expands to *multiple expressions* based on the contents of the FROM clause. In the simplest case, * expands to **all** expressions in the FROM clause. Columns can also be selected using regular expressions or lambda functions. See the star expression page for more details.

Distinct Clause

```
-- select all unique cities from the addresses table
SELECT DISTINCT city FROM addresses;
```

The DISTINCT clause can be used to return **only** the unique rows in the result - so that any duplicate rows are filtered out.

> **Note.** Queries starting with SELECT DISTINCT run deduplication, which is an expensive operation. Therefore, only use DISTINCT if necessary.

Distinct On Clause

```
-- select only the highest population city for each country
SELECT DISTINCT ON(country) city, population FROM cities ORDER BY population
↪ DESC;
```

The DISTINCT ON clause returns only one row per unique value in the set of expressions as defined in the ON clause. If an ORDER BY clause is present, the row that is returned is the first row that is encountered *as per the ORDER BY* criteria. If an ORDER BY clause is not present, the first row that is encountered is not defined and can be any row in the table.

> **Note.** When querying large data sets, using DISTINCT on all columns can be expensive. Therefore, consider using DISTINCT ON on a column (or a set of columns) which guaranetees a sufficient degree of uniqueness for your results. For example, using DISTINCT ON on the key column(s) of a table guarantees full uniqueness.

Aggregates

```
-- return the total number of rows in the addresses table
SELECT COUNT(*) FROM addresses;
-- return the total number of rows in the addresses table grouped by city
SELECT city, COUNT(*) FROM addresses GROUP BY city;
```

Aggregate functions are special functions that *combine* multiple rows into a single value. When aggregate functions are present in the SELECT clause, the query is turned into an aggregate query. In an aggregate query, **all** expressions must either be part of an aggregate function, or part of a group (as specified by the GROUP BY clause).

Window Functions

```
-- generate a "row_number" column containing incremental identifiers for
↪ each row
SELECT row_number() OVER () FROM sales;
```

```
-- compute the difference between the current amount, and the previous
 ↪  amount, by order of time
SELECT amount - lag(amount) OVER (ORDER BY time) FROM sales;
```

Window functions are special functions that allow the computation of values relative to *other rows* in a result. Window functions are marked by the OVER clause which contains the *window specification*. The window specification defines the frame or context in which the window function is computed. See the window functions page for more information.

Unnest

```
-- unnest an array by one level
SELECT UNNEST([1, 2, 3]);
-- unnest a struct by one level
SELECT UNNEST({'a': 42, 'b': 84});
```

Unnest is a special function that can be used together with arrays or structs. The unnest function strips one level of nesting from the type. For example, `INT[]` is transformed into `INT`. `STRUCT(a INT, b INT)` is transformed into `a INT, b INT`. The unnest function can be used to transform nested types into regular scalar types, which makes them easier to operate on.

FROM & JOIN Clauses

The FROM clause specifies the *source* of the data on which the remainder of the query should operate. Logically, the FROM clause is where the query starts execution. The FROM clause can contain a single table, a combination of multiple tables that are joined together using JOIN clauses, or another SELECT query inside a subquery node. DuckDB also has an optional FROM-first syntax which enables you to also query without a SELECT statement.

Examples

```
-- select all columns from the table called "table_name"
SELECT * FROM table_name;
-- select all columns from the table called "table_name" using the
 ↪  FROM-first syntax
FROM table_name SELECT *;
-- select all columns using the FROM-first syntax and omitting the SELECT
 ↪  clause
FROM table_name;
-- select all columns from the table called "table_name" in the schema
 ↪  "schema_name
```

```
SELECT * FROM schema_name.table_name;
-- select the column "i" from the table function "range", where the first
 ↪  column of the range function is renamed to "i"
SELECT t.i FROM range(100) AS t(i);
-- select all columns from the CSV file called "test.csv"
SELECT * FROM 'test.csv';
-- select all columns from a subquery
SELECT * FROM (SELECT * FROM table_name);
-- select the entire row of the table as a struct
SELECT t FROM t;
-- select the entire row of the subquery as a struct (i.e., a single column)
SELECT t FROM (SELECT unnest(generate_series(41, 43)) AS x, 'hello' AS y) t;
-- join two tables together
SELECT * FROM table_name JOIN other_table ON (table_name.key = other_
 ↪  table.key);
-- select a 10% sample from a table
SELECT * FROM table_name TABLESAMPLE 10%;
-- select a sample of 10 rows from a table
SELECT * FROM table_name TABLESAMPLE 10 ROWS;
-- use the FROM-first syntax with WHERE clause and aggregation
FROM range(100) AS t(i) SELECT sum(t.i) WHERE i % 2 = 0;
```

Joins

Joins are a fundamental relational operation used to connect two tables or relations horizontally. The relations are referred to as the *left* and *right* sides of the join based on how they are written in the join clause. Each result row has the columns from both relations.

A join uses a rule to match pairs of rows from each relation. Often this is a predicate, but there are other implied rules that may be specified.

Outer Joins Rows that do not have any matches can still be returned if an OUTER join is specified. Outer joins can be one of:

- LEFT (All rows from the left relation appear at least once)
- RIGHT (All rows from the right relation appear at least once)
- FULL (All rows from both relations appear at least once)

A join that is not OUTER is INNER (only rows that get paired are returned).

When an unpaired row is returned, the attributes from the other table are set to NULL.

Cross Product Joins The simplest type of join is a `CROSS JOIN`. There are no conditions for this type of join, and it just returns all the possible pairs.

```
-- return all pairs of rows
SELECT a.*, b.* FROM a CROSS JOIN b;
```

Conditional Joins Most joins are specified by a predicate that connects attributes from one side to attributes from the other side. The conditions can be explicitly specified using an `ON` clause with the join (clearer) or implied by the `WHERE` clause (old-fashioned).

We use the `l_regions` and the `l_nations` tables from the TPC-H schema:

```
CREATE TABLE l_regions(r_regionkey INTEGER NOT NULL PRIMARY KEY,
                       r_name      CHAR(25) NOT NULL,
                       r_comment   VARCHAR(152));

CREATE TABLE l_nations (n_nationkey INTEGER NOT NULL PRIMARY KEY,
                        n_name      CHAR(25) NOT NULL,
                        n_regionkey INTEGER NOT NULL,
                        n_comment   VARCHAR(152),
                        FOREIGN KEY (n_regionkey) REFERENCES l_regions(r_
                         ↪  regionkey));

-- return the regions for the nations
SELECT n.*, r.*
FROM l_nations n JOIN l_regions r ON (n_regionkey = r_regionkey);
```

If the column names are the same and are required to be equal, then the simpler `USING` syntax can be used:

```
CREATE TABLE l_regions(regionkey INTEGER NOT NULL PRIMARY KEY,
                       name      CHAR(25) NOT NULL,
                       comment   VARCHAR(152));

CREATE TABLE l_nations (nationkey INTEGER NOT NULL PRIMARY KEY,
                        name      CHAR(25) NOT NULL,
                        regionkey INTEGER NOT NULL,
                        comment   VARCHAR(152),
                        FOREIGN KEY (regionkey) REFERENCES l_
                         ↪  regions(regionkey));

-- return the regions for the nations
SELECT n.*, r.*
FROM l_nations n JOIN l_regions r USING (regionkey);
```

The expressions to not have to be equalities - any predicate can be used:

```
-- return the pairs of jobs where one ran longer but cost less
SELECT s1.t_id, s2.t_id
FROM west s1, west s2
WHERE s1.time > s2.time
  AND s1.cost < s2.cost;
```

Semi and Anti Joins Semi joins return rows from the left table that have at least one match in the right table. Anti joins return rows from the left table that have *no* matches in the right table. When using a semi or anti join the result will never have more rows than the left hand side table. Semi and anti joins provide the same logic as (NOT) IN statements.

```
-- return a list of cars that have a valid region.
SELECT cars.name, cars.manufacturer
FROM cars SEMI JOIN region
ON cars.region = region.id;

-- return a list of cars with no recorded safety data.
SELECT cars.name, cars.manufacturer
FROM cars ANTI JOIN safety_data
ON cars.safety_report_id = safety_data.report_id;
```

Lateral Joins The LATERAL keyword allows subqueries in the FROM clause to refer to previous subqueries. This feature is also known as a *lateral join*.

```
SELECT *
FROM range(3) t(i), LATERAL (SELECT i + 1) t2(j);
```

i	j
int64	int64
0	1
1	2
2	3

Lateral joins are a generalization of correlated subqueries, as they can return multiple values per input value rather than only a single value.

```
SELECT *
FROM generate_series(0, 1) t(i), LATERAL (SELECT i + 10 UNION ALL SELECT i +
 ↪  100) t2(j);
```

i	j
int64	int64
0	10
1	11
0	100
1	101

It may be helpful to think about LATERAL as a loop where we iterate through the rows of the first subquery and use it as input to the second (LATERAL) subquery. In the examples above, we iterate through table `t` and refer to its column `i` from the definition of table `t2`. The rows of `t2` form column `j` in the result.

It is possible to refer to multiple attributes from the LATERAL subquery. Using the table from the first example:

```
CREATE TABLE t1 AS SELECT * FROM range(3) t(i), LATERAL (SELECT i + 1) t2(j);
SELECT * FROM t1, LATERAL (SELECT i + j) t2(k);
```

i	j	k
int64	int64	int64
0	1	1
1	2	3
2	3	5

> **Note.** DuckDB detects when LATERAL joins should be used, making the use of the LATERAL keyword optional.

Positional Joins When working with data frames or other embedded tables of the same size, the rows may have a natural correspondence based on their physical order. In scripting languages, this is easily expressed using a loop:

```
for (i=0;i<n;i++)
    f(t1.a[i], t2.b[i])
```

It is difficult to express this in standard SQL because relational tables are not ordered, but imported tables (like data frames) or disk files (like CSVs or Parquet files) do have a natural ordering.

Connecting them using this ordering is called a *positional join*:

```
-- treat two data frames as a single table
SELECT df1.*, df2.*
FROM df1 POSITIONAL JOIN df2;
```

Positional joins are always `FULL OUTER` joins.

As-Of Joins A common operation when working with temporal or similarly-ordered data is to find the nearest (first) event in a reference table (such as prices). This is called an *as-of join*:

```
-- attach prices to stock trades
SELECT t.*, p.price
FROM trades t ASOF JOIN prices p
  ON t.symbol = p.symbol AND t.when >= p.when;
```

The `ASOF` join requires at least one inequality condition on the ordering field. The inequality can be any inequality condition (`>=`, `>`, `<=`, `<`) on any data type, but the most common form is `>=` on a temporal type. Any other conditions must be equalities (or `NOT DISTINCT`). This means that the left/right order of the tables is significant.

`ASOF` joins each left side row with at most one right side row. It can be specified as an `OUTER` join to find unpaired rows (e.g., trades without prices or prices which have no trades.)

```
-- attach prices or NULLs to stock trades
SELECT *
FROM trades t ASOF LEFT JOIN prices p
  ON t.symbol = p.symbol AND t.when >= p.when;
```

`ASOF` joins can also specify join conditions on matching column names with the `USING` syntax, but the *last* attribute in the list must be the inequality, which will be greater than or equal to (`>=`):

```
SELECT *
FROM trades t ASOF JOIN prices p USING (symbol, when);
-- Returns symbol, trades.when, price (but NOT prices.when)
```

If you combine `USING` with a `SELECT *` like this, the query will return the left side (probe) column values for the matches, not the right side (build) column values. To get the `prices` times in the example, you will need to list the columns explicitly:

```
SELECT t.symbol, t.when AS trade_when, p.when AS price_when, price
FROM trades t ASOF LEFT JOIN prices p USING (symbol, when);
```

Syntax

WHERE Clause

The WHERE clause specifies any filters to apply to the data. This allows you to select only a subset of the data in which you are interested. Logically the WHERE clause is applied immediately after the FROM clause.

Examples

```
-- select all rows that have id equal to 3
SELECT *
FROM table_name
WHERE id=3;
-- select all rows that match the given case-insensitive LIKE expression
SELECT *
FROM table_name
WHERE name ILIKE '%mark%';
-- select all rows that match the given composite expression
SELECT *
FROM table_name
WHERE id=3 OR id=7;
```

Syntax

GROUP BY Clause

The GROUP BY clause specifies which grouping columns should be used to perform any aggregations in the SELECT clause. If the GROUP BY clause is specified, the query is always an aggregate query, even if no aggregations are present in the SELECT clause.

When a GROUP BY clause is specified, all tuples that have matching data in the grouping columns (i.e., all tuples that belong to the same group) will be combined. The values of the grouping columns themselves are unchanged, and any other columns can be combined using an aggregate function (such as COUNT, SUM, AVG, etc).

GROUP BY ALL

Use GROUP BY ALL to GROUP BY all columns in the SELECT statement that are not wrapped in aggregate functions. This simplifies the syntax by allowing the columns list to be maintained in a single

location, and prevents bugs by keeping the SELECT granularity aligned to the GROUP BY granularity (Ex: Prevents any duplication). See examples below and additional examples in the Friendlier SQL with DuckDB blog post.

Multiple Dimensions

Normally, the GROUP BY clause groups along a single dimension. Using the GROUPING SETS, CUBE or ROLLUP clauses it is possible to group along multiple dimensions. See the GROUPING SETS page for more information.

Examples

```
-- count the number of entries in the "addresses" table that belong to each
 ↪ different city
SELECT city, COUNT(*)
FROM addresses
GROUP BY city;

-- compute the average income per city per street_name
SELECT city, street_name, AVG(income)
FROM addresses
GROUP BY city, street_name;
```

GROUP BY ALL Examples

```
-- Group by city and street_name to remove any duplicate values
SELECT city, street_name
FROM addresses
GROUP BY ALL;
-- GROUP BY city, street_name
;

-- compute the average income per city per street_name
-- Since income is wrapped in an aggregate function, do not include it in
 ↪ the GROUP BY
SELECT city, street_name, AVG(income)
FROM addresses
GROUP BY ALL;
-- GROUP BY city, street_name
;
```

Syntax

GROUPING SETS

GROUPING SETS, ROLLUP and CUBE can be used in the GROUP BY clause to perform a grouping over multiple dimensions within the same query. Note that this syntax is not compatible with GROUP BY ALL.

Examples

```
-- compute the average income along the provided four different dimensions
-- () signifies the empty set (i.e., computing an ungrouped aggregate)
SELECT city, street_name, AVG(income)
FROM addresses
GROUP BY GROUPING SETS ((city, street_name), (city), (street_name), ());
-- compute the average income along the same dimensions
SELECT city, street_name, AVG(income)
FROM addresses
GROUP BY CUBE (city, street_name);
-- compute the average income along the dimensions (city, street_name),
 ↪  (city) and ()
SELECT city, street_name, AVG(income)
FROM addresses
GROUP BY ROLLUP (city, street_name);
```

Description

GROUPING SETS perform the same aggregate across different GROUP BY clauses in a single query.

```
CREATE TABLE students (course VARCHAR, type VARCHAR);
INSERT INTO students (course, type) VALUES ('CS', 'Bachelor'), ('CS',
 ↪  'Bachelor'), ('CS', 'PhD'), ('Math', 'Masters'), ('CS', NULL), ('CS',
 ↪  NULL), ('Math', NULL);
```

```
SELECT course, type, COUNT(*)
FROM students
GROUP BY GROUPING SETS ((course, type), course, type, ());
```

course	type	count_star()

```
┌─────────┬──────────┬──────────────┐
│ CS      │ Bachelor │ 2            │
│ CS      │ PhD      │ 1            │
│ Math    │ Masters  │ 1            │
│ CS      │ NULL     │ 2            │
│ Math    │ NULL     │ 1            │
│ CS      │ NULL     │ 5            │
│ Math    │ NULL     │ 2            │
│ NULL    │ Bachelor │ 2            │
│ NULL    │ PhD      │ 1            │
│ NULL    │ Masters  │ 1            │
│ NULL    │ NULL     │ 3            │
│ NULL    │ NULL     │ 7            │
└─────────┴──────────┴──────────────┘
```

In the above query, we group across four different sets: `course, type`, `course`, `type` and `()` (the empty group). The result contains `NULL` for a group which is not in the grouping set for the result, i.e., the above query is equivalent to the following UNION statement:

```
-- group by course, type
SELECT course, type, COUNT(*)
FROM students
GROUP BY course, type
UNION ALL
-- group by type
SELECT NULL AS course, type, COUNT(*)
FROM students
GROUP BY type
UNION ALL
-- group by course
SELECT course, NULL AS type, COUNT(*)
FROM students
GROUP BY course
UNION ALL
-- group by nothing
SELECT NULL AS course, NULL AS type, COUNT(*)
FROM students;
```

CUBE and ROLLUP are syntactic sugar to easily produce commonly used grouping sets.

The ROLLUP clause will produce all "sub-groups" of a grouping set, e.g., `ROLLUP (country, city, zip)` produces the grouping sets `(country, city, zip)`, `(country, city)`, `(country)`, `()`. This can be useful for producing different levels of detail of a group by clause. This produces n+1 grouping sets where n is the amount of terms in the ROLLUP clause.

CUBE produces grouping sets for all combinations of the inputs, e.g., `CUBE (country, city, zip)` will produce `(country, city, zip)`, `(country, city)`, `(country, zip)`, `(city, zip)`, `(country)`, `(city)`, `(zip)`, `()`. This produces `2^n` grouping sets.

GROUPING (alias GROUPING_ID) is a special aggregate function that can be used in combination with grouping sets. The GROUPING function takes as parameters a group, and returns 0 if the group is included in the grouping for that row, or 1 otherwise. This is primarily useful because the grouping columns by which we do not aggregate return NULL, which is ambiguous with groups that are actually the value NULL. The GROUPING (or GROUPING_ID) function can be used to distinguish these two cases.

Syntax

HAVING Clause

The HAVING clause can be used after the GROUP BY clause to provide filter criteria *after* the grouping has been completed. In terms of syntax the HAVING clause is identical to the WHERE clause, but while the WHERE clause occurs before the grouping, the HAVING clause occurs after the grouping.

Examples

```
-- count the number of entries in the "addresses" table that belong to each
 ↪  different city
-- filtering out cities with a count below 50
SELECT city, COUNT(*)
FROM addresses
GROUP BY city
HAVING COUNT(*) >= 50;
-- compute the average income per city per street_name
-- filtering out cities with an average income bigger than twice the median
 ↪  income
SELECT city, street_name, AVG(income)
FROM addresses
GROUP BY city, street_name
HAVING AVG(income) > 2 * MEDIAN(income);
```

Syntax

ORDER BY Clause

`ORDER BY` is an output modifier. Logically it is applied near the very end of the query (just prior to `LIMIT` or `OFFSET`, if present). The `ORDER BY` clause sorts the rows on the sorting criteria in either ascending or descending order. In addition, every order clause can specify whether `NULL` values should be moved to the beginning or to the end.

The `ORDER BY` clause may contain one or more expressions, separated by commas. An error will be thrown if no expressions are included, since the `ORDER BY` clause should be removed in that situation. The expressions may begin with either an arbitrary scalar expression (which could be a column name), a column position number (Ex: `1`. Note that it is 1-indexed), or the keyword `ALL`. Each expression can optionally be followed by an order modifier (`ASC` or `DESC`, default is `ASC`), and/or a `NULL` order modifier (`NULLS FIRST` or `NULLS LAST`, default is `NULLS LAST`).

ORDER BY ALL

The `ALL` keyword indicates that the output should be sorted by every column in order from left to right. The direction of this sort may be modified using either `ORDER BY ALL ASC` or `ORDER BY ALL DESC` and/or `NULLS FIRST` or `NULLS LAST`. Note that `ALL` may not be used in combination with other expressions in the `ORDER BY` clause - it must be by itself. See examples below.

NULL Order Modifier

By default if no modifiers are provided, DuckDB sorts `ASC NULLS LAST`, i.e., the values are sorted in ascending order and null values are placed last. This is identical to the default sort order of PostgreSQL. The default sort order can be changed using the following `PRAGMA` statements.

> **Note.** Using `ASC NULLS LAST` as default the default sorting order was a breaking change in version 0.8.0. Prior to 0.8.0, DuckDB sorted using `ASC NULLS FIRST`.

```
-- change the default null sorting order to either NULLS FIRST and NULLS LAST
PRAGMA default_null_order='NULLS FIRST';
-- change the default sorting order to either DESC or ASC
PRAGMA default_order='DESC';
```

Collations

Text is sorted using the binary comparison collation by default, which means values are sorted on their binary UTF8 values. While this works well for ASCII text (e.g., for English language data), the sorting order can be incorrect for other languages. For this purpose, DuckDB provides collations. For more information on collations, see the Collation page.

Examples

All examples use this example table:

```
CREATE OR REPLACE TABLE addresses AS
    SELECT '123 Quack Blvd' AS address, 'DuckTown' AS city, '11111' AS zip
    UNION ALL
    SELECT '111 Duck Duck Goose Ln', 'DuckTown', '11111'
    UNION ALL
    SELECT '111 Duck Duck Goose Ln', 'Duck Town', '11111'
    UNION ALL
    SELECT '111 Duck Duck Goose Ln', 'Duck Town', '11111-0001';

-- select the addresses, ordered by city name using the default null order
 ↪  and default order
SELECT *
FROM addresses
ORDER BY city;
-- select the addresses, ordered by city name in descending order with nulls
 ↪  at the end
SELECT *
FROM addresses
ORDER BY city DESC NULLS LAST;
-- order by city and then by zip code, both using the default orderings
SELECT *
FROM addresses
ORDER BY city, zip;
-- order by city using german collation rules
SELECT *
FROM addresses
ORDER BY city COLLATE DE;
```

ORDER BY ALL Examples

```
-- Order from left to right (by address, then by city, then by zip) in
 ↪  ascending order
SELECT *
FROM addresses
ORDER BY ALL;
```

address	city	zip
111 Duck Duck Goose Ln	Duck Town	11111
111 Duck Duck Goose Ln	Duck Town	11111-0001
111 Duck Duck Goose Ln	DuckTown	11111
123 Quack Blvd	DuckTown	11111

```
-- Order from left to right (by address, then by city, then by zip) in
 ↪  descending order
SELECT *
FROM addresses
ORDER BY ALL DESC;
```

address	city	zip
123 Quack Blvd	DuckTown	11111
111 Duck Duck Goose Ln	DuckTown	11111
111 Duck Duck Goose Ln	Duck Town	11111-0001
111 Duck Duck Goose Ln	Duck Town	11111

Syntax

LIMIT Clause

`LIMIT` is an output modifier. Logically it is applied at the very end of the query. The `LIMIT` clause restricts the amount of rows fetched. The `OFFSET` clause indicates at which position to start reading the values, i.e., the first `OFFSET` values are ignored.

Note that while `LIMIT` can be used without an `ORDER BY` clause, the results might not be deterministic without the `ORDER BY` clause. This can still be useful, however, for example when you want to inspect a quick snapshot of the data.

Examples

```
-- select the first 5 rows from the addresses table
SELECT *
FROM addresses
LIMIT 5;
-- select the 5 rows from the addresses table, starting at position 5 (i.e.,
 ↪  ignoring the first 5 rows)
SELECT *
FROM addresses
LIMIT 5
OFFSET 5;
-- select the top 5 cities with the highest population
SELECT city, COUNT(*) AS population
FROM addresses
GROUP BY city
ORDER BY population DESC
LIMIT 5;
```

Syntax

SAMPLE Clause

The `SAMPLE clause` allows you to run the query on a sample from the base table. This can significantly speed up processing of queries, at the expense of accuracy in the result. Samples can also be used to quickly see a snapshot of the data when exploring a data set. The sample clause is applied right after anything in the FROM clause (i.e., after any joins, but before the where clause or any aggregates). See the sample page for more information.

Examples

```
-- select a sample of 1% of the addresses table using default (system)
 ↪  sampling
SELECT *
FROM addresses
USING SAMPLE 1%;
-- select a sample of 1% of the addresses table using bernoulli sampling
SELECT *
FROM addresses
USING SAMPLE 1% (BERNOULLI);
-- select a sample of 10 rows from the subquery
```

```sql
SELECT *
FROM (SELECT * FROM addresses)
USING SAMPLE 10 ROWS;
```

Syntax

UNNEST

Examples

```sql
-- unnest a list, generating 3 rows (1, 2, 3)
SELECT UNNEST([1, 2, 3]);
-- unnesting a struct, generating two columns (a, b)
SELECT UNNEST({'a': 42, 'b': 84});
-- recursive unnest of a list of structs
SELECT UNNEST([{'a': 42, 'b': 84}, {'a': 100, 'b': NULL}], recursive :=
 ↪  true);
```

The UNNEST function is used to unnest lists or structs by one level. The function can be used as a regular scalar function, but only in the SELECT clause. UNNEST with the `recursive` parameter will unnest lists and structs of multiple levels.

Unnesting Lists

```sql
-- unnest a list, generating 3 rows (1, 2, 3)
SELECT UNNEST([1, 2, 3]);
-- unnest a scalar list, generating 3 rows ((1, 10), (2, 11), (3, NULL))
SELECT UNNEST([1, 2, 3]), UNNEST([10, 11]);
-- unnest a scalar list, generating 3 rows ((1, 10), (2, 10), (3, 10))
SELECT UNNEST([1, 2, 3]), 10;
-- unnest a list column generated from a subquery
SELECT UNNEST(l) + 10 FROM (VALUES ([1, 2, 3]), ([4, 5])) tbl(l);
-- empty result
SELECT UNNEST([]);
-- empty result
SELECT UNNEST(NULL);
```

UNNEST on a list will emit one tuple per entry in the list. When UNNEST is combined with regular scalar expressions, those expressions are repeated for every entry in the list. When multiple lists are unnested in the same SELECT clause, the lists are unnested side-by-side. If one list is longer than the other, the shorter list will be padded with NULL values.

An empty list and a NULL list will both unnest to zero elements.

Unnesting Structs

```
-- unnesting a struct, generating two columns (a, b)
SELECT UNNEST({'a': 42, 'b': 84});
-- unnesting a struct, generating two columns (a, b)
SELECT UNNEST({'a': 42, 'b': {'x': 84}});
```

UNNEST on a struct will emit one column per entry in the struct.

Recursive Unnest

```
-- unnesting a list of lists recursively, generating 5 rows (1, 2, 3, 4, 5)
SELECT UNNEST([[1, 2, 3], [4, 5]], recursive := true);
-- unnesting a list of structs recursively, generating two rows of two
 ↪  columns (a, b)
SELECT UNNEST([{'a': 42, 'b': 84}, {'a': 100, 'b': NULL}], recursive :=
 ↪  true);
-- unnesting a struct, generating two columns (a, b)
SELECT UNNEST({'a': [1, 2, 3], 'b': 88}, recursive := true);
```

Calling UNNEST with the `recursive` setting will fully unnest lists, followed by fully unnesting structs. This can be useful to fully flatten columns that contain lists within lists, or lists of structs. Note that lists *within* structs are not unnested.

WITH Clause

The `WITH` clause allows you to specify common table expressions (CTEs). Regular (non-recursive) common-table-expressions are essentially views that are limited in scope to a particular query. CTEs can reference each-other and can be nested.

Basic CTE Examples

```
-- create a CTE called "cte" and use it in the main query
WITH cte AS (SELECT 42 AS x)
SELECT * FROM cte;
```

x
42

```
-- create two CTEs, where the second CTE references the first CTE
WITH cte AS (SELECT 42 AS i),
     cte2 AS (SELECT i*100 AS x FROM cte)
SELECT * FROM cte2;
```

x
4200

Materialized CTEs

By default, CTEs are inlined into the main query. Inlining can result in duplicate work, because the definition is copied for each reference. Take this query for example:

```
WITH t(x) AS (⌧Q_t⌧)
SELECT * FROM t AS t1,
              t AS t2,
              t AS t3;
```

Inlining duplicates the definition of `t` for each reference which results in the following query:

```
SELECT * FROM (⌧Q_t⌧) AS t1(x),
              (⌧Q_t⌧) AS t2(x),
              (⌧Q_t⌧) AS t3(x);
```

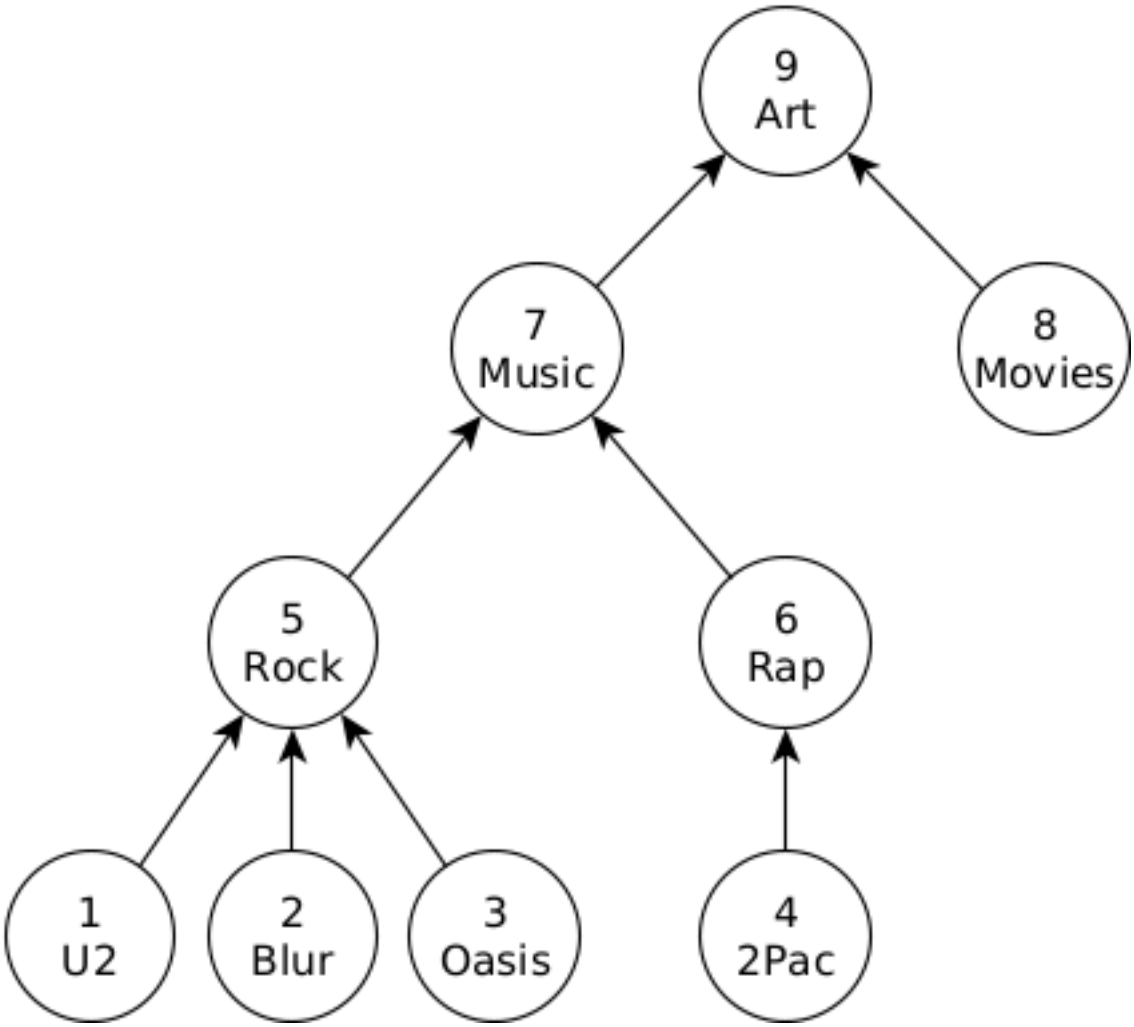
If ⌧Q_t⌧ is expensive, materializing it with the `MATERIALIZED` keyword can improve performance. In this case, ⌧Q_t⌧ is evaluated only once.

```
WITH t(x) AS MATERIALIZED (⌧Q_t⌧)
SELECT * FROM t AS t1,
              t AS t2,
              t AS t3;
```

Recursive CTEs

`WITH RECURSIVE` allows the definition of CTEs which can refer to themselves. Note that the query must be formulated in a way that ensures termination, otherwise, it may run into an infinite loop.

Tree Traversal `WITH RECURSIVE` can be used to traverse trees. For example, take a hierarchy of tags:




```
CREATE TABLE tag(id int, name varchar, subclassof int);
INSERT INTO tag VALUES
 (1, 'U2',     5),
 (2, 'Blur',   5),
 (3, 'Oasis',  5),
 (4, '2Pac',   6),
 (5, 'Rock',   7),
 (6, 'Rap',    7),
 (7, 'Music',  9),
 (8, 'Movies', 9),
 (9, 'Art', NULL);
```

The following query returns the path from the node `Oasis` to the root of the tree (`Art`).

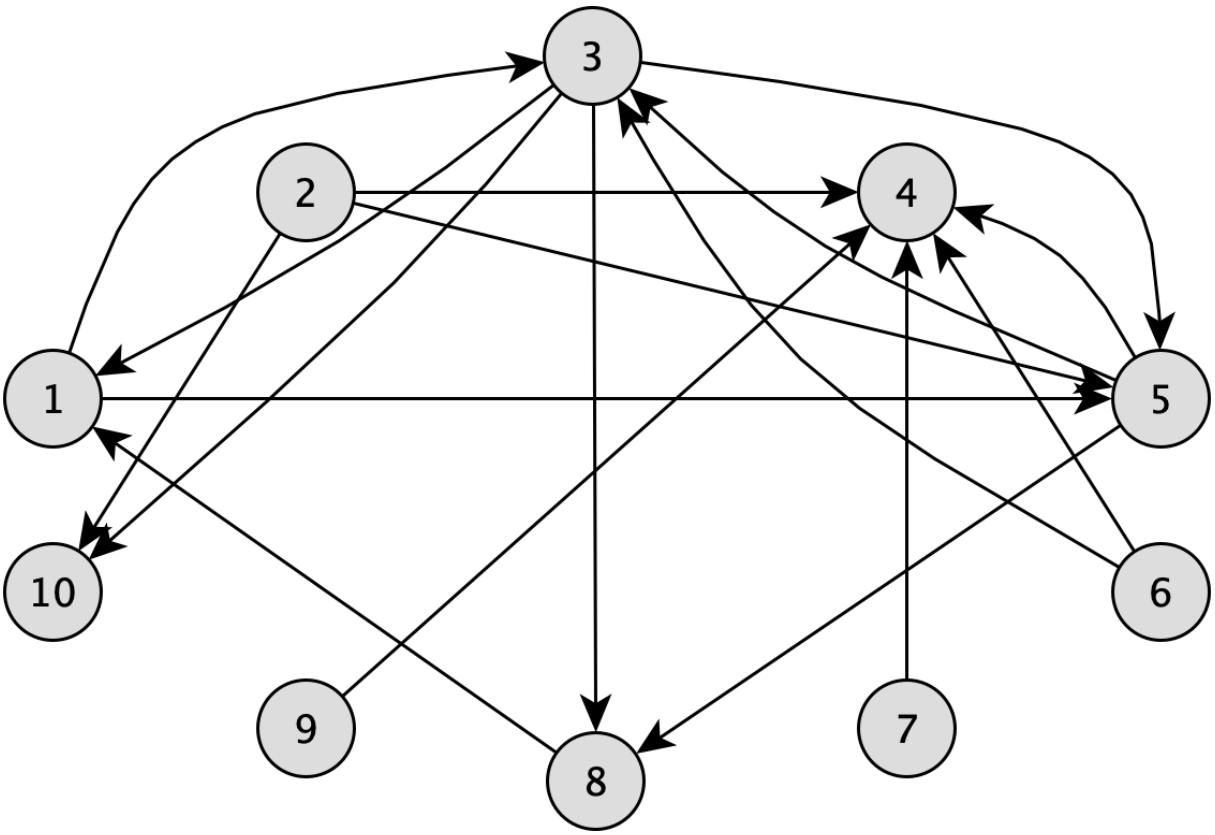
```
WITH RECURSIVE tag_hierarchy(id, source, path) AS (
  SELECT id, name, [name] AS path
```

```
  FROM tag
  WHERE subclassof IS NULL
UNION ALL
  SELECT tag.id, tag.name, list_prepend(tag.name, tag_hierarchy.path)
  FROM tag, tag_hierarchy
  WHERE tag.subclassof = tag_hierarchy.id
)
SELECT path
FROM tag_hierarchy
WHERE source = 'Oasis';
```

```
┌──────────────────────────┐
│           path           │
├──────────────────────────┤
│ [Oasis, Rock, Music, Art] │
└──────────────────────────┘
```

Graph Traversal The `WITH RECURSIVE` clause can be used to express graph traversal on arbitrary graphs. However, if the graph has cycles, the query must perform cycle detection to prevent infinite loops. One way to achieve this is to store the path of a traversal in a list and, before extending the path with a new edge, check whether its endpoint has been visited before (see the example later).

Take the following directed graph from the LDBC Graphalytics benchmark:




```
CREATE TABLE edge(node1id int, node2id int);
INSERT INTO edge VALUES (1, 3), (1, 5), (2, 4), (2, 5), (2, 10), (3, 1), (3,
↪  5),
  (3, 8), (3, 10), (5, 3), (5, 4), (5, 8), (6, 3), (6, 4), (7, 4), (8, 1),
↪  (9, 4);
```

Note that the graph contains directed cycles, e.g., between nodes 1, 2, and 5.

Enumerate All Paths from a Node The following query returns **all paths** starting in node 1:

```
WITH RECURSIVE paths(startNode, endNode, path) AS (
   SELECT -- define the path as the first edge of the traversal
        node1id AS startNode,
        node2id AS endNode,
        [node1id, node2id] AS path
     FROM edge
     WHERE startNode = 1
   UNION ALL
   SELECT -- concatenate new edge to the path
        paths.startNode AS startNode,
        node2id AS endNode,
```

```
        array_append(path, node2id) AS path
     FROM paths
     JOIN edge ON paths.endNode = node1id
    -- Prevent adding a repeated node to the path.
    -- This ensures that no cycles occur.
    WHERE node2id != ALL(paths.path)
)
SELECT startNode, endNode, path
FROM paths
ORDER BY length(path), path;
```

startNode	endNode	path
1	3	[1, 3]
1	5	[1, 5]
1	5	[1, 3, 5]
1	8	[1, 3, 8]
1	10	[1, 3, 10]
1	3	[1, 5, 3]
1	4	[1, 5, 4]
1	8	[1, 5, 8]
1	4	[1, 3, 5, 4]
1	8	[1, 3, 5, 8]
1	8	[1, 5, 3, 8]
1	10	[1, 5, 3, 10]

Note that the result of this query is not restricted to shortest paths, e.g., for node 5, the results include paths `[1, 5]` and `[1, 3, 5]`.

Enumerate Unweighted Shortest Paths from a Node In most cases, enumerating all paths is not practical or feasible. Instead, only the **(unweighted) shortest paths** are of interest. To find these, the second half of the `WITH RECURSIVE` query should be adjusted such that it only includes a node if it has not yet been visited. This is implemented by using a subquery that checks if any of the previous paths includes the node:

```
WITH RECURSIVE paths(startNode, endNode, path) AS (
   SELECT -- define the path as the first edge of the traversal
        node1id AS startNode,
        node2id AS endNode,
        [node1id, node2id] AS path
     FROM edge
```

```
    WHERE startNode = 1
  UNION ALL
  SELECT -- concatenate new edge to the path
       paths.startNode AS startNode,
       node2id AS endNode,
       array_append(path, node2id) AS path
    FROM paths
    JOIN edge ON paths.endNode = node1id
   -- Prevent adding a node that was visited previously by any path.
   -- This ensures that (1) no cycles occur and (2) only nodes that
   -- were not visited by previous (shorter) paths are added to a path.
   WHERE NOT EXISTS (SELECT 1
                     FROM paths previous_paths
                     WHERE list_contains(previous_paths.path, node2id))
)
SELECT startNode, endNode, path
FROM paths
ORDER BY length(path), path;
```

startNode	endNode	path
1	3	[1, 3]
1	5	[1, 5]
1	8	[1, 3, 8]
1	10	[1, 3, 10]
1	4	[1, 5, 4]
1	8	[1, 5, 8]

Enumerate Unweighted Shortest Paths between Two Nodes `WITH RECURSIVE` can also be used to find **all (unweighted) shortest paths between two nodes**. To ensure that the recursive query is stopped as soon as we reach the end node, we use a window function which checks whether the end node is among the newly added nodes.

The following query returns all unweighted shortest paths between nodes 1 (start node) and 8 (end node):

```
WITH RECURSIVE paths(startNode, endNode, path, endReached) AS (
  SELECT -- define the path as the first edge of the traversal
       node1id AS startNode,
       node2id AS endNode,
       [node1id, node2id] AS path,
```

```
        (node2id = 8) AS endReached
    FROM edge
    WHERE startNode = 1
  UNION ALL
  SELECT -- concatenate new edge to the path
        paths.startNode AS startNode,
        node2id AS endNode,
        array_append(path, node2id) AS path,
        max(CASE WHEN node2id = 8 THEN 1 ELSE 0 END)
            OVER (ROWS BETWEEN UNBOUNDED PRECEDING
                           AND UNBOUNDED FOLLOWING) AS endReached
    FROM paths
    JOIN edge ON paths.endNode = node1id
   WHERE NOT EXISTS (SELECT 1
                     FROM paths previous_paths
                     WHERE list_contains(previous_paths.path, node2id))
     AND paths.endReached = 0
)
SELECT startNode, endNode, path
FROM paths
WHERE endNode = 8
ORDER BY length(path), path;
```

startNode	endNode	path
1	8	[1, 3, 8]
1	8	[1, 5, 8]

Common Table Expressions

WINDOW Clause

The `WINDOW` clause allows you to specify named windows that can be used within window functions. These are useful when you have multiple window functions, as they allow you to avoid repeating the same window clause.

Syntax

QUALIFY Clause

The QUALIFY clause is used to filter the results of WINDOW functions. This filtering of results is similar to how a HAVING clause filters the results of aggregate functions applied based on the GROUP BY clause.

The QUALIFY clause avoids the need for a subquery or WITH clause to perform this filtering (much like HAVING avoids a subquery). An example using a WITH clause instead of QUALIFY is included below the QUALIFY examples.

Note that this is filtering based on WINDOW functions, not necessarily based on the WINDOW clause. The WINDOW clause is optional and can be used to simplify the creation of multiple WINDOW function expressions.

The position of where to specify a QUALIFY clause is following the WINDOW clause in a SELECT statement (WINDOW does not need to be specified), and before the ORDER BY.

Examples

Each of the following examples produce the same output, located below.

```
-- Filter based on a WINDOW function defined in the QUALIFY clause
SELECT
    schema_name,
    function_name,
    -- In this example the function_rank column in the select clause is for
     ↪  reference
    row_number() OVER (PARTITION BY schema_name ORDER BY function_name) AS
 ↪  function_rank
FROM duckdb_functions()
QUALIFY
    row_number() OVER (PARTITION BY schema_name ORDER BY function_name) < 3;

-- Filter based on a WINDOW function defined in the SELECT clause
SELECT
    schema_name,
    function_name,
    row_number() OVER (PARTITION BY schema_name ORDER BY function_name) AS
 ↪  function_rank
FROM duckdb_functions()
QUALIFY
```

```
    function_rank < 3;

-- Filter based on a WINDOW function defined in the QUALIFY clause, but
 ↪  using the WINDOW clause
SELECT
    schema_name,
    function_name,
    -- In this example the function_rank column in the select clause is for
     ↪  reference
    row_number() OVER my_window AS function_rank
FROM duckdb_functions()
WINDOW
    my_window AS (PARTITION BY schema_name ORDER BY function_name)
QUALIFY
    row_number() OVER my_window < 3;

-- Filter based on a WINDOW function defined in the SELECT clause, but using
 ↪  the WINDOW clause
SELECT
    schema_name,
    function_name,
    row_number() OVER my_window AS function_rank
FROM duckdb_functions()
WINDOW
    my_window AS (PARTITION BY schema_name ORDER BY function_name)
QUALIFY
    function_rank < 3;

-- Equivalent query based on a WITH clause (without QUALIFY clause)
WITH ranked_functions AS (
    SELECT
        schema_name,
        function_name,
        row_number() OVER (PARTITION BY schema_name ORDER BY function_name)
 ↪  AS function_rank
    FROM duckdb_functions()
)
SELECT
    *
FROM ranked_functions
WHERE
    function_rank < 3;
```

schema_name	function_name	function_rank
main	!__postfix	1
main	!~~	2
pg_catalog	col_description	1
pg_catalog	format_pg_type	2

Syntax

VALUES Clause

The `VALUES` clause is used to specify a fixed number of rows. The `VALUES` clause can be used as a stand-alone statement, as part of the `FROM` clause, or as input to an `INSERT INTO` statement.

Examples

```
-- generate two rows and directly return them
VALUES ('Amsterdam', 1), ('London', 2);
-- generate two rows as part of a FROM clause, and rename the columns
SELECT * FROM (VALUES ('Amsterdam', 1), ('London', 2)) Cities(Name, Id);
-- generate two rows and insert them into a table
INSERT INTO Cities VALUES ('Amsterdam', 1), ('London', 2);
-- create a table directly from a VALUES clause
CREATE TABLE Cities AS SELECT * FROM (VALUES ('Amsterdam', 1), ('London',
 ↪  2)) Cities(Name, Id);
```

Syntax

FILTER Clause

The `FILTER` clause may optionally follow an aggregate function in a `SELECT` statement. This will filter the rows of data that are fed into the aggregate function in the same way that a `WHERE` clause filters rows, but localized to the specific aggregate function. `FILTER`s are not currently able to be used when the aggregate function is in a windowing context.

There are multiple types of situations where this is useful, including when evaluating multiple aggregates with different filters, and when creating a pivoted view of a dataset. `FILTER` provides a cleaner

syntax for pivoting data when compared with the more traditional `CASE WHEN` approach discussed below.

Some aggregate functions also do not filter out null values, so using a `FILTER` clause will return valid results when at times the `CASE WHEN` approach will not. This occurs with the functions `FIRST` and `LAST`, which are desirable in a non-aggregating pivot operation where the goal is to simply re-orient the data into columns rather than re-aggregate it. `FILTER` also improves null handling when using the `LIST` and `ARRAY_AGG` functions, as the `CASE WHEN` approach will include null values in the list result, while the `FILTER` clause will remove them.

Examples

```
-- Compare total row count to:
--   The number of rows where i <= 5
--   The number of rows where i is odd
SELECT
    count(*) AS total_rows,
    count(*) FILTER (WHERE i <= 5) AS lte_five,
    count(*) FILTER (WHERE i % 2 = 1) AS odds
FROM generate_series(1, 10) tbl(i);
```

total_rows	lte_five	odds
10	5	5

```
-- Different aggregate functions may be used, and multiple WHERE expressions
 ↪ are also permitted
--   The sum of i for rows where i <= 5
--   The median of i where i is odd
SELECT
    sum(i) FILTER (WHERE i <= 5) AS lte_five_sum,
    median(i) FILTER (WHERE i % 2 = 1) AS odds_median,
    median(i) FILTER (WHERE i % 2 = 1 AND i <= 5) AS odds_lte_five_median
FROM generate_series(1, 10) tbl(i);
```

lte_five_sum	odds_median	odds_lte_five_median
15	5.0	3.0

The FILTER clause can also be used to pivot data from rows into columns. This is a static pivot, as columns must be defined prior to runtime in SQL. However, this kind of statement can be dynamically generated in a host programming language to leverage DuckDB's SQL engine for rapid, larger than memory pivoting.

```
--First generate an example dataset
CREATE TEMP TABLE stacked_data AS
    SELECT
        i,
        CASE WHEN i <= rows * 0.25 THEN 2022
             WHEN i <= rows * 0.5 THEN 2023
             WHEN i <= rows * 0.75 THEN 2024
             WHEN i <= rows * 0.875 THEN 2025
             ELSE NULL
             END AS year
    FROM (
        SELECT
            i,
            count(*) OVER () AS rows
        FROM generate_series(1, 100000000) tbl(i)
    ) tbl;

--"Pivot" the data out by year (move each year out to a separate column)
SELECT
    count(i) FILTER (WHERE year = 2022) AS "2022",
    count(i) FILTER (WHERE year = 2023) AS "2023",
    count(i) FILTER (WHERE year = 2024) AS "2024",
    count(i) FILTER (WHERE year = 2025) AS "2025",
    count(i) FILTER (WHERE year IS NULL) AS "NULLs"
FROM stacked_data;

--This syntax produces the same results as the FILTER clauses above
SELECT
    count(CASE WHEN year = 2022 THEN i END) AS "2022",
    count(CASE WHEN year = 2023 THEN i END) AS "2023",
    count(CASE WHEN year = 2024 THEN i END) AS "2024",
    count(CASE WHEN year = 2025 THEN i END) AS "2025",
    count(CASE WHEN year IS NULL THEN i END) AS "NULLs"
FROM stacked_data;
```

2022	2023	2024	2025	NULLs
25000000	25000000	25000000	12500000	12500000

However, the `CASE WHEN` approach will not work as expected when using an aggregate function that does not ignore `NULL` values. The `FIRST` function falls into this category, so `FILTER` is preferred in this case.

```
--"Pivot" the data out by year (move each year out to a separate column)
SELECT
    first(i) FILTER (WHERE year = 2022) AS "2022",
    first(i) FILTER (WHERE year = 2023) AS "2023",
    first(i) FILTER (WHERE year = 2024) AS "2024",
    first(i) FILTER (WHERE year = 2025) AS "2025",
    first(i) FILTER (WHERE year IS NULL) AS "NULLs"
FROM stacked_data;
```

2022	2023	2024	2025	NULLs
1474561	25804801	50749441	76431361	87500001

```
--This will produce NULL values whenever the first evaluation of the CASE
 ↪  WHEN clause returns a NULL
SELECT
    first(CASE WHEN year = 2022 THEN i END) AS "2022",
    first(CASE WHEN year = 2023 THEN i END) AS "2023",
    first(CASE WHEN year = 2024 THEN i END) AS "2024",
    first(CASE WHEN year = 2025 THEN i END) AS "2025",
    first(CASE WHEN year IS NULL THEN i END) AS "NULLs"
FROM stacked_data;
```

2022	2023	2024	2025	NULLs
1228801	NULL	NULL	NULL	NULL

Aggregate Function Syntax (Including Filter Clause)

Set Operations

Set operations allow queries to be combined according to set operation semantics. Set operations refer to the `UNION [ALL]`, `INTERSECT` and `EXCEPT` clauses.

Traditional set operations unify queries **by column position**, and require the to-be-combined queries to have the same number of input columns. If the columns are not of the same type, casts may be added. The result will use the column names from the first query.

DuckDB also supports `UNION BY NAME`, which joins columns by name instead of by position. `UNION BY NAME` does not require the inputs to have the same number of columns. `NULL` values will be added in case of missing columns.

Examples

```
-- the values [0..10) and [0..5)
SELECT * FROM range(10) t1 UNION ALL SELECT * FROM range(5) t2;
-- the values [0..10) (` UNION` eliminates duplicates)
SELECT * FROM range(10) t1 UNION SELECT * FROM range(5) t2;
-- the values [0..5] (all values that are both in t1 and t2)
SELECT * FROM range(10) t1 INTERSECT SELECT * FROM range(6) t2;
-- the values [5..10)
SELECT * FROM range(10) t1 EXCEPT SELECT * FROM range(5) t2;
-- two rows, (24, NULL) and (NULL, Amsterdam)
SELECT 24 AS id UNION ALL BY NAME SELECT 'Amsterdam' AS City;
```

Syntax

Example Table

```
CREATE TABLE capitals(city VARCHAR, country VARCHAR);
INSERT INTO capitals VALUES ('Amsterdam', 'NL'), ('Berlin', 'Germany');

CREATE TABLE weather(city VARCHAR, degrees INTEGER, date DATE);
INSERT INTO weather VALUES ('Amsterdam', 10, '2022-10-14'), ('Seattle', 8,
 ↪  '2022-10-12');
```

UNION (ALL)

The `UNION` clause can be used to combine rows from multiple queries. The queries are required to have the same number of columns and the same column types.

The `UNION` clause performs duplicate elimination by default - only unique rows will be included in the result.

`UNION ALL` returns all rows of both queries *without* duplicate elimination.

```
SELECT city FROM capitals UNION SELECT city FROM weather;
-- Amsterdam, Berlin, Seattle

SELECT city FROM capitals UNION ALL SELECT city FROM weather;
-- Amsterdam, Amsterdam, Berlin, Seattle
```

INTERSECT

The `INTERSECT` clause can be used to select all rows that occur in the result of **both** queries. Note that `INTERSECT` performs duplicate elimination, so only unique rows are returned.

```
SELECT city FROM capitals INTERSECT SELECT city FROM weather;
-- Amsterdam
```

EXCEPT

The `EXCEPT` clause can be used to select all rows that **only** occur in the left query. Note that `EXCEPT` performs duplicate elimination, so only unique rows are returned.

```
SELECT city FROM capitals EXCEPT SELECT city FROM weather;
-- Berlin
```

UNION (ALL) BY NAME

The `UNION (ALL) BY NAME` clause can be used to combine rows from different tables by name, instead of by position. `UNION BY NAME` does not require both queries to have the same number of columns. Any columns that are only found in one of the queries are filled with `NULL` values for the other query.

```
SELECT * FROM capitals UNION BY NAME SELECT * FROM weather;
```

```
┌──────────┬─────────┬─────────┬────────────┐
│   city   │ country │ degrees │    date    │
│ varchar  │ varchar │  int32  │    date    │
├──────────┼─────────┼─────────┼────────────┤
│ Amsterdam │ NULL    │      10 │ 2022-10-14 │
│ Seattle  │ NULL    │       8 │ 2022-10-12 │
│ Amsterdam │ NL      │    NULL │ NULL       │
│ Berlin   │ Germany │    NULL │ NULL       │
└──────────┴─────────┴─────────┴────────────┘
```

`UNION BY NAME` performs duplicate elimination, whereas `UNION ALL BY NAME` does not.

Data Types

Data Types

General-Purpose Data Types

The table below shows all the built-in general-purpose data types. The alternatives listed in the aliases column can be used to refer to these types as well, however, note that the aliases are not part of the SQL standard and hence might not be accepted by other database engines.

Name	Aliases	Description
`BIGINT`	`INT8`, `LONG`	signed eight-byte integer
`BIT`	`BITSTRING`	string of 1's and 0's
`BOOLEAN`	`BOOL`, `LOGICAL`	logical boolean (true/false)
`BLOB`	`BYTEA`, `BINARY`, `VARBINARY`	variable-length binary data
`DATE`		calendar date (year, month day)
`DOUBLE`	`FLOAT8`, `NUMERIC`, `DECIMAL`	double precision floating-point number (8 bytes)
`DECIMAL(prec, scale)`		fixed-precision number with the given width (precision) and scale
`HUGEINT`		signed sixteen-byte integer
`INTEGER`	`INT4`, `INT`, `SIGNED`	signed four-byte integer

Name	Aliases	Description
`INTERVAL`		date / time delta
`REAL`	`FLOAT4`, `FLOAT`	single precision floating-point number (4 bytes)
`SMALLINT`	`INT2`, `SHORT`	signed two-byte integer
`TIME`		time of day (no time zone)
`TIMESTAMP`	`DATETIME`	combination of time and date
`TIMESTAMP WITH TIME ZONE`	`TIMESTAMPTZ`	combination of time and date that uses the current time zone
`TINYINT`	`INT1`	signed one-byte integer
`UBIGINT`		unsigned eight-byte integer
`UINTEGER`		unsigned four-byte integer
`USMALLINT`		unsigned two-byte integer
`UTINYINT`		unsigned one-byte integer
`UUID`		UUID data type
`VARCHAR`	`CHAR`, `BPCHAR`, `TEXT`, `STRING`	variable-length character string

Nested / Composite Types

DuckDB supports four nested data types: `LIST`, `STRUCT`, `MAP` and `UNION`. Each supports different use cases and has a different structure.

Name	Description	Rules when used in a column	Build from values	Define in DDL/CREATE
LIST	An ordered sequence of data values of the same type.	Each row must have the same data type within each LIST, but can have any number of elements.	`[1, 2, 3]`	INT[]

Name	Description	Rules when used in a column	Build from values	Define in DDL/CREATE
STRUCT	A dictionary of multiple named values, where each key is a string, but the value can be a different type for each key.	Each row must have the same keys.	`{'i': 42, 'j': 'a'}`	STRUCT(i INT, j VARCHAR)
MAP	A dictionary of multiple named values, each key having the same type and each value having the same type. Keys and values can be any type and can be different types from one another.	Rows may have different keys.	`map([1, 2], ['a', 'b'])`	MAP(INT, VARCHAR)
UNION	A union of multiple alternative data types, storing one of them in each value at a time. A union also contains a discriminator "tag" value to inspect and access the currently set member type.	Rows may be set to different member types of the union.	union_value(num := 2)	UNION(num INT, text VARCHAR)

Nesting

`LIST`s, `STRUCT`s, `MAP`s and `UNION`s can be arbitrarily nested to any depth, so long as the type rules are observed.

```
-- Struct with lists
SELECT {'birds': ['duck', 'goose', 'heron'], 'aliens': NULL, 'amphibians':
 ↪  ['frog', 'toad']};
```

```
-- Struct with list of maps
SELECT {'test': [map([1, 5], [42.1, 45]), map([1, 5], [42.1, 45])]};
-- A list of unions
SELECT [union_value(num := 2), union_value(str := 'ABC')::UNION(str VARCHAR,
 ↪  num INTEGER)];
```

Bitstring Type

Name	Aliases	Description
BIT	BITSTRING	variable-length strings of 1's and 0's

Bitstrings are strings of 1's and 0's. The bit type data is of variable length. A bitstring value requires 1 byte for each group of 8 bits, plus a fixed amount to store some metadata.

By default bitstrings will not be padded with zeroes. Bitstrings can be very large, having the same size restrictions as BLOBs.

```
-- create a bitstring
SELECT '101010'::BIT
-- create a bitstring with predefined length
-- the resulting bitstring will be left-padded with zeroes. This returns
 ↪  000000101011
SELECT bitstring('0101011', 12);
```

Functions

See Bitstring Functions.

Blob Type

Name	Aliases	Description
BLOB	BYTEA, BINARY, VARBINARY	variable-length binary data

The blob (**B**inary **L**arge **OB**ject) type represents an arbitrary binary object stored in the database system. The blob type can contain any type of binary data with no restrictions. What the actual bytes represent is opaque to the database system.

```
-- create a blob value with a single byte (170)
SELECT '\xAA'::BLOB;
-- create a blob value with three bytes (170, 171, 172)
SELECT '\xAA\xAB\xAC'::BLOB;
-- create a blob value with two bytes (65, 66)
SELECT 'AB'::BLOB;
```

Blobs are typically used to store non-textual objects that the database does not provide explicit support for, such as images. While blobs can hold objects up to 4GB in size, typically it is not recommended to store very large objects within the database system. In many situations it is better to store the large file on the file system, and store the path to the file in the database system in a VARCHAR field.

Functions

See Blob Functions.

Boolean Type

Name	Aliases	Description
BOOLEAN	BOOL	logical boolean (`true`/`false`)

The BOOLEAN type represents a statement of truth (”true” or ”false”). In SQL, the boolean field can also have a third state ”unknown” which is represented by the SQL NULL value.

```
-- select the three possible values of a boolean column
SELECT true, false, NULL::BOOLEAN;
```

Boolean values can be explicitly created using the literals `true` and `false`. However, they are most often created as a result of comparisons or conjunctions. For example, the comparison `i > 10` results in a boolean value. Boolean values can be used in the WHERE and HAVING clauses of a SQL statement to filter out tuples from the result. In this case, tuples for which the predicate evaluates to `true` will pass the filter, and tuples for which the predicate evaluates to `false` or NULL will be filtered out. Consider the following example:

```
-- create a table with the value (5), (15) and (NULL)
CREATE TABLE integers(i INTEGER);
INSERT INTO integers VALUES (5), (15), (NULL);
```

```
-- select all entries where i > 10
SELECT * FROM integers WHERE i > 10;
-- in this case (5) and (NULL) are filtered out:
-- 5 > 10    = false
-- NULL > 10 = NULL
-- The result is (15)
```

Conjunctions

The AND/OR conjunctions can be used to combine boolean values.

Below is the truth table for the AND conjunction (i.e., x AND y).

X	X AND true	X AND false	X AND NULL
true	true	false	NULL
false	false	false	false
NULL	NULL	false	NULL

Below is the truth table for the OR conjunction (i.e., x OR y).

X	X OR true	X OR false	X OR NULL
true	true	true	true
false	true	false	NULL
NULL	true	NULL	NULL

Expressions

See Logical Operators and Comparison Operators.

Date Types

Name	Aliases	Description
DATE		calendar date (year, month day)

A date specifies a combination of year, month and day. DuckDB follows the SQL standard's lead by counting dates exclusively in the Gregorian calendar, even for years before that calendar was in use. Dates can be created using the DATE keyword, where the data must be formatted according to the ISO 8601 format (YYYY-MM-DD).

```
-- 20 September, 1992
SELECT DATE '1992-09-20';
```

Special Values

There are also three special date values that can be used on input:

Input String	Description
epoch	1970-01-01 (Unix system day zero)
infinity	later than all other dates
-infinity	earlier than all other dates

The values `infinity` and `-infinity` are specially represented inside the system and will be displayed unchanged; but `epoch` is simply a notational shorthand that will be converted to the date value when read.

```
SELECT '-infinity'::DATE, 'epoch'::DATE, 'infinity'::DATE;
```

Negative	Epoch	Positive
-infinity	1970-01-01	infinity

Functions

See Date Functions.

Enum Types

Name	Description
ENUM	Dictionary Encoding representing all possible string values of a column.

Enums

The ENUM type represents a dictionary data structure with all possible unique values of a column. For example, a column storing the days of the week can be an Enum holding all possible days. Enums are particularly interesting for string columns with low cardinality (i.e., fewer distinct values). This is because the column only stores a numerical reference to the string in the Enum dictionary, resulting in immense savings in disk storage and faster query performance.

Enum Definition Enum types are created from either a hardcoded set of values or from a select statement that returns a single column of varchars. The set of values in the select statement will be deduplicated, but if the enum is created from a hardcoded set there may not be any duplicates.

```
-- Create enum using hardcoded values
CREATE TYPE ${enum_name} AS ENUM ([${value_1}, ${value_2},...]);

-- Create enum using a select statement that returns a single column of
 ↪  varchars
CREATE TYPE ${enum_name} AS ENUM (${SELECT expression});
```

For example:

```
-- Creates new user defined type 'mood' as an Enum
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');

-- This will fail since the mood type already exists
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy', 'anxious');

-- This will fail since Enums cannot hold null values
CREATE TYPE breed AS ENUM ('maltese', NULL);

-- This will fail since Enum values must be unique
CREATE TYPE breed AS ENUM ('maltese', 'maltese');
```

```
-- Create an enum from a select statement
-- First create an example table of values
CREATE TABLE my_inputs AS
    SELECT 'duck'  AS my_varchar UNION ALL
    SELECT 'duck'  AS my_varchar UNION ALL
    SELECT 'goose' AS my_varchar;

-- Create an enum using the unique string values in the my_varchar column
CREATE TYPE birds AS ENUM (SELECT my_varchar FROM my_inputs);

-- Show the available values in the birds enum using the enum_range function
SELECT enum_range(NULL::birds) AS my_enum_range;
```

my_enum_range
[duck, goose]

Enum Usage After an enum has been created, it can be used anywhere a standard built-in type is used. For example, we can create a table with a column that references the enum.

```
-- Creates a table person, with attributes name (string type) and current_
 ↪  mood (mood type)
CREATE TABLE person (
    name text,
    current_mood mood
);

-- Inserts tuples in the person table
INSERT INTO person VALUES ('Pedro', 'happy'), ('Mark', NULL), ('Pagliacci',
 ↪  'sad'), ('Mr. Mackey', 'ok');

-- This will fail since the mood type does not have a 'quackity-quack' value.
INSERT INTO person VALUES ('Hannes', 'quackity-quack');

-- The string 'sad' is cast to the type Mood, returning a numerical
 ↪  reference value.
-- This makes the comparison a numerical comparison instead of a string
 ↪  comparison.
SELECT * FROM person WHERE current_mood = 'sad';
----
Pagliacci
```

```
-- If you are importing data from a file, you can create an Enum for a
 ↪  VARCHAR column before importing
-- The subquery select automatically selects only distinct values
CREATE TYPE mood AS ENUM (SELECT mood FROM 'path/to/file.csv');

-- Then you can create a table with the ENUM type and import using any data
 ↪  import statement
CREATE TABLE person(name text, current_mood mood);
COPY person FROM 'path/to/file.csv' (AUTO_DETECT true);
```

Enum vs. Strings DuckDB Enums are automatically cast to VARCHAR types whenever necessary. This characteristic allows for ENUM columns to be used in any VARCHAR function. In addition, it also allows for comparisons between different ENUM columns, or an ENUM and a VARCHAR column.

For example:

```
-- regexp_matches is a function that takes a VARCHAR, hence current_mood is
 ↪  cast to VARCHAR
SELECT regexp_matches(current_mood, '.*a.*') FROM person;
----
true

true
false


CREATE TYPE new_mood AS ENUM ('happy', 'anxious');

CREATE TABLE person_2(
    name text,
    current_mood mood,
    future_mood new_mood
    past_mood VARCHAR
);

-- Since current_mood and future_mood are constructed on different ENUMs
-- DuckDB will cast both ENUMs to strings and perform a string comparison.
SELECT * FROM person_2 where current_mood = future_mood;

-- Since current_mood is an ENUM
-- DuckDB will cast the current_mood ENUM to VARCHAR and perform a string
 ↪  comparison
```

```
SELECT * FROM person_2 where current_mood = past_mood;
```

Enum Removal Enum types are stored in the catalog, and a catalog dependency is added to each table that uses them. It is possible to drop an Enum from the catalog using the following command:

```
DROP TYPE ${enum_name};
```

Currently, it is possible to drop Enums that are used in tables without affecting the tables.

> **Note.** This feature is subject to change in future releases.

For example:

```
-- This will fail since person has a catalog dependency to the mood type
DROP TYPE mood;

DROP TABLE person;
DROP TABLE person_2;

-- This successfully removes the mood type.
-- Another option would be to DROP TYPE mood CASCADE (Drops the type and its
 ↪  dependents)
DROP TYPE mood;
```

Interval Type

Intervals represent a period of time. This period can be measured in a specific unit or combination of units, for example years, days, or seconds. Intervals are generally used to *modify* timestamps or dates by either adding or subtracting them.

Name	Description
INTERVAL	Period of time

An INTERVAL can be constructed by providing an amount together with a unit. Intervals can be added or subtracted from DATE or TIMESTAMP values.

```
-- 1 year
SELECT INTERVAL 1 YEAR;
-- add 1 year to a specific date
SELECT DATE '2000-01-01' + INTERVAL 1 YEAR;
```

```
-- subtract 1 year from a specific date
SELECT DATE '2000-01-01' - INTERVAL 1 YEAR;
-- construct an interval from a column, instead of a constant
SELECT INTERVAL (i) YEAR FROM range(1, 5) t(i);
-- construct an interval with mixed units
SELECT INTERVAL '1 month 1 day';

-- WARNING: If a decimal value is specified, it will be automatically
 ↪  rounded to an integer
-- To use more precise values, simply use a more granular date part
-- (In this example use 18 MONTHS instead of 1.5 YEARS)
-- The statement below is equivalent to to_years(CAST(1.5 AS INTEGER))
-- 2 years
SELECT INTERVAL '1.5' YEARS; --WARNING! This returns 2 years!
```

Details

The interval class represents a period of time using three distinct components: the *month*, *day* and *microsecond*. These three components are required because there is no direct translation between them. For example, a month does not correspond to a fixed amount of days. That depends on *which month is referenced*. February has fewer days than March.

The division into components makes the interval class suitable for adding or subtracting specific time units to a date. For example, we can generate a table with the first day of every month using the following SQL query:

```
SELECT DATE '2000-01-01' + INTERVAL (i) MONTH FROM range(12) t(i);
```

Difference between Dates

If we subtract two timestamps from one another, we obtain an interval describing the difference between the timestamps with the *days and microseconds* components. For example:

```
SELECT TIMESTAMP '2000-02-01 12:00:00' - TIMESTAMP '2000-01-01 11:00:00' AS
 ↪  diff;
```

diff
interval
31 days 01:00:00

The `datediff` function can be used to obtain the difference between two dates for a specific unit.

```
SELECT datediff('month', TIMESTAMP '2000-01-01 11:00:00', TIMESTAMP
 ↪  '2000-02-01 12:00:00') AS diff;
```

```
┌─────────┐
│  diff   │
│  int64  │
├─────────┤
│       1 │
└─────────┘
```

Functions

See the Date Part Functions page for a list of available date parts for use with an `INTERVAL`.

See the Interval Operators page for functions that operate on intervals.

List

List Data Type

A `LIST` column can have values with different lengths, but they must all have the same underlying type. `LIST`s are typically used to store arrays of numbers, but can contain any uniform data type, including other `LIST`s and `STRUCT`s.

`LIST`s are similar to PostgreSQL's ARRAY type. DuckDB uses the `LIST` terminology, but some array functions are provided for PostgreSQL compatibility.

See the data types overview for a comparison between nested data types.

Lists can be created using the `LIST_VALUE(expr, ...)` function or the equivalent bracket notation `[expr, ...]`. The expressions can be constants or arbitrary expressions.

Creating Lists

```
-- List of integers
SELECT [1, 2, 3];
-- List of strings with a NULL value
SELECT ['duck', 'goose', NULL, 'heron'];
-- List of lists with NULL values
SELECT [['duck', 'goose', 'heron'], NULL, ['frog', 'toad'], []];
-- Create a list with the list_value function
```

```
SELECT list_value(1, 2, 3);
-- Create a table with an integer list column and a varchar list column
CREATE TABLE list_table (int_list INT[], varchar_list VARCHAR[]);
```

Retrieving from Lists Retrieving one or more values from a list can be accomplished using brackets and slicing notation, or through list functions like `list_extract`. Multiple equivalent functions are provided as aliases for compatibility with systems that refer to lists as arrays. For example, the function `array_slice`.

```
-- Note that we wrap the list creation in parenthesis so that it happens
 ↪  first.
-- This is only needed in our basic examples here, not when working with a
 ↪  list column
-- For example, this can't be parsed: SELECT ['a', 'b', 'c'][1]
```

example	result
`SELECT (['a', 'b', 'c'])[3]`	'c'
`SELECT (['a', 'b', 'c'])[-1]`	'c'
`SELECT (['a', 'b', 'c'])[2 + 1]`	'c'
`SELECT list_extract(['a', 'b', 'c'], 3)`	'c'
`SELECT (['a', 'b', 'c'])[1:2]`	['a','b']
`SELECT (['a', 'b', 'c'])[:2]`	['a','b']
`SELECT (['a', 'b', 'c'])[-2:]`	['b','c']
`SELECT list_slice(['a', 'b', 'c'], 2, 3)`	['b','c']

Ordering

The ordering is defined positionally. `NULL` values compare greater than all other values and are considered equal to each other.

Null Comparisons

At the top level, `NULL` nested values obey standard SQL `NULL` comparison rules: comparing a `NULL` nested value to a non-`NULL` nested value produces a `NULL` result. Comparing nested value *members*

, however, uses the internal nested value rules for NULLs, and a NULL nested value member will compare above a non-NULL nested value member.

Functions

See Nested Functions.

Map

Map Data Type

MAPs are similar to STRUCTs in that they are an ordered list of "entries" where a key maps to a value. However, MAPs do not need to have the same keys present for each row, and thus are suitable for other use cases. MAPs are useful when the schema is unknown beforehand or when the schema varies per row; their flexibility is a key differentiator.

MAPs must have a single type for all keys, and a single type for all values. Keys and values can be any type, and the type of the keys does not need to match the type of the values (Ex: a MAP of VARCHAR to INT is valid). MAPs may not have duplicate keys. MAPs return an empty list if a key is not found rather than throwing an error as structs do.

In contrast, STRUCTs must have string keys, but each key may have a value of a different type. See the data types overview for a comparison between nested data types.

To construct a MAP, use the bracket syntax preceded by the MAP keyword.

Creating Maps

```
-- A map with varchar keys and integer values. This returns {key1=1, key2=5}
SELECT map { 'key1': 1, 'key2': 5 };
-- Alternatively use the map_from_entries function. This returns {key1=1,
 ↪  key2=5}
SELECT map_from_entries([(key1, 1), (key2, 5)]);
-- A map with integer keys and numeric values. This returns {1=42.001,
 ↪  5=-32.100}
SELECT map { 1: 42.001, 5: -32.1 };
-- Keys and/or values can also be nested types.
-- This returns {[a, b]=[1.1, 2.2], [c, d]=[3.3, 4.4]}
SELECT map { ['a', 'b']: [1.1, 2.2], ['c', 'd']: [3.3, 4.4] };
-- Create a table with a map column that has integer keys and double values
CREATE TABLE map_table (map_col MAP(INT, DOUBLE));
```

Retrieving from Maps MAPs use bracket notation for retrieving values. Selecting from a MAP returns a LIST rather than an individual value, with an empty LIST meaning that the key was not found.

```
-- Use bracket notation to retrieve a list containing the value at a key's
 ↪  location. This returns [5]
-- Note that the expression in bracket notation must match the type of the
 ↪  map's key
SELECT map { 'key1': 5, 'key2': 43 }['key1'];
-- To retrieve the underlying value, use list selection syntax to grab the
 ↪  first element.
-- This returns 5
SELECT map { 'key1': 5, 'key2': 43 }['key1'][1];
-- If the element is not in the map, an empty list will be returned. Returns
 ↪  []
-- Note that the expression in bracket notation must match the type of the
 ↪  map's key else an error is returned
SELECT map { 'key1': 5, 'key2': 43 }['key3'];
-- The element_at function can also be used to retrieve a map value. This
 ↪  returns [5]
SELECT element_at(map { 'key1': 5, 'key2': 43 }, 'key1');
```

Comparison Operators

Nested types can be compared using all the comparison operators. These comparisons can be used in logical expressions for both WHERE and HAVING clauses, as well as for creating Boolean values.

The ordering is defined positionally in the same way that words can be ordered in a dictionary. NULL values compare greater than all other values and are considered equal to each other.

At the top level, NULL nested values obey standard SQL NULL comparison rules: comparing a NULL nested value to a non-NULL nested value produces a NULL result. Comparing nested value *members*, however, uses the internal nested value rules for NULLs, and a NULL nested value member will compare above a non-NULL nested value member.

Functions

See Nested Functions.

NULL Values

NULL values are special values that are used to represent missing data in SQL. Columns of any type can contain NULL values. Logically, a NULL value can be seen as "the value of this field is unknown".

```
-- insert a null value into a table
CREATE TABLE integers(i INTEGER);
INSERT INTO integers VALUES (NULL);
```

NULL values have special semantics in many parts of the query as well as in many functions:

> **Note.** Any comparison with a NULL value returns NULL, including NULL=NULL.

You can use `IS NOT DISTINCT FROM` to perform an equality comparison where NULL values compare equal to each other. Use `IS (NOT) NULL` to check if a value is NULL.

```
SELECT NULL=NULL;
-- returns NULL
SELECT NULL IS NOT DISTINCT FROM NULL;
-- returns true
SELECT NULL IS NULL;
-- returns true
```

NULL and Functions

A function that has input argument as NULL **usually** returns NULL.

```
SELECT COS(NULL);
-- NULL
```

COALESCE is an exception to this. COALESCE takes any number of arguments, and returns for each row the first argument that is not NULL. If all arguments are NULL, COALESCE also returns NULL.

```
SELECT COALESCE(NULL, NULL, 1);
-- 1
SELECT COALESCE(10, 20);
-- 10
SELECT COALESCE(NULL, NULL);
-- NULL
```

IFNULL is a two-argument version of COALESCE

```
SELECT IFNULL(NULL, 'default_string');
-- default_string
SELECT IFNULL(1, 'default_string');
-- 1
```

NULL and Conjunctions

NULL values have special semantics in AND/OR conjunctions. For the ternary logic truth tables, see the Boolean Type documentation.

NULL and Aggregate Functions

NULL values are ignored in most aggregate functions.

Aggregate functions that do not ignore NULL values include: FIRST, LAST, LIST, and ARRAY_AGG. To exclude NULL values from those aggregate functions, the FILTER clause can be used.

```
CREATE TABLE integers(i INTEGER);
INSERT INTO integers VALUES (1), (10), (NULL);

SELECT MIN(i) FROM integers;
-- 1

SELECT MAX(i) FROM integers;
-- 10
```

Numeric Types

Integer Types

The types TINYINT, SMALLINT, INTEGER, BIGINT and HUGEINT store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error. The types UTINYINT, USMALLINT, UINTEGER, UBIGINT store whole unsigned numbers. Attempts to store negative numbers or values outside of the allowed range will result in an error

Name	Aliases	Min	Max
TINYINT	INT1	-128	127
SMALLINT	INT2, SHORT	-32768	32767
INTEGER	INT4, INT, SIGNED	-2147483648	2147483647
BIGINT	INT8, LONG	-9223372036854775808	9223372036854775807

Name	Aliases	Min	Max
HUGEINT		-170141183460469231731687303715884105728	$2^{127} - 1$
UTINYINT	-	0	255
USMALLINT	-	0	65535
UINTEGER	-	0	4294967295
UBIGINT	-	0	18446744073709551615

The type integer is the common choice, as it offers the best balance between range, storage size, and performance. The SMALLINT type is generally only used if disk space is at a premium. The BIGINT and HUGEINT types are designed to be used when the range of the integer type is insufficient.

Fixed-Point Decimals

The data type DECIMAL(WIDTH, SCALE) represents an exact fixed-point decimal value. When creating a value of type DECIMAL, the WIDTH and SCALE can be specified to define which size of decimal values can be held in the field. The WIDTH field determines how many digits can be held, and the scale determines the amount of digits after the decimal point. For example, the type DECIMAL(3, 2) can fit the value 1.23, but cannot fit the value 12.3 or the value 1.234. The default WIDTH and SCALE is DECIMAL(18, 3), if none are specified.

Internally, decimals are represented as integers depending on their specified width.

Width	Internal	Size (Bytes)
1-4	INT16	2
5-9	INT32	4
10-18	INT64	8
19-38	INT128	16

Performance can be impacted by using too large decimals when not required. In particular decimal values with a width above 19 are very slow, as arithmetic involving the INT128 type is much more expensive than operations involving the INT32 or INT64 types. It is therefore recommended to stick with a width of 18 or below, unless there is a good reason for why this is insufficient.

Floating-Point Types

The data types REAL and DOUBLE precision are inexact, variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Name	Aliases	Description
REAL	FLOAT4, FLOAT	single precision floating-point number (4 bytes)
DOUBLE	FLOAT8	double precision floating-point number (8 bytes)

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and retrieving a value might show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed here, except for the following points:

- If you require exact storage and calculations (such as for monetary amounts), use the numeric type instead.
- If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
- Comparing two floating-point values for equality might not always work as expected.

On most platforms, the REAL type has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The DOUBLE type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding might take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

In addition to ordinary numeric values, the floating-point types have several special values:

- `Infinity`
- `-Infinity`
- `NaN`

These represent the IEEE 754 special values ”infinity”, ”negative infinity”, and ”not-a-number”, respectively. (On a machine whose floating-point arithmetic does not follow IEEE 754, these values will probably not work as expected.) When writing these values as constants in an SQL command, you must put quotes around them, for example: `UPDATE table SET x = '-Infinity'`. On input, these strings are recognized in a case-insensitive manner.

Functions

See Numeric Functions and Operators.

Struct

Struct Data Type

Conceptually, a STRUCT column contains an ordered list of columns called "entries". The entries are referenced by name using strings. This document refers to those entry names as keys. Each row in the STRUCT column must have the same keys. The names of the struct entries are part of the *schema*. Each row in a STRUCT column must have the same layout. The names of the struct entries are case-insensitive.

STRUCTs are typically used to nest multiple columns into a single column, and the nested column can be of any type, including other STRUCTs and `LIST`s.

STRUCTs are similar to PostgreSQL's `ROW` type. The key difference is that DuckDB STRUCTs require the same keys in each row of a STRUCT column. This allows DuckDB to provide significantly improved performance by fully utilizing its vectorized execution engine, and also enforces type consistency for improved correctness. DuckDB includes a `row` function as a special way to produce a struct, but does not have a `ROW` data type. See an example below and the nested functions docs for details.

See the data types overview for a comparison between nested data types.

Structs can be created using the `STRUCT_PACK(name := expr, ...)` function or the equivalent array notation `{'name': expr, ...}` notation. The expressions can be constants or arbitrary expressions.

Creating Structs

```
-- Struct of integers
SELECT {'x': 1, 'y': 2, 'z': 3};
-- Struct of strings with a NULL value
SELECT {'yes': 'duck', 'maybe': 'goose', 'huh': NULL, 'no': 'heron'};
-- Struct with a different type for each key
SELECT {'key1': 'string', 'key2': 1, 'key3': 12.345};
-- Struct using the struct_pack function.
-- Note the lack of single quotes around the keys and the use of the :=
 ↪  operator
SELECT struct_pack(key1 := 'value1', key2 := 42);
-- Struct of structs with NULL values
```

```
SELECT {'birds':
            {'yes': 'duck', 'maybe': 'goose', 'huh': NULL, 'no': 'heron'},
        'aliens':
            NULL,
        'amphibians':
            {'yes':'frog', 'maybe': 'salamander', 'huh': 'dragon',
 ↪  'no':'toad'}
        };
-- Create a struct from columns and/or expressions using the row function.
-- This returns {'x': 1, 'v2': 2, 'y': a}
SELECT row(x, x + 1, y) FROM (SELECT 1 AS x, 'a' AS y);
-- If using multiple expressions when creating a struct, the row function is
 ↪  optional
-- This also returns {'x': 1, 'v2': 2, 'y': a}
SELECT (x, x + 1, y) FROM (SELECT 1 AS x, 'a' AS y);
```

Adding Field(s)/Value(s) to Structs

```
-- Add to a Struct of integers
SELECT struct_insert({'a': 1, 'b': 2, 'c': 3}, d := 4);
```

Retrieving from Structs Retrieving a value from a struct can be accomplished using dot notation, bracket notation, or through struct functions like `struct_extract`.

```
-- Use dot notation to retrieve the value at a key's location. This returns 1
-- The subquery generates a struct column "a", which we then query with a.x
SELECT a.x FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS a);
-- If key contains a space, simply wrap it in double quotes. This returns 1
-- Note: Use double quotes not single quotes
-- This is because this action is most similar to selecting a column from
 ↪  within the struct
SELECT a."x space" FROM (SELECT {'x space': 1, 'y': 2, 'z': 3} AS a);
-- Bracket notation may also be used. This returns 1
-- Note: Use single quotes since the goal is to specify a certain string
 ↪  key.
-- Only constant expressions may be used inside the brackets (no columns)
SELECT a['x space'] FROM (SELECT {'x space': 1, 'y': 2, 'z': 3} AS a);
-- The struct_extract function is also equivalent. This returns 1
SELECT struct_extract({'x space': 1, 'y': 2, 'z': 3}, 'x space');
```

Struct.* Rather than retrieving a single key from a struct, star notation (*) can be used to retrieve all keys from a struct as separate columns. This is particularly useful when a prior operation creates a

struct of unknown shape, or if a query must handle any potential struct keys.

```
-- All keys within a struct can be returned as separate columns using *
SELECT a.* FROM (SELECT {'x':1, 'y':2, 'z':3} AS a);
```

x	y	z
1	2	3

Dot Notation Order of Operations Referring to structs with dot notation can be ambiguous with referring to schemas and tables. In general, DuckDB looks for columns first, then for struct keys within columns. DuckDB resolves references in these orders, using the first match to occur:

No Dots

```
SELECT part1 FROM tbl
```

1. part1 is a column

One Dot

```
SELECT part1.part2 FROM tbl
```

1. part1 is a table, part2 is a column
2. part1 is a column, part2 is a property of that column

Two (or More) Dots

```
SELECT part1.part2.part3 FROM tbl
```

1. part1 is a schema, part2 is a table, part3 is a column
2. part1 is a table, part2 is a column, part3 is a property of that column
3. part1 is a column, part2 is a property of that column, part3 is a property of that column

Any extra parts (e.g., .part4.part5 etc) are always treated as properties

Creating Structs with the Row Function The `row` function can be used to automatically convert multiple columns to a single struct column.

> **Note.** This behavior was introduced in DuckDB v0.9.0

When casting structs, the names of fields have to match.

```
SELECT {'a': 42} AS a, a::STRUCT(b INTEGER);
```

Will result in:

```
Mismatch Type Error: Type STRUCT(a INTEGER) does not match with STRUCT(b
↪  INTEGER). Cannot cast STRUCTs with different names
```

Because `row` does not require explicit aliases, it creates an "unnamed struct" which can be cast to any name. `row` can be used to populate a table with a struct column without needing to repeat the field names for every added row. This makes it not a good candidate to use directly in the result of a query, `struct_pack` would be more suited for that.

Example Data Table Named t1

my_column	another_column
1	a
2	b

Row Function Example

```
SELECT
    row(my_column, another_column) AS my_struct_column,
    (my_column, another_column) AS identical_struct_column
FROM t1;
```

Example Output

my_struct_column	identical_struct_column
{'my_column': 1, 'another_column': a}	{'my_column': 1, 'another_column': a}
{'my_column': 2, 'another_column': b}	{'my_column': 2, 'another_column': b}

The `row` function (or simplified parenthesis syntax) may also be used with arbitrary expressions as input rather than column names. In the case of an expression, a key will be automatically generated

in the format of 'vN' where N is a number that refers to its parameter location in the row function (Ex: v1, v2, etc.). This can be combined with column names as an input in the same call to the `row` function. This example uses the same input table as above.

Row Function Example with a Column Name, a Constant, and an Expression as Input

```
SELECT
    row(my_column, 42, my_column + 1) AS my_struct_column,
    (my_column, 42, my_column + 1) AS identical_struct_column
FROM t1;
```

Example Output

my_struct_column	identical_struct_column
{'my_column': 1, 'v2': 42, 'v3': 2}	{'my_column': 1, 'v2': 42, 'v3': 2}
{'my_column': 2, 'v2': 42, 'v3': 3}	{'my_column': 2, 'v2': 42, 'v3': 3}

Comparison Operators

Nested types can be compared using all the comparison operators. These comparisons can be used in logical expressions for both `WHERE` and `HAVING` clauses, as well as for creating `BOOLEAN` values.

The ordering is defined positionally in the same way that words can be ordered in a dictionary. `NULL` values compare greater than all other values and are considered equal to each other.

At the top level, `NULL` nested values obey standard SQL `NULL` comparison rules: comparing a `NULL` nested value to a non-`NULL` nested value produces a `NULL` result. Comparing nested value *members*, however, uses the internal nested value rules for `NULL`s, and a `NULL` nested value member will compare above a non-`NULL` nested value member.

Functions

See Nested Functions.

Text Types

In DuckDB, strings can be stored in the VARCHAR field.

Name	Aliases	Description
VARCHAR	CHAR, BPCHAR, TEXT, STRING	variable-length character string
VARCHAR(n)		variable-length character string with maximum length n

It is possible to supply a number along with the type by initializing a type as VARCHAR(n), where n is a positive integer. **Note that specifying this length is not required and has no effect on the system. Specifying this length will not improve performance or reduce storage space of the strings in the database.** This variant is supported for compatibility reasons with other systems that do require a length to be specified for strings.

If you wish to restrict the number of characters in a VARCHAR column for data integrity reasons the CHECK constraint should be used, for example:

```
CREATE TABLE strings(
    val VARCHAR CHECK(LENGTH(val) <= 10) -- val has a maximum length of 10
 ↪  characters
);
```

The VARCHAR field allows storage of Unicode characters. Internally, the data is encoded as UTF-8.

Formatting Strings

Strings in DuckDB are surrounded by single quote (apostrophe) characters (').

```
SELECT 'Hello world' AS msg;
```

```
┌─────────────┐
│     msg     │
│   varchar   │
├─────────────┤
│ Hello world │
└─────────────┘
```

To include a single quote character in a string, use '':

```
SELECT 'Hello ''world''' AS msg;
```

```
┌─────────────┐
│     msg     │
```

```
┌──────────────────┐
│     varchar      │
├──────────────────┤
│ Hello 'world'    │
└──────────────────┘
```

To include special characters such as newline, use the `chr` character function:

```
SELECT 'Hello' || chr(10) || 'world' AS msg;
```

```
┌──────────────────┐
│       msg        │
│     varchar      │
├──────────────────┤
│ Hello\nworld     │
└──────────────────┘
```

Double Quote Characters

Double quote characters (`"`) are used to denote table and column names. Surrounding their names allows the use of keywords, e.g.:

```
CREATE TABLE "table"("order" BIGINT);
```

While DuckDB occasionally accepts both single quote and double quotes for strings (e.g., both `FROM "filename.csv"` and `FROM 'filename.csv'` work), their use is not recommended.

Functions

See Character Functions and Pattern Matching.

Time Types

The `TIME` and `TIMETZ` types specify the hour, minute, second, microsecond of a day.

Name	Aliases	Description
`TIME`	`TIME WITHOUT TIME ZONE`	time of day (ignores time zone)
`TIMETZ`	`TIME WITH TIME ZONE`	time of day (uses time zone)

Instances can be created using the type names as a keyword, where the data must be formatted according to the ISO 8601 format (`hh:mm:ss[.zzzzzz][+-TT[:tt]]`).

```
SELECT TIME   '1992-09-20 11:30:00.123456';       -- 11:30:00.123456
SELECT TIMETZ '1992-09-20 11:30:00.123456';       -- 11:30:00.123456+00
SELECT TIMETZ '1992-09-20 11:30:00.123456-02:00'; -- 13:30:00.123456+00
SELECT TIMETZ '1992-09-20 11:30:00.123456+05:30'; -- 06:00:00.123456+00
```

> **Note.** The `TIME` type should only be used in rare cases, where the date part of the timestamp can be disregarded. Most applications should use the `TIMESTAMP` types to represent their timestamps.

Timestamp Types

Timestamps represent points in absolute time, usually called *instants*. DuckDB represents instants as the number of microseconds (µs) since `1970-01-01 00:00:00+00`.

Timestamp types

Name	Aliases	Description
`TIMESTAMP_NS`	`TIMESTAMP`, `DATETIME`	timestamp with nanosecond precision (ignores time zone)
`TIMESTAMP_MS`		timestamp with millisecond precision (ignores time zone)
`TIMESTAMP_S`		timestamp with second precision (ignores time zone)
`TIMESTAMPTZ`	`TIMESTAMP WITH TIME ZONE`	timestamp (uses time zone)

A timestamp specifies a combination of `DATE` (year, month, day) and a `TIME` (hour, minute, second, microsecond). Timestamps can be created using the `TIMESTAMP` keyword, where the data must be formatted according to the ISO 8601 format (`YYYY-MM-DD hh:mm:ss[.zzzzzz][+-TT[:tt]]`).

```
SELECT TIMESTAMP_NS '1992-09-20 11:30:00.123456'; -- 1992-09-20
↪ 11:30:00.123456
SELECT TIMESTAMP    '1992-09-20 11:30:00.123456'; -- 1992-09-20
↪ 11:30:00.123456
```

```
SELECT DATETIME      '1992-09-20 11:30:00.123456'; -- 1992-09-20
 ↪  11:30:00.123456
SELECT TIMESTAMP_MS '1992-09-20 11:30:00.123456'; -- 1992-09-20 11:30:00.123
SELECT TIMESTAMP_S  '1992-09-20 11:30:00.123456'; -- 1992-09-20 11:30:00
SELECT TIMESTAMPTZ  '1992-09-20 11:30:00.123456'; -- 1992-09-20
 ↪  11:30:00.123456+00
SELECT TIMESTAMP WITH TIME ZONE '1992-09-20 11:30:00.123456';
-- 1992-09-20 11:30:00.123456+00
```

Special Values

There are also three special date values that can be used on input:

Input String	Valid Types	Description
epoch	`TIMESTAMP`, `TIMESTAMPTZ`	1970-01-01 00:00:00+00 (Unix system time zero)
infinity	`TIMESTAMP`, `TIMESTAMPTZ`	later than all other time stamps
-infinity	`TIMESTAMP`, `TIMESTAMPTZ`	earlier than all other time stamps

The values `infinity` and `-infinity` are specially represented inside the system and will be displayed unchanged; but `epoch` is simply a notational shorthand that will be converted to the time stamp value when read.

```
SELECT '-infinity'::TIMESTAMP, 'epoch'::TIMESTAMP, 'infinity'::TIMESTAMP;
```

Negative	Epoch	Positive
-infinity	1970-01-01 00:00:00	infinity

Functions

See Timestamp Functions.

Time Zones

The `TIMESTAMPTZ` type can be binned into calendar and clock bins using a suitable extension. The built-in ICU extension implements all the binning and arithmetic functions using the International

Components for Unicode time zone and calendar functions.

To set the time zone to use, first load the ICU extension. The ICU extension comes pre-bundled with several DuckDB clients (including Python, R, JDBC, and ODBC), so this step can be skipped in those cases. In other cases you might first need to install and load the ICU extension.

```
INSTALL icu;
LOAD icu;
```

Next, use the `SET TimeZone` command:

```
SET TimeZone='America/Los_Angeles';
```

Time binning operations for `TIMESTAMPTZ` will then be implemented using the given time zone.

A list of available time zones can be pulled from the `pg_timezone_names()` table function:

```
SELECT
    name,
    abbrev,
    utc_offset
FROM pg_timezone_names()
ORDER BY
    name;
```

You can also find a reference table of available time zones here.

Calendars

The ICU extension also supports non-Gregorian calendars using the `SET Calendar` command. Note that the `INSTALL` and `LOAD` steps are only required if the DuckDB client does not bundle the ICU extension.

```
INSTALL ICU;
LOAD icu;
SET Calendar='japanese';
```

Time binning operations for `TIMESTAMPTZ` will then be implemented using the given calendar. In this example, the `era` part will now report the Japanese imperial era number.

A list of available calendars can be pulled from the `icu_calendar_names()` table function:

```
SELECT name FROM icu_calendar_names() ORDER BY 1;
```

Settings

The current value of the `TimeZone` and `Calendar` settings are determined by ICU when it starts up. They can be looked from in the `duckdb_settings()` table function:

```
SELECT * FROM duckdb_settings() WHERE name = 'TimeZone';
-- America/Los_Angeles
SELECT * FROM duckdb_settings() WHERE name = 'Calendar';
-- gregorian
```

Time Zones

Time Zone Reference List

An up-to-date version of this list can be pulled from the `pg_timezone_names()` table function:

```
SELECT
    name,
    abbrev,
    utc_offset
FROM pg_timezone_names()
ORDER BY
    name;
```

name	abbrev	utc_offset
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
Africa/Abidjan	Iceland	00:00:00
Africa/Accra	Iceland	00:00:00
Africa/Addis_Ababa	EAT	03:00:00
Africa/Algiers	Africa/Algiers	01:00:00
Africa/Asmara	EAT	03:00:00
Africa/Asmera	EAT	03:00:00

name	abbrev	utc_offset
Africa/Bamako	Iceland	00:00:00
Africa/Bangui	Africa/Bangui	01:00:00
Africa/Banjul	Iceland	00:00:00
Africa/Bissau	Africa/Bissau	00:00:00
Africa/Blantyre	CAT	02:00:00
Africa/Brazzaville	Africa/Brazzaville	01:00:00
Africa/Bujumbura	CAT	02:00:00
Africa/Cairo	ART	02:00:00
Africa/Casablanca	Africa/Casablanca	00:00:00
Africa/Ceuta	Africa/Ceuta	01:00:00
Africa/Conakry	Iceland	00:00:00
Africa/Dakar	Iceland	00:00:00
Africa/Dar_es_Salaam	EAT	03:00:00
Africa/Djibouti	EAT	03:00:00
Africa/Douala	Africa/Douala	01:00:00
Africa/El_Aaiun	Africa/El_Aaiun	00:00:00
Africa/Freetown	Iceland	00:00:00
Africa/Gaborone	CAT	02:00:00
Africa/Harare	CAT	02:00:00
Africa/Johannesburg	Africa/Johannesburg	02:00:00
Africa/Juba	Africa/Juba	02:00:00
Africa/Kampala	EAT	03:00:00
Africa/Khartoum	Africa/Khartoum	02:00:00
Africa/Kigali	CAT	02:00:00
Africa/Kinshasa	Africa/Kinshasa	01:00:00
Africa/Lagos	Africa/Lagos	01:00:00
Africa/Libreville	Africa/Libreville	01:00:00

name	abbrev	utc_offset
Africa/Lome	Iceland	00:00:00
Africa/Luanda	Africa/Luanda	01:00:00
Africa/Lubumbashi	CAT	02:00:00
Africa/Lusaka	CAT	02:00:00
Africa/Malabo	Africa/Malabo	01:00:00
Africa/Maputo	CAT	02:00:00
Africa/Maseru	Africa/Maseru	02:00:00
Africa/Mbabane	Africa/Mbabane	02:00:00
Africa/Mogadishu	EAT	03:00:00
Africa/Monrovia	Africa/Monrovia	00:00:00
Africa/Nairobi	EAT	03:00:00
Africa/Ndjamena	Africa/Ndjamena	01:00:00
Africa/Niamey	Africa/Niamey	01:00:00
Africa/Nouakchott	Iceland	00:00:00
Africa/Ouagadougou	Iceland	00:00:00
Africa/Porto-Novo	Africa/Porto-Novo	01:00:00
Africa/Sao_Tome	Africa/Sao_Tome	00:00:00
Africa/Timbuktu	Iceland	00:00:00
Africa/Tripoli	Libya	02:00:00
Africa/Tunis	Africa/Tunis	01:00:00
Africa/Windhoek	Africa/Windhoek	02:00:00
America/Adak	America/Adak	-10:00:00
America/Anchorage	AST	-09:00:00
America/Anguilla	PRT	-04:00:00
America/Antigua	PRT	-04:00:00
America/Araguaina	America/Araguaina	-03:00:00
America/Argentina/Buenos_Aires	AGT	-03:00:00

name	abbrev	utc_offset
America/Argentina/Catamarca	America/Argentina/Catamarca	-03:00:00
America/Argentina/ComodRivadavia	America/Argentina/ComodRivadavia	-03:00:00
America/Argentina/Cordoba	America/Argentina/Cordoba	-03:00:00
America/Argentina/Jujuy	America/Argentina/Jujuy	-03:00:00
America/Argentina/La_Rioja	America/Argentina/La_Rioja	-03:00:00
America/Argentina/Mendoza	America/Argentina/Mendoza	-03:00:00
America/Argentina/Rio_Gallegos	America/Argentina/Rio_Gallegos	-03:00:00
America/Argentina/Salta	America/Argentina/Salta	-03:00:00
America/Argentina/San_Juan	America/Argentina/San_Juan	-03:00:00
America/Argentina/San_Luis	America/Argentina/San_Luis	-03:00:00
America/Argentina/Tucuman	America/Argentina/Tucuman	-03:00:00
America/Argentina/Ushuaia	America/Argentina/Ushuaia	-03:00:00
America/Aruba	PRT	-04:00:00
America/Asuncion	America/Asuncion	-04:00:00
America/Atikokan	America/Atikokan	-05:00:00
America/Atka	America/Atka	-10:00:00
America/Bahia	America/Bahia	-03:00:00
America/Bahia_Banderas	America/Bahia_Banderas	-06:00:00
America/Barbados	America/Barbados	-04:00:00
America/Belem	America/Belem	-03:00:00
America/Belize	America/Belize	-06:00:00
America/Blanc-Sablon	PRT	-04:00:00
America/Boa_Vista	America/Boa_Vista	-04:00:00
America/Bogota	America/Bogota	-05:00:00
America/Boise	America/Boise	-07:00:00
America/Buenos_Aires	AGT	-03:00:00
America/Cambridge_Bay	America/Cambridge_Bay	-07:00:00

name	abbrev	utc_offset
America/Campo_Grande	America/Campo_Grande	-04:00:00
America/Cancun	America/Cancun	-05:00:00
America/Caracas	America/Caracas	-04:00:00
America/Catamarca	America/Catamarca	-03:00:00
America/Cayenne	America/Cayenne	-03:00:00
America/Cayman	America/Cayman	-05:00:00
America/Chicago	CST	-06:00:00
America/Chihuahua	America/Chihuahua	-06:00:00
America/Coral_Harbour	America/Coral_Harbour	-05:00:00
America/Cordoba	America/Cordoba	-03:00:00
America/Costa_Rica	America/Costa_Rica	-06:00:00
America/Creston	PNT	-07:00:00
America/Cuiaba	America/Cuiaba	-04:00:00
America/Curacao	PRT	-04:00:00
America/Danmarkshavn	America/Danmarkshavn	00:00:00
America/Dawson	America/Dawson	-07:00:00
America/Dawson_Creek	America/Dawson_Creek	-07:00:00
America/Denver	Navajo	-07:00:00
America/Detroit	America/Detroit	-05:00:00
America/Dominica	PRT	-04:00:00
America/Edmonton	America/Edmonton	-07:00:00
America/Eirunepe	America/Eirunepe	-05:00:00
America/El_Salvador	America/El_Salvador	-06:00:00
America/Ensenada	America/Ensenada	-08:00:00
America/Fort_Nelson	America/Fort_Nelson	-07:00:00
America/Fort_Wayne	IET	-05:00:00
America/Fortaleza	America/Fortaleza	-03:00:00

name	abbrev	utc_offset
America/Glace_Bay	America/Glace_Bay	-04:00:00
America/Godthab	America/Godthab	-02:00:00
America/Goose_Bay	America/Goose_Bay	-04:00:00
America/Grand_Turk	America/Grand_Turk	-05:00:00
America/Grenada	PRT	-04:00:00
America/Guadeloupe	PRT	-04:00:00
America/Guatemala	America/Guatemala	-06:00:00
America/Guayaquil	America/Guayaquil	-05:00:00
America/Guyana	America/Guyana	-04:00:00
America/Halifax	America/Halifax	-04:00:00
America/Havana	Cuba	-05:00:00
America/Hermosillo	America/Hermosillo	-07:00:00
America/Indiana/Indianapolis	IET	-05:00:00
America/Indiana/Knox	America/Indiana/Knox	-06:00:00
America/Indiana/Marengo	America/Indiana/Marengo	-05:00:00
America/Indiana/Petersburg	America/Indiana/Petersburg	-05:00:00
America/Indiana/Tell_City	America/Indiana/Tell_City	-06:00:00
America/Indiana/Vevay	America/Indiana/Vevay	-05:00:00
America/Indiana/Vincennes	America/Indiana/Vincennes	-05:00:00
America/Indiana/Winamac	America/Indiana/Winamac	-05:00:00
America/Indianapolis	IET	-05:00:00
America/Inuvik	America/Inuvik	-07:00:00
America/Iqaluit	America/Iqaluit	-05:00:00
America/Jamaica	Jamaica	-05:00:00
America/Jujuy	America/Jujuy	-03:00:00
America/Juneau	America/Juneau	-09:00:00
America/Kentucky/Louisville	America/Kentucky/Louisville	-05:00:00

name	abbrev	utc_offset
America/Kentucky/Monticello	America/Kentucky/Monticello	-05:00:00
America/Knox_IN	America/Knox_IN	-06:00:00
America/Kralendijk	PRT	-04:00:00
America/La_Paz	America/La_Paz	-04:00:00
America/Lima	America/Lima	-05:00:00
America/Los_Angeles	PST	-08:00:00
America/Louisville	America/Louisville	-05:00:00
America/Lower_Princes	PRT	-04:00:00
America/Maceio	America/Maceio	-03:00:00
America/Managua	America/Managua	-06:00:00
America/Manaus	America/Manaus	-04:00:00
America/Marigot	PRT	-04:00:00
America/Martinique	America/Martinique	-04:00:00
America/Matamoros	America/Matamoros	-06:00:00
America/Mazatlan	America/Mazatlan	-07:00:00
America/Mendoza	America/Mendoza	-03:00:00
America/Menominee	America/Menominee	-06:00:00
America/Merida	America/Merida	-06:00:00
America/Metlakatla	America/Metlakatla	-09:00:00
America/Mexico_City	America/Mexico_City	-06:00:00
America/Miquelon	America/Miquelon	-03:00:00
America/Moncton	America/Moncton	-04:00:00
America/Monterrey	America/Monterrey	-06:00:00
America/Montevideo	America/Montevideo	-03:00:00
America/Montreal	America/Montreal	-05:00:00
America/Montserrat	PRT	-04:00:00
America/Nassau	America/Nassau	-05:00:00

name	abbrev	utc_offset
America/New_York	America/New_York	-05:00:00
America/Nipigon	America/Nipigon	-05:00:00
America/Nome	America/Nome	-09:00:00
America/Noronha	America/Noronha	-02:00:00
America/North_Dakota/Beulah	America/North_Dakota/Beulah	-06:00:00
America/North_Dakota/Center	America/North_Dakota/Center	-06:00:00
America/North_Dakota/New_Salem	America/North_Dakota/New_Salem	-06:00:00
America/Nuuk	America/Nuuk	-02:00:00
America/Ojinaga	America/Ojinaga	-06:00:00
America/Panama	America/Panama	-05:00:00
America/Pangnirtung	America/Pangnirtung	-05:00:00
America/Paramaribo	America/Paramaribo	-03:00:00
America/Phoenix	PNT	-07:00:00
America/Port-au-Prince	America/Port-au-Prince	-05:00:00
America/Port_of_Spain	PRT	-04:00:00
America/Porto_Acre	America/Porto_Acre	-05:00:00
America/Porto_Velho	America/Porto_Velho	-04:00:00
America/Puerto_Rico	PRT	-04:00:00
America/Punta_Arenas	America/Punta_Arenas	-03:00:00
America/Rainy_River	America/Rainy_River	-06:00:00
America/Rankin_Inlet	America/Rankin_Inlet	-06:00:00
America/Recife	America/Recife	-03:00:00
America/Regina	America/Regina	-06:00:00
America/Resolute	America/Resolute	-06:00:00
America/Rio_Branco	America/Rio_Branco	-05:00:00
America/Rosario	America/Rosario	-03:00:00
America/Santa_Isabel	America/Santa_Isabel	-08:00:00

name	abbrev	utc_offset
America/Santarem	America/Santarem	-03:00:00
America/Santiago	America/Santiago	-04:00:00
America/Santo_Domingo	America/Santo_Domingo	-04:00:00
America/Sao_Paulo	BET	-03:00:00
America/Scoresbysund	America/Scoresbysund	-01:00:00
America/Shiprock	Navajo	-07:00:00
America/Sitka	America/Sitka	-09:00:00
America/St_Barthelemy	PRT	-04:00:00
America/St_Johns	CNT	-03:30:00
America/St_Kitts	PRT	-04:00:00
America/St_Lucia	PRT	-04:00:00
America/St_Thomas	PRT	-04:00:00
America/St_Vincent	PRT	-04:00:00
America/Swift_Current	America/Swift_Current	-06:00:00
America/Tegucigalpa	America/Tegucigalpa	-06:00:00
America/Thule	America/Thule	-04:00:00
America/Thunder_Bay	America/Thunder_Bay	-05:00:00
America/Tijuana	America/Tijuana	-08:00:00
America/Toronto	America/Toronto	-05:00:00
America/Tortola	PRT	-04:00:00
America/Vancouver	America/Vancouver	-08:00:00
America/Virgin	PRT	-04:00:00
America/Whitehorse	America/Whitehorse	-07:00:00
America/Winnipeg	America/Winnipeg	-06:00:00
America/Yakutat	America/Yakutat	-09:00:00
America/Yellowknife	America/Yellowknife	-07:00:00
Antarctica/Casey	Antarctica/Casey	11:00:00

name	abbrev	utc_offset
Antarctica/Davis	Antarctica/Davis	07:00:00
Antarctica/DumontDUrville	Antarctica/DumontDUrville	10:00:00
Antarctica/Macquarie	Antarctica/Macquarie	10:00:00
Antarctica/Mawson	Antarctica/Mawson	05:00:00
Antarctica/McMurdo	NZ	12:00:00
Antarctica/Palmer	Antarctica/Palmer	-03:00:00
Antarctica/Rothera	Antarctica/Rothera	-03:00:00
Antarctica/South_Pole	NZ	12:00:00
Antarctica/Syowa	Antarctica/Syowa	03:00:00
Antarctica/Troll	Antarctica/Troll	00:00:00
Antarctica/Vostok	Antarctica/Vostok	06:00:00
Arctic/Longyearbyen	Arctic/Longyearbyen	01:00:00
Asia/Aden	Asia/Aden	03:00:00
Asia/Almaty	Asia/Almaty	06:00:00
Asia/Amman	Asia/Amman	03:00:00
Asia/Anadyr	Asia/Anadyr	12:00:00
Asia/Aqtau	Asia/Aqtau	05:00:00
Asia/Aqtobe	Asia/Aqtobe	05:00:00
Asia/Ashgabat	Asia/Ashgabat	05:00:00
Asia/Ashkhabad	Asia/Ashkhabad	05:00:00
Asia/Atyrau	Asia/Atyrau	05:00:00
Asia/Baghdad	Asia/Baghdad	03:00:00
Asia/Bahrain	Asia/Bahrain	03:00:00
Asia/Baku	Asia/Baku	04:00:00
Asia/Bangkok	Asia/Bangkok	07:00:00
Asia/Barnaul	Asia/Barnaul	07:00:00
Asia/Beirut	Asia/Beirut	02:00:00

name	abbrev	utc_offset
Asia/Bishkek	Asia/Bishkek	06:00:00
Asia/Brunei	Asia/Brunei	08:00:00
Asia/Calcutta	IST	05:30:00
Asia/Chita	Asia/Chita	09:00:00
Asia/Choibalsan	Asia/Choibalsan	08:00:00
Asia/Chongqing	CTT	08:00:00
Asia/Chungking	CTT	08:00:00
Asia/Colombo	Asia/Colombo	05:30:00
Asia/Dacca	BST	06:00:00
Asia/Damascus	Asia/Damascus	03:00:00
Asia/Dhaka	BST	06:00:00
Asia/Dili	Asia/Dili	09:00:00
Asia/Dubai	Asia/Dubai	04:00:00
Asia/Dushanbe	Asia/Dushanbe	05:00:00
Asia/Famagusta	Asia/Famagusta	02:00:00
Asia/Gaza	Asia/Gaza	02:00:00
Asia/Harbin	CTT	08:00:00
Asia/Hebron	Asia/Hebron	02:00:00
Asia/Ho_Chi_Minh	VST	07:00:00
Asia/Hong_Kong	Hongkong	08:00:00
Asia/Hovd	Asia/Hovd	07:00:00
Asia/Irkutsk	Asia/Irkutsk	08:00:00
Asia/Istanbul	Turkey	03:00:00
Asia/Jakarta	Asia/Jakarta	07:00:00
Asia/Jayapura	Asia/Jayapura	09:00:00
Asia/Jerusalem	Israel	02:00:00
Asia/Kabul	Asia/Kabul	04:30:00

name	abbrev	utc_offset
Asia/Kamchatka	Asia/Kamchatka	12:00:00
Asia/Karachi	PLT	05:00:00
Asia/Kashgar	Asia/Kashgar	06:00:00
Asia/Kathmandu	Asia/Kathmandu	05:45:00
Asia/Katmandu	Asia/Katmandu	05:45:00
Asia/Khandyga	Asia/Khandyga	09:00:00
Asia/Kolkata	IST	05:30:00
Asia/Krasnoyarsk	Asia/Krasnoyarsk	07:00:00
Asia/Kuala_Lumpur	Singapore	08:00:00
Asia/Kuching	Asia/Kuching	08:00:00
Asia/Kuwait	Asia/Kuwait	03:00:00
Asia/Macao	Asia/Macao	08:00:00
Asia/Macau	Asia/Macau	08:00:00
Asia/Magadan	Asia/Magadan	11:00:00
Asia/Makassar	Asia/Makassar	08:00:00
Asia/Manila	Asia/Manila	08:00:00
Asia/Muscat	Asia/Muscat	04:00:00
Asia/Nicosia	Asia/Nicosia	02:00:00
Asia/Novokuznetsk	Asia/Novokuznetsk	07:00:00
Asia/Novosibirsk	Asia/Novosibirsk	07:00:00
Asia/Omsk	Asia/Omsk	06:00:00
Asia/Oral	Asia/Oral	05:00:00
Asia/Phnom_Penh	Asia/Phnom_Penh	07:00:00
Asia/Pontianak	Asia/Pontianak	07:00:00
Asia/Pyongyang	Asia/Pyongyang	09:00:00
Asia/Qatar	Asia/Qatar	03:00:00
Asia/Qostanay	Asia/Qostanay	06:00:00

name	abbrev	utc_offset
Asia/Qyzylorda	Asia/Qyzylorda	05:00:00
Asia/Rangoon	Asia/Rangoon	06:30:00
Asia/Riyadh	Asia/Riyadh	03:00:00
Asia/Saigon	VST	07:00:00
Asia/Sakhalin	Asia/Sakhalin	11:00:00
Asia/Samarkand	Asia/Samarkand	05:00:00
Asia/Seoul	ROK	09:00:00
Asia/Shanghai	CTT	08:00:00
Asia/Singapore	Singapore	08:00:00
Asia/Srednekolymsk	Asia/Srednekolymsk	11:00:00
Asia/Taipei	ROC	08:00:00
Asia/Tashkent	Asia/Tashkent	05:00:00
Asia/Tbilisi	Asia/Tbilisi	04:00:00
Asia/Tehran	Iran	03:30:00
Asia/Tel_Aviv	Israel	02:00:00
Asia/Thimbu	Asia/Thimbu	06:00:00
Asia/Thimphu	Asia/Thimphu	06:00:00
Asia/Tokyo	JST	09:00:00
Asia/Tomsk	Asia/Tomsk	07:00:00
Asia/Ujung_Pandang	Asia/Ujung_Pandang	08:00:00
Asia/Ulaanbaatar	Asia/Ulaanbaatar	08:00:00
Asia/Ulan_Bator	Asia/Ulan_Bator	08:00:00
Asia/Urumqi	Asia/Urumqi	06:00:00
Asia/Ust-Nera	Asia/Ust-Nera	10:00:00
Asia/Vientiane	Asia/Vientiane	07:00:00
Asia/Vladivostok	Asia/Vladivostok	10:00:00
Asia/Yakutsk	Asia/Yakutsk	09:00:00

name	abbrev	utc_offset
Asia/Yangon	Asia/Yangon	06:30:00
Asia/Yekaterinburg	Asia/Yekaterinburg	05:00:00
Asia/Yerevan	NET	04:00:00
Atlantic/Azores	Atlantic/Azores	-01:00:00
Atlantic/Bermuda	Atlantic/Bermuda	-04:00:00
Atlantic/Canary	Atlantic/Canary	00:00:00
Atlantic/Cape_Verde	Atlantic/Cape_Verde	-01:00:00
Atlantic/Faeroe	Atlantic/Faeroe	00:00:00
Atlantic/Faroe	Atlantic/Faroe	00:00:00
Atlantic/Jan_Mayen	Atlantic/Jan_Mayen	01:00:00
Atlantic/Madeira	Atlantic/Madeira	00:00:00
Atlantic/Reykjavik	Iceland	00:00:00
Atlantic/South_Georgia	Atlantic/South_Georgia	-02:00:00
Atlantic/St_Helena	Iceland	00:00:00
Atlantic/Stanley	Atlantic/Stanley	-03:00:00
Australia/ACT	AET	10:00:00
Australia/Adelaide	Australia/Adelaide	09:30:00
Australia/Brisbane	Australia/Brisbane	10:00:00
Australia/Broken_Hill	Australia/Broken_Hill	09:30:00
Australia/Canberra	AET	10:00:00
Australia/Currie	Australia/Currie	10:00:00
Australia/Darwin	ACT	09:30:00
Australia/Eucla	Australia/Eucla	08:45:00
Australia/Hobart	Australia/Hobart	10:00:00
Australia/LHI	Australia/LHI	10:30:00
Australia/Lindeman	Australia/Lindeman	10:00:00
Australia/Lord_Howe	Australia/Lord_Howe	10:30:00

name	abbrev	utc_offset
Australia/Melbourne	Australia/Melbourne	10:00:00
Australia/NSW	AET	10:00:00
Australia/North	ACT	09:30:00
Australia/Perth	Australia/Perth	08:00:00
Australia/Queensland	Australia/Queensland	10:00:00
Australia/South	Australia/South	09:30:00
Australia/Sydney	AET	10:00:00
Australia/Tasmania	Australia/Tasmania	10:00:00
Australia/Victoria	Australia/Victoria	10:00:00
Australia/West	Australia/West	08:00:00
Australia/Yancowinna	Australia/Yancowinna	09:30:00
BET	BET	-03:00:00
BST	BST	06:00:00
Brazil/Acre	Brazil/Acre	-05:00:00
Brazil/DeNoronha	Brazil/DeNoronha	-02:00:00
Brazil/East	BET	-03:00:00
Brazil/West	Brazil/West	-04:00:00
CAT	CAT	02:00:00
CET	CET	01:00:00
CNT	CNT	-03:30:00
CST	CST	-06:00:00
CST6CDT	CST6CDT	-06:00:00
CTT	CTT	08:00:00
Canada/Atlantic	Canada/Atlantic	-04:00:00
Canada/Central	Canada/Central	-06:00:00
Canada/East-Saskatchewan	Canada/East-Saskatchewan	-06:00:00
Canada/Eastern	Canada/Eastern	-05:00:00

name	abbrev	utc_offset
Canada/Mountain	Canada/Mountain	-07:00:00
Canada/Newfoundland	CNT	-03:30:00
Canada/Pacific	Canada/Pacific	-08:00:00
Canada/Saskatchewan	Canada/Saskatchewan	-06:00:00
Canada/Yukon	Canada/Yukon	-07:00:00
Chile/Continental	Chile/Continental	-04:00:00
Chile/EasterIsland	Chile/EasterIsland	-06:00:00
Cuba	Cuba	-05:00:00
EAT	EAT	03:00:00
ECT	ECT	01:00:00
EET	EET	02:00:00
EST	EST	-05:00:00
EST5EDT	EST5EDT	-05:00:00
Egypt	ART	02:00:00
Eire	Eire	00:00:00
Etc/GMT	GMT	00:00:00
Etc/GMT+0	GMT	00:00:00
Etc/GMT+1	Etc/GMT+1	-01:00:00
Etc/GMT+10	Etc/GMT+10	-10:00:00
Etc/GMT+11	Etc/GMT+11	-11:00:00
Etc/GMT+12	Etc/GMT+12	-12:00:00
Etc/GMT+2	Etc/GMT+2	-02:00:00
Etc/GMT+3	Etc/GMT+3	-03:00:00
Etc/GMT+4	Etc/GMT+4	-04:00:00
Etc/GMT+5	Etc/GMT+5	-05:00:00
Etc/GMT+6	Etc/GMT+6	-06:00:00
Etc/GMT+7	Etc/GMT+7	-07:00:00

name	abbrev	utc_offset
Etc/GMT+8	Etc/GMT+8	-08:00:00
Etc/GMT+9	Etc/GMT+9	-09:00:00
Etc/GMT-0	GMT	00:00:00
Etc/GMT-1	Etc/GMT-1	01:00:00
Etc/GMT-10	Etc/GMT-10	10:00:00
Etc/GMT-11	Etc/GMT-11	11:00:00
Etc/GMT-12	Etc/GMT-12	12:00:00
Etc/GMT-13	Etc/GMT-13	13:00:00
Etc/GMT-14	Etc/GMT-14	14:00:00
Etc/GMT-2	Etc/GMT-2	02:00:00
Etc/GMT-3	Etc/GMT-3	03:00:00
Etc/GMT-4	Etc/GMT-4	04:00:00
Etc/GMT-5	Etc/GMT-5	05:00:00
Etc/GMT-6	Etc/GMT-6	06:00:00
Etc/GMT-7	Etc/GMT-7	07:00:00
Etc/GMT-8	Etc/GMT-8	08:00:00
Etc/GMT-9	Etc/GMT-9	09:00:00
Etc/GMT0	GMT	00:00:00
Etc/Greenwich	GMT	00:00:00
Etc/UCT	UCT	00:00:00
Etc/UTC	UCT	00:00:00
Etc/Universal	UCT	00:00:00
Etc/Zulu	UCT	00:00:00
Europe/Amsterdam	Europe/Amsterdam	01:00:00
Europe/Andorra	Europe/Andorra	01:00:00
Europe/Astrakhan	Europe/Astrakhan	04:00:00
Europe/Athens	Europe/Athens	02:00:00

name	abbrev	utc_offset
Europe/Belfast	GB	00:00:00
Europe/Belgrade	Europe/Belgrade	01:00:00
Europe/Berlin	Europe/Berlin	01:00:00
Europe/Bratislava	Europe/Bratislava	01:00:00
Europe/Brussels	Europe/Brussels	01:00:00
Europe/Bucharest	Europe/Bucharest	02:00:00
Europe/Budapest	Europe/Budapest	01:00:00
Europe/Busingen	Europe/Busingen	01:00:00
Europe/Chisinau	Europe/Chisinau	02:00:00
Europe/Copenhagen	Europe/Copenhagen	01:00:00
Europe/Dublin	Eire	00:00:00
Europe/Gibraltar	Europe/Gibraltar	01:00:00
Europe/Guernsey	GB	00:00:00
Europe/Helsinki	Europe/Helsinki	02:00:00
Europe/Isle_of_Man	GB	00:00:00
Europe/Istanbul	Turkey	03:00:00
Europe/Jersey	GB	00:00:00
Europe/Kaliningrad	Europe/Kaliningrad	02:00:00
Europe/Kiev	Europe/Kiev	02:00:00
Europe/Kirov	Europe/Kirov	03:00:00
Europe/Kyiv	Europe/Kyiv	02:00:00
Europe/Lisbon	Portugal	00:00:00
Europe/Ljubljana	Europe/Ljubljana	01:00:00
Europe/London	GB	00:00:00
Europe/Luxembourg	Europe/Luxembourg	01:00:00
Europe/Madrid	Europe/Madrid	01:00:00
Europe/Malta	Europe/Malta	01:00:00

name	abbrev	utc_offset
Europe/Mariehamn	Europe/Mariehamn	02:00:00
Europe/Minsk	Europe/Minsk	03:00:00
Europe/Monaco	ECT	01:00:00
Europe/Moscow	W-SU	03:00:00
Europe/Nicosia	Europe/Nicosia	02:00:00
Europe/Oslo	Europe/Oslo	01:00:00
Europe/Paris	ECT	01:00:00
Europe/Podgorica	Europe/Podgorica	01:00:00
Europe/Prague	Europe/Prague	01:00:00
Europe/Riga	Europe/Riga	02:00:00
Europe/Rome	Europe/Rome	01:00:00
Europe/Samara	Europe/Samara	04:00:00
Europe/San_Marino	Europe/San_Marino	01:00:00
Europe/Sarajevo	Europe/Sarajevo	01:00:00
Europe/Saratov	Europe/Saratov	04:00:00
Europe/Simferopol	Europe/Simferopol	03:00:00
Europe/Skopje	Europe/Skopje	01:00:00
Europe/Sofia	Europe/Sofia	02:00:00
Europe/Stockholm	Europe/Stockholm	01:00:00
Europe/Tallinn	Europe/Tallinn	02:00:00
Europe/Tirane	Europe/Tirane	01:00:00
Europe/Tiraspol	Europe/Tiraspol	02:00:00
Europe/Ulyanovsk	Europe/Ulyanovsk	04:00:00
Europe/Uzhgorod	Europe/Uzhgorod	02:00:00
Europe/Vaduz	Europe/Vaduz	01:00:00
Europe/Vatican	Europe/Vatican	01:00:00
Europe/Vienna	Europe/Vienna	01:00:00

name	abbrev	utc_offset
Europe/Vilnius	Europe/Vilnius	02:00:00
Europe/Volgograd	Europe/Volgograd	03:00:00
Europe/Warsaw	Poland	01:00:00
Europe/Zagreb	Europe/Zagreb	01:00:00
Europe/Zaporozhye	Europe/Zaporozhye	02:00:00
Europe/Zurich	Europe/Zurich	01:00:00
Factory	Factory	00:00:00
GB	GB	00:00:00
GB-Eire	GB	00:00:00
GMT	GMT	00:00:00
GMT+0	GMT	00:00:00
GMT-0	GMT	00:00:00
GMT0	GMT	00:00:00
Greenwich	GMT	00:00:00
HST	HST	-10:00:00
Hongkong	Hongkong	08:00:00
IET	IET	-05:00:00
IST	IST	05:30:00
Iceland	Iceland	00:00:00
Indian/Antananarivo	EAT	03:00:00
Indian/Chagos	Indian/Chagos	06:00:00
Indian/Christmas	Indian/Christmas	07:00:00
Indian/Cocos	Indian/Cocos	06:30:00
Indian/Comoro	EAT	03:00:00
Indian/Kerguelen	Indian/Kerguelen	05:00:00
Indian/Mahe	Indian/Mahe	04:00:00
Indian/Maldives	Indian/Maldives	05:00:00

name	abbrev	utc_offset
Indian/Mauritius	Indian/Mauritius	04:00:00
Indian/Mayotte	EAT	03:00:00
Indian/Reunion	Indian/Reunion	04:00:00
Iran	Iran	03:30:00
Israel	Israel	02:00:00
JST	JST	09:00:00
Jamaica	Jamaica	-05:00:00
Japan	JST	09:00:00
Kwajalein	Kwajalein	12:00:00
Libya	Libya	02:00:00
MET	MET	01:00:00
MIT	MIT	13:00:00
MST	MST	-07:00:00
MST7MDT	MST7MDT	-07:00:00
Mexico/BajaNorte	Mexico/BajaNorte	-08:00:00
Mexico/BajaSur	Mexico/BajaSur	-07:00:00
Mexico/General	Mexico/General	-06:00:00
NET	NET	04:00:00
NST	NZ	12:00:00
NZ	NZ	12:00:00
NZ-CHAT	NZ-CHAT	12:45:00
Navajo	Navajo	-07:00:00
PLT	PLT	05:00:00
PNT	PNT	-07:00:00
PRC	CTT	08:00:00
PRT	PRT	-04:00:00
PST	PST	-08:00:00

name	abbrev	utc_offset
PST8PDT	PST8PDT	-08:00:00
Pacific/Apia	MIT	13:00:00
Pacific/Auckland	NZ	12:00:00
Pacific/Bougainville	Pacific/Bougainville	11:00:00
Pacific/Chatham	NZ-CHAT	12:45:00
Pacific/Chuuk	Pacific/Chuuk	10:00:00
Pacific/Easter	Pacific/Easter	-06:00:00
Pacific/Efate	Pacific/Efate	11:00:00
Pacific/Enderbury	Pacific/Enderbury	13:00:00
Pacific/Fakaofo	Pacific/Fakaofo	13:00:00
Pacific/Fiji	Pacific/Fiji	12:00:00
Pacific/Funafuti	Pacific/Funafuti	12:00:00
Pacific/Galapagos	Pacific/Galapagos	-06:00:00
Pacific/Gambier	Pacific/Gambier	-09:00:00
Pacific/Guadalcanal	SST	11:00:00
Pacific/Guam	Pacific/Guam	10:00:00
Pacific/Honolulu	Pacific/Honolulu	-10:00:00
Pacific/Johnston	Pacific/Johnston	-10:00:00
Pacific/Kanton	Pacific/Kanton	13:00:00
Pacific/Kiritimati	Pacific/Kiritimati	14:00:00
Pacific/Kosrae	Pacific/Kosrae	11:00:00
Pacific/Kwajalein	Kwajalein	12:00:00
Pacific/Majuro	Pacific/Majuro	12:00:00
Pacific/Marquesas	Pacific/Marquesas	-09:30:00
Pacific/Midway	Pacific/Midway	-11:00:00
Pacific/Nauru	Pacific/Nauru	12:00:00
Pacific/Niue	Pacific/Niue	-11:00:00

name	abbrev	utc_offset
Pacific/Norfolk	Pacific/Norfolk	11:00:00
Pacific/Noumea	Pacific/Noumea	11:00:00
Pacific/Pago_Pago	Pacific/Pago_Pago	-11:00:00
Pacific/Palau	Pacific/Palau	09:00:00
Pacific/Pitcairn	Pacific/Pitcairn	-08:00:00
Pacific/Pohnpei	SST	11:00:00
Pacific/Ponape	SST	11:00:00
Pacific/Port_Moresby	Pacific/Port_Moresby	10:00:00
Pacific/Rarotonga	Pacific/Rarotonga	-10:00:00
Pacific/Saipan	Pacific/Saipan	10:00:00
Pacific/Samoa	Pacific/Samoa	-11:00:00
Pacific/Tahiti	Pacific/Tahiti	-10:00:00
Pacific/Tarawa	Pacific/Tarawa	12:00:00
Pacific/Tongatapu	Pacific/Tongatapu	13:00:00
Pacific/Truk	Pacific/Truk	10:00:00
Pacific/Wake	Pacific/Wake	12:00:00
Pacific/Wallis	Pacific/Wallis	12:00:00
Pacific/Yap	Pacific/Yap	10:00:00
Poland	Poland	01:00:00
Portugal	Portugal	00:00:00
ROC	ROC	08:00:00
ROK	ROK	09:00:00
SST	SST	11:00:00
Singapore	Singapore	08:00:00
SystemV/AST4	SystemV/AST4	-04:00:00
SystemV/AST4ADT	SystemV/AST4ADT	-04:00:00
SystemV/CST6	SystemV/CST6	-06:00:00

name	abbrev	utc_offset
SystemV/CST6CDT	SystemV/CST6CDT	-06:00:00
SystemV/EST5	SystemV/EST5	-05:00:00
SystemV/EST5EDT	SystemV/EST5EDT	-05:00:00
SystemV/HST10	SystemV/HST10	-10:00:00
SystemV/MST7	SystemV/MST7	-07:00:00
SystemV/MST7MDT	SystemV/MST7MDT	-07:00:00
SystemV/PST8	SystemV/PST8	-08:00:00
SystemV/PST8PDT	SystemV/PST8PDT	-08:00:00
SystemV/YST9	SystemV/YST9	-09:00:00
SystemV/YST9YDT	SystemV/YST9YDT	-09:00:00
Turkey	Turkey	03:00:00
UCT	UCT	00:00:00
US/Alaska	AST	-09:00:00
US/Aleutian	US/Aleutian	-10:00:00
US/Arizona	PNT	-07:00:00
US/Central	CST	-06:00:00
US/East-Indiana	IET	-05:00:00
US/Eastern	US/Eastern	-05:00:00
US/Hawaii	US/Hawaii	-10:00:00
US/Indiana-Starke	US/Indiana-Starke	-06:00:00
US/Michigan	US/Michigan	-05:00:00
US/Mountain	Navajo	-07:00:00
US/Pacific	PST	-08:00:00
US/Pacific-New	PST	-08:00:00
US/Samoa	US/Samoa	-11:00:00
UTC	UCT	00:00:00
Universal	UCT	00:00:00

name	abbrev	utc_offset
VST	VST	07:00:00
W-SU	W-SU	03:00:00
WET	WET	00:00:00
Zulu	UCT	00:00:00

Union

Union Data Type

A `UNION` *type* (not to be confused with the SQL `UNION` operator) is a nested type capable of holding one of multiple "alternative" values, much like the `union` in C. The main difference being that these `UNION` types are *tagged unions* and thus always carry a discriminator "tag" which signals which alternative it is currently holding, even if the inner value itself is null. `UNION` types are thus more similar to C++17's `std::variant`, Rust's `Enum` or the "sum type" present in most functional languages.

`UNION` types must always have at least one member, and while they can contain multiple members of the same type, the tag names must be unique. `UNION` types can have at most 256 members.

Under the hood, `UNION` types are implemented on top of `STRUCT` types, and simply keep the "tag" as the first entry.

`UNION` values can be created with the `UNION_VALUE(tag := expr)` function or by casting from a member type.

Example

```
-- Create a table with a union column
CREATE TABLE tbl1(u UNION(num INT, str VARCHAR));

-- Any type can be implicitly cast to a union containing the type.
-- Any union can also be implicitly cast to another union if
-- the source union members are a subset of the targets.
-- Note: only if the cast is unambiguous!
-- More details in the 'Union casts' section below.
INSERT INTO tbl1 values (1) , ('two') , (union_value(str := 'three'));
-- Union use the member types varchar cast functions when casting to varchar.
SELECT u FROM tbl1;
-- returns:
```

```
--    1
--    two
--    three
-- Select all the 'str' members
SELECT union_extract(u, 'str') FROM tbl1;
-- Alternatively, you can use 'dot syntax' like with structs
SELECT u.str FROM tbl1;
-- returns:
--    NULL
--    two
--    three

-- Select the currently active tag from the union as an enum.
SELECT union_tag(u) FROM tbl1;
-- returns:
--    num
--    str
--    str
```

Union Casts

Compared to other nested types, `UNION`s allow a set of implicit casts to facilitate unintrusive and natural usage when working with their members as "subtypes". However, these casts have been designed with two principles in mind, to avoid ambiguity and to avoid casts that could lead to loss of information. This prevents `UNION`s from being completely "transparent", while still allowing `UNION` types to have a "supertype" relationship with their members.

Thus `UNION` types can't be implicitly cast to any of their member types in general, since the information in the other members not matching the target type would be "lost". If you want to coerce a `UNION` into one of its members, you should use the `union_extract` function explicitly instead.

The only exception to this is when casting a `UNION` to `VARCHAR`, in which case the members will all use their corresponding `VARCHAR` casts. Since everything can be cast to `VARCHAR`, this is "safe" in a sense.

Casting to Unions A type can always be implicitly cast to a `UNION` if it can be implicitly cast to one of the `UNION` member types.

- If there are multiple candidates, the built in implicit casting priority rules determine the target type. For example, a `FLOAT -> UNION(i INT, v VARCHAR)` cast will always cast the `FLOAT` to the `INT` member before `VARCHAR`.

- If the cast still is ambiguous, i.e., there are multiple candidates with the same implicit casting priority, an error is raised. This usually happens when the `UNION` contains multiple members of the same type, e.g., a `FLOAT -> UNION(i INT, num INT)` is always ambiguous.

So how do we disambiguate if we want to create a `UNION` with multiple members of the same type? By using the `union_value` function, which takes a keyword argument specifying the tag. For example, `union_value(num := 2::INT)` will create a `UNION` with a single member of type `INT` with the tag `num`. This can then be used to disambiguate in an explicit (or implicit, read on below!) `UNION` to `UNION` cast, like `CAST(union_value(b := 2) AS UNION(a INT, b INT))`.

Casting between Unions `UNION` types can be cast between each other if the source type is a "subset" of the target type. In other words, all the tags in the source `UNION` must be present in the target `UNION`, and all the types of the matching tags must be implicitly castable between source and target. In essence, this means that `UNION` types are covariant with respect to their members.

Ok	Source	Target	Comments
✅	UNION(a A, b B)	UNION(a A, b B, c C)	
✅	UNION(a A, b B)	UNION(a A, b C)	if B can be implicitly cast to C
❌	UNION(a A, b B, c C)	UNION(a A, b B)	
❌	UNION(a A, b B)	UNION(a A, b C)	if B can't be implicitly cast to C
❌	UNION(A, B, D)	UNION(A, B, C)	

Comparison and Sorting

Since `UNION` types are implemented on top of `STRUCT` types internally, they can be used with all the comparison operators as well as in both `WHERE` and `HAVING` clauses with the same semantics as STRUCTs. The "tag" is always stored as the first struct entry, which ensures that the `UNION` types are compared and ordered by "tag" first.

Functions

See Nested Functions.

Expressions

Expressions

An expression is a combination of values, operators and functions. Expressions are highly composable, and range from very simple to arbitrarily complex. They can be found in many different parts of SQL statements. In this section, we provide the different types of operators and functions that can be used within expressions.

Case Statement

The CASE statement performs a switch based on a condition. The basic form is identical to the ternary condition used in many programming languages (`CASE WHEN cond THEN a ELSE b END` is equivalent to `cond ? a : b`). With a single condition this can be expressed with `IF(cond, a, b)`.

```
CREATE OR REPLACE TABLE INTEGERS AS SELECT UNNEST([1, 2, 3]) AS i;
SELECT i, CASE WHEN i>2 THEN 1 ELSE 0 END AS test FROM integers;
-- 1, 2, 3
-- 0, 0, 1

-- this is equivalent to:
SELECT i, IF(i > 2, 1, 0) AS test FROM integers;
-- 1, 2, 3
-- 0, 0, 1
```

The `WHEN cond THEN expr` part of the CASE statement can be chained, whenever any of the conditions returns true for a single tuple, the corresponding expression is evaluated and returned.

```
CREATE OR REPLACE TABLE INTEGERS AS SELECT UNNEST([1, 2, 3]) AS i;
SELECT i, CASE WHEN i=1 THEN 10 WHEN i=2 THEN 20 ELSE 0 END AS test FROM
 ↪  integers;
-- 1, 2, 3
-- 10, 20, 0
```

The `ELSE` part of the CASE statement is optional. If no else statement is provided and none of the conditions match, the CASE statement will return `NULL`.

```
CREATE OR REPLACE TABLE INTEGERS AS SELECT UNNEST([1, 2, 3]) AS i;
SELECT i, CASE WHEN i=1 THEN 10 END AS test FROM integers;
-- 1, 2, 3
-- 10, NULL, NULL
```

After the CASE but before the WHEN an individual expression can also be provided. When this is done, the CASE statement is essentially transformed into a switch statement.

```
CREATE OR REPLACE TABLE INTEGERS AS SELECT UNNEST([1, 2, 3]) AS i;
SELECT i, CASE i WHEN 1 THEN 10 WHEN 2 THEN 20 WHEN 3 THEN 30 END AS test
 ↪  FROM integers;
-- 1, 2, 3
-- 10, 20, 30

-- this is equivalent to:
SELECT i, CASE WHEN i=1 THEN 10 WHEN i=2 THEN 20 WHEN i=3 THEN 30 END AS
 ↪  test FROM integers;
```

Casting

Casting refers to the process of changing the type of a row from one type to another. The standard SQL syntax for this is `CAST(expr AS typename)`. DuckDB also supports the easier to type shorthand `expr::typename`, which is also present in PostgreSQL.

```
SELECT CAST(i AS VARCHAR) FROM generate_series(1, 3) tbl(i);
-- "1", "2", "3"
SELECT i::DOUBLE FROM generate_series(1, 3) tbl(i);
-- 1.0, 2.0, 3.0

SELECT CAST('hello' AS INTEGER);
-- Conversion Error: Could not convert string 'hello' to INT32
SELECT TRY_CAST('hello' AS INTEGER);
-- NULL
```

The exact behavior of the cast depends on the source and destination types. For example, when casting from VARCHAR to any other type, the string will be attempted to be converted.

Not all casts are possible. For example, it is not possible to convert an `INTEGER` to a `DATE`. Casts may also throw errors when the cast could not be successfully performed. For example, trying to cast the string `'hello'` to an `INTEGER` will result in an error being thrown.

`TRY_CAST` can be used when the preferred behavior is not to throw an error, but instead to return a `NULL` value. `TRY_CAST` will never throw an error, and will instead return `NULL` if a cast is not possible.

Implicit Casting

In many situations, the system will add casts by itself. This is called *implicit* casting. This happens for example when a function is called with an argument that does not match the type of the function, but can be casted to the desired type.

Consider the function `SIN(DOUBLE)`. This function takes as input argument a column of type `DOUBLE`, however, it can be called with an integer as well: `SIN(1)`. The integer is converted into a double before being passed to the `SIN` function.

Generally, implicit casts only cast upwards. That is to say, we can implicitly cast an `INTEGER` to a `BIGINT`, but not the other way around.

Collations

Collations provide rules for how text should be sorted or compared in the execution engine. Collations are useful for localization, as the rules for how text should be ordered are different for different languages or for different countries. These orderings are often incompatible with one another. For example, in English the letter "y" comes between "x" and "z". However, in Lithuanian the letter "y" comes between the "i" and "j". For that reason, different collations are supported. The user must choose which collation they want to use when performing sorting and comparison operations.

By default, the `BINARY` collation is used. That means that strings are ordered and compared based only on their binary contents. This makes sense for standard ASCII characters (i.e., the letters A-Z and numbers 0-9), but generally does not make much sense for special unicode characters. It is, however, by far the fastest method of performing ordering and comparisons. Hence it is recommended to stick with the `BINARY` collation unless required otherwise.

Using Collations

In the stand-alone installation of DuckDB three collations are included: `NOCASE`, `NOACCENT` and `NFC`. The `NOCASE` collation compares characters as equal regardless of their casing. The `NOACCENT` collation compares characters as equal regardless of their accents. The `NFC` collation performs NFC-normalized comparisons, see here for more information.

```
SELECT 'hello'='hElLO';
-- false
SELECT 'hello' COLLATE NOCASE='hElLO';
-- true

SELECT 'hello' = 'hëllo';
```

```
-- false
SELECT 'hello' COLLATE NOACCENT = 'hëllo';
-- true
```

Collations can be combined by chaining them using the dot operator. Note, however, that not all collations can be combined together. In general, the NOCASE collation can be combined with any other collator, but most other collations cannot be combined.

```
SELECT 'hello' COLLATE NOCASE='hElLÖ';
-- false
SELECT 'hello' COLLATE NOACCENT='hElLÖ';
-- false
SELECT 'hello' COLLATE NOCASE.NOACCENT='hElLÖ';
-- true
```

Default Collations

The collations we have seen so far have all been specified *per expression*. It is also possible to specify a default collator, either on the global database level or on a base table column. The PRAGMA `default_collation` can be used to specify the global default collator. This is the collator that will be used if no other one is specified.

```
PRAGMA default_collation=NOCASE;

SELECT 'hello'='HeLlo';
-- true
```

Collations can also be specified per-column when creating a table. When that column is then used in a comparison, the per-column collation is used to perform that comparison.

```
CREATE TABLE names(name VARCHAR COLLATE NOACCENT);
INSERT INTO names VALUES ('hännes');
SELECT name FROM names WHERE name='hannes';
-- hännes
```

Be careful here, however, as different collations cannot be combined. This can be problematic when you want to compare columns that have a different collation specified.

```
SELECT name FROM names WHERE name='hannes' COLLATE NOCASE;
-- ERROR: Cannot combine types with different collation!

CREATE TABLE other_names(name VARCHAR COLLATE NOCASE);
INSERT INTO other_names VALUES ('HÄNNES');
```

```
SELECT * FROM names, other_names WHERE names.name=other_names.name;
-- ERROR: Cannot combine types with different collation!

-- need to manually overwrite the collation!

SELECT * FROM names, other_names WHERE names.name COLLATE
 ↪  NOACCENT.NOCASE=other_names.name COLLATE NOACCENT.NOCASE;
-- hännes|HÄNNES
```

ICU Collations

The collations we have seen so far are not region-dependent, and do not follow any specific regional rules. If you wish to follow the rules of a specific region or language, you will need to use one of the ICU collations. For that, you need to load the ICU extension.

If you are using the C++ API, you may find the extension in the `extension/icu` folder of the DuckDB project. Using the C++ API, the extension can be loaded as follows:

```
DuckDB db;
db.LoadExtension<ICUExtension>();
```

Loading this extension will add a number of language and region specific collations to your database. These can be queried using `PRAGMA collations` command, or by querying the `pragma_collations` function.

```
PRAGMA collations;
SELECT * FROM pragma_collations();
-- [af, am, ar, as, az, be, bg, bn, bo, bs, bs, ca, ceb, chr, cs, cy, da,
 ↪  de, de_AT, dsb, dz, ee, el, en, en_US, en_US, eo, es, et, fa, fa_AF, fi,
 ↪  fil, fo, fr, fr_CA, ga, gl, gu, ha, haw, he, he_IL, hi, hr, hsb, hu, hy,
 ↪  id, id_ID, ig, is, it, ja, ka, kk, kl, km, kn, ko, kok, ku, ky, lb, lkt,
 ↪  ln, lo, lt, lv, mk, ml, mn, mr, ms, mt, my, nb, nb_NO, ne, nl, nn, om,
 ↪  or, pa, pa, pa_IN, pl, ps, pt, ro, ru, se, si, sk, sl, smn, sq, sr, sr,
 ↪  sr_BA, sr_ME, sr_RS, sr, sr_BA, sr_RS, sv, sw, ta, te, th, tk, to, tr,
 ↪  ug, uk, ur, uz, vi, wae, wo, xh, yi, yo, zh, zh, zh_CN, zh_SG, zh, zh_
 ↪  HK, zh_MO, zh_TW, zu]
```

These collations can then be used as the other collations would be used before. They can also be combined with the NOCASE collation. For example, to use the German collation rules you could use the following code snippet:

```
CREATE TABLE strings(s VARCHAR COLLATE DE);
```

```
INSERT INTO strings VALUES ('Gabel'), ('Göbel'), ('Goethe'), ('Goldmann'),
 ↪  ('Göthe'), ('Götz');
SELECT * FROM strings ORDER BY s;
-- "Gabel", "Göbel", "Goethe", "Goldmann", "Göthe", "Götz"
```

Comparisons

Comparison Operators

The table below shows the standard comparison operators. Whenever either of the input arguments is NULL, the output of the comparison is NULL.

Operator	Description	Example	Result
<	less than	2 < 3	true
>	greater than	2 > 3	false
<=	less than or equal to	2 <= 3	true
>=	greater than or equal to	4 >= NULL	NULL
=	equal	NULL = NULL	NULL
<> or !=	not equal	2 <> 2	false

The table below shows the standard distinction operators. These operators treat NULL values as equal.

Operator	Description	Example	Result
IS DISTINCT FROM	not equal, including NULL	2 IS DISTINCT FROM NULL	true
IS NOT DISTINCT FROM	equal, including NULL	NULL IS NOT DISTINCT FROM NULL	true

BETWEEN and IS (NOT) NULL

Besides the standard comparison operators there are also the BETWEEN and IS (NOT) NULL operators. These behave much like operators, but have special syntax mandated by the SQL standard.

They are shown in the table below.
Note that BETWEEN and NOT BETWEEN are only equivalent to the examples below in the cases where both a, x and y are of the same type, as BETWEEN will cast all of its inputs to the same type.

Predicate	Description
`a BETWEEN x AND y`	equivalent to `a >= x AND a <= y`
`a NOT BETWEEN x AND y`	equivalent to `a < x OR a > y`
`expression IS NULL`	`true` if expression is `NULL`, `false` otherwise
`expression ISNULL`	alias for `IS NULL` (non-standard)
`expression IS NOT NULL`	`false` if expression is `NULL`, `true` otherwise
`expression NOTNULL`	alias for `IS NOT NULL` (non-standard)

IN Operator

The `IN` operator checks containment of the left expression inside the set of expressions on the right hand side (RHS). The `IN` operator returns true if the expression is present in the RHS, false if the expression is not in the RHS and the RHS has no `NULL` values, or `NULL` if the expression is not in the RHS and the RHS has `NULL` values.

```
SELECT 'Math' IN ('CS', 'Math');
-- true
SELECT 'English' IN ('CS', 'Math');
-- false

SELECT 'Math' IN ('CS', 'Math', NULL);
-- true
SELECT 'English' IN ('CS', 'Math', NULL);
-- NULL
```

`NOT IN` can be used to check if an element is not present in the set. `X NOT IN Y` is equivalent to `NOT(X IN Y)`.

The `IN` operator can also be used with a subquery that returns a single column. See the subqueries page for more information.

Logical Operators

The following logical operators are available: AND, OR and NOT. SQL uses a three-valuad logic system with `true`, `false` and `NULL`. Note that logical operators involving `NULL` do not always evaluate to `NULL`. For example, `NULL AND false` will evaluate to `false`, and `NULL OR true` will evaluate to `true`. Below are the complete truth tables:

a	b	a AND b	a OR b
true	true	true	true
true	false	false	true
true	NULL	NULL	true
false	false	false	false
false	NULL	false	NULL
NULL	NULL	NULL	NULL

a	NOT a
true	false
false	true
NULL	NULL

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result.

Star Expression

Examples

```
-- select all columns present in the FROM clause
SELECT * FROM table_name;
-- select all columns from the table called "table_name"
SELECT table_name.* FROM table_name JOIN other_table_name USING (id);
-- select all columns except the city column from the addresses table
SELECT * EXCLUDE (city) FROM addresses;
```

```
-- select all columns from the addresses table, but replace city with
 ↪  LOWER(city)
SELECT * REPLACE (LOWER(city) AS city) FROM addresses;
-- select all columns matching the given expression
SELECT COLUMNS(c -> c LIKE '%num%') FROM addresses;
-- select all columns matching the given regex from the table
SELECT COLUMNS('number\d+') FROM addresses;
```

Syntax

Star Expression

The * expression can be used in a SELECT statement to select all columns that are projected in the FROM clause.

```
SELECT * FROM tbl;
```

The * expression can be modified using the EXCLUDE and REPLACE.

EXCLUDE Clause EXCLUDE allows us to exclude specific columns from the * expression.

```
SELECT * EXCLUDE (col) FROM tbl;
```

Replace Clause REPLACE allows us to replace specific columns with different expressions.

```
SELECT * REPLACE (col / 1000 AS col) FROM tbl;
```

COLUMNS

The COLUMNS expression can be used to execute the same expression on multiple columns. Like the * expression, it can only be used in the SELECT clause.

```
CREATE TABLE numbers(id int, number int);
INSERT INTO numbers VALUES (1, 10), (2, 20), (3, NULL);
SELECT MIN(COLUMNS(*)), COUNT(COLUMNS(*)) FROM numbers;
```

min(numbers.id)	min(numbers.number)	count(numbers.id)	count(numbers.number)
1	10	3	2

The * expression in the COLUMNS statement can also contain EXCLUDE or REPLACE, similar to regular star expressions.

```
SELECT MIN(COLUMNS(* REPLACE (number + id AS number))), COUNT(COLUMNS(*
↪  EXCLUDE (number))) FROM numbers;
```

min(numbers.id)	min(number := (number + id))	count(numbers.id)
1	11	3

COLUMNS expressions can also be combined, as long as the COLUMNS contains the same (star) expression:

```
SELECT COLUMNS(*) + COLUMNS(*) FROM numbers;
```

(numbers.id + numbers.id)	(numbers.number + numbers.number)
2	20
4	40
6	NULL

COLUMNS Regular Expression

COLUMNS supports passing a regex in as a string constant:

```
SELECT COLUMNS('(id|numbers?)') FROM numbers;
```

id	number
1	10
2	20
3	NULL

COLUMNS Lambda Function

COLUMNS also supports passing in a lambda function. The lambda function will be evaluated for all columns present in the FROM clause, and only columns that match the lambda function will be returned. This allows the execution of arbitrary expressions in order to select columns.

```
SELECT COLUMNS(c -> c LIKE '%num%') FROM numbers;
```

number
10
20
NULL

Struct.*

The * expression can also be used to retrieve all keys from a struct as separate columns. This is particularly useful when a prior operation creates a struct of unknown shape, or if a query must handle any potential struct keys. See the struct and nested function pages for more details on working with structs.

```
-- All keys within a struct can be returned as separate columns using *
SELECT a.* FROM (SELECT {'x':1, 'y':2, 'z':3} AS a);
```

x	y	z
1	2	3

Subqueries

Scalar Subquery

Scalar subqueries are subqueries that return a single value. They can be used anywhere where a regular expression can be used. If a scalar subquery returns more than a single value, the first value returned will be used.

Consider the following table:

Grades

grade	course
7	Math
9	Math
8	CS

```
CREATE TABLE grades(grade INTEGER, course VARCHAR);
INSERT INTO grades VALUES (7, 'Math'), (9, 'Math'), (8, 'CS');
```

We can run the following query to obtain the minimum grade:

```
SELECT MIN(grade) FROM grades;
-- {7}
```

By using a scalar subquery in the WHERE clause, we can figure out for which course this grade was obtained:

```
SELECT course FROM grades WHERE grade = (SELECT MIN(grade) FROM grades);
-- {Math}
```

Exists

The EXISTS operator tests for the existence of any row inside the subquery. It returns either true when the subquery returns one or more records, and false otherwise. The EXISTS operator is generally the most useful as a *correlated* subquery to express semijoin operations. However, it can be used as an uncorrelated subquery as well.

For example, we can use it to figure out if there are any grades present for a given course:

```
SELECT EXISTS (SELECT * FROM grades WHERE course='Math');
-- true

SELECT EXISTS (SELECT * FROM grades WHERE course='History');
-- false
```

Not exists The NOT EXISTS operator tests for the absence of any row inside the subquery. It returns either true when the subquery returns an empty result, and false otherwise. The NOT EXISTS operator is generally the most useful as a *correlated* subquery to express antijoin operations. For example, to find Person nodes without an interest:

```
CREATE TABLE Person(id BIGINT, name VARCHAR);
CREATE TABLE interest(PersonId BIGINT, topic VARCHAR);

INSERT INTO Person VALUES (1, 'Jane'), (2, 'Joe');
INSERT INTO interest VALUES (2, 'Music');

SELECT *
FROM Person
WHERE NOT EXISTS (SELECT * FROM interest WHERE interest.PersonId =
 ↪  Person.id);
```

id int64	name varchar
1	Jane

> **Note.** DuckDB automatically detects when a `NOT EXISTS` query expresses an antijoin operation. There is no need to manually rewrite such queries to use `LEFT OUTER JOIN ... WHERE ... IS NULL`.

In Operator

The `IN` operator checks containment of the left expression inside the result defined by the subquery or the set of expressions on the right hand side (RHS). The `IN` operator returns true if the expression is present in the RHS, false if the expression is not in the RHS and the RHS has no `NULL` values, or `NULL` if the expression is not in the RHS and the RHS has `NULL` values.

We can use the `IN` operator in a similar manner as we used the `EXISTS` operator:

```
SELECT 'Math' IN (SELECT course FROM grades);
-- true
```

Correlated Subqueries

All the subqueries presented here so far have been **uncorrelated** subqueries, where the subqueries themselves are entirely self-contained and can be run without the parent query. There exists a second type of subqueries called **correlated** subqueries. For correlated subqueries, the subquery uses values from the parent subquery.

Conceptually, the subqueries are run once for every single row in the parent query. Perhaps a simple way of envisioning this is that the correlated subquery is a **function** that is applied to every row in the source data set.

For example, suppose that we want to find the minimum grade for every course. We could do that as follows:

```
SELECT *
FROM grades grades_parent
WHERE grade=
    (SELECT MIN(grade)
     FROM grades
     WHERE grades.course=grades_parent.course);
-- {7, Math}, {8, CS}
```

The subquery uses a column from the parent query (`grades_parent.course`). Conceptually, we can see the subquery as a function where the correlated column is a parameter to that function:

```
SELECT MIN(grade) FROM grades WHERE course=?;
```

Now when we execute this function for each of the rows, we can see that for `Math` this will return 7, and for CS it will return 8. We then compare it against the grade for that actual row. As a result, the row `(Math, 9)` will be filtered out, as `9 <> 7`.

Returning Each Row of the Subquery as a Struct

Using the name of a subquery in the SELECT clause (without referring to a specific column) turns each row of the subquery into a struct whose fields correspond to the columns of the subquery. For example:

```
SELECT t FROM (SELECT unnest(generate_series(41, 43)) AS x, 'hello' AS y) t;
```

```
┌──────────────────────────────┐
│              t               │
│ struct(x bigint, y varchar)  │
├──────────────────────────────┤
│ {'x': 41, 'y': hello}        │
│ {'x': 42, 'y': hello}        │
│ {'x': 43, 'y': hello}        │
└──────────────────────────────┘
```

Functions

Functions

Function Syntax

Query Functions `duckdb_functions` table function shows the list of functions currently built into the system.

```
SELECT DISTINCT ON(function_name) function_name, function_type, return_type,
 ↪  parameters, parameter_types, description
FROM duckdb_functions()
WHERE function_type = 'scalar' AND function_name LIKE 'b%'
ORDER BY function_name;
```

```
┌───────────────┬───────────────┬─────────────┬──────────────────┬──────────────
│ function_name │ function_type │ return_type │      parameters      │
↪  parameter_types    │              description               │
│    varchar    │    varchar    │   varchar   │      varchar[]       │
↪  varchar[]       │                varchar                 │
├───────────────┼───────────────┼─────────────┼──────────────────┼──────────────
│ bar           │ scalar        │ VARCHAR     │ [x, min, max, width] │
↪  [DOUBLE, DOUBLE, D…  │ Draws a band whose width is proportion…  │
│ base64        │ scalar        │ VARCHAR     │ [blob]               │ [BLOB]
↪  │ Converts a blob to a base64 encoded st…  │
│ bin           │ scalar        │ VARCHAR     │ [value]              │
↪  [VARCHAR]           │ Converts the value to binary represent…  │
│ bit_count     │ scalar        │ TINYINT     │ [x]                  │
↪  [TINYINT]           │ Returns the number of bits that are set  │
│ bit_length    │ scalar        │ BIGINT      │ [col0]               │
↪  [VARCHAR]           │                                          │
│ bit_position  │ scalar        │ INTEGER     │ [substring, bitstr…  │ [BIT,
↪  BIT]          │ Returns first starting index of the sp…  │
│ bitstring     │ scalar        │ BIT         │ [bitstring, length]  │
↪  [VARCHAR, INTEGER]   │ Pads the bitstring until the specified…  │
└───────────────┴───────────────┴─────────────┴──────────────────┴──────────────
```

Currently the description and parameter names of functions are still missing.

Bitstring Functions

This section describes functions and operators for examining and manipulating bit values. Bitstrings must be of equal length when performing the bitwise operands AND, OR and XOR. When bit shifting,

the original length of the string is preserved.

Bitstring Operators

The table below shows the available mathematical operators for `BIT` type.

Operator	Description	Example	Result
`&`	bitwise AND	`'10101'::BIT & '10001'::BIT`	`10001`
`|`	bitwise OR	`'1011'::BIT | '0001'::BIT`	`1011`
`xor`	bitwise XOR	`xor('101'::BIT, '001'::BIT)`	`100`
`~`	bitwise NOT	`~('101'::BIT)`	`010`
`<<`	bitwise shift left	`'1001011'::BIT << 3`	`1011000`
`>>`	bitwise shift right	`'1001011'::BIT >> 3`	`0001001`

Bitstring Functions

The table below shows the available scalar functions for `BIT` type.

Function	Description	Example	Result
`bit_count(`*`bitstring`*`)`	Returns the number of set bits in the bitstring.	`bit_count('1101011'::BIT)`	`5`
`bit_length(`*`bitstring`*`)`	Returns the number of bits in the bitstring.	`bit_length('1101011'::BIT)`	`7`
`bit_position(`*`substring`*`,` *`bitstring`*`)`	Returns first starting index of the specified substring within bits, or zero if it's not present. The first (leftmost) bit is indexed 1	`bit_position('010'::BIT, '1110101'::BIT)`	`4`
`bitstring(`*`bitstring`*`,` *`length`*`)`	Returns a bitstring of determined length.	`bitstring('1010'::BIT, 7)`	`0001010`

Function	Description	Example	Result
`get_bit(`*`bitstring`*`,`*`index`*`)`	Extracts the nth bit from bitstring; the first (leftmost) bit is indexed 0.	`get_bit('0110010'::BIT, 2)`	`1`
`length(`*`bitstring`*`)`	Alias for `bit_length`.	`length('1101011'::BIT)`	`7`
`octet_length(`*`bitstring`*`)`	Returns the number of bytes in the bitstring.	`octet_length('1101011'::BIT)`	`1`
`set_bit(`*`bitstring`*`,`*`index`*`,`*`new_value`*`)`	Sets the nth bit in bitstring to newvalue; the first (leftmost) bit is indexed 0. Returns a new bitstring.	`set_bit('0110010'::BIT, 2, 0)`	`0100010`

Bitstring Aggregate Functions

These aggregate functions are available for `BIT` type.

Function	Description	Example
`bit_and(arg)`	Returns the bitwise AND operation performed on all bitstrings in a given expression.	`bit_and(A)`
`bit_or(arg)`	Returns the bitwise OR operation performed on all bitstrings in a given expression.	`bit_or(A)`
`bit_xor(arg)`	Returns the bitwise XOR operation performed on all bitstrings in a given expression.	`bit_xor(A)`
`bitstring_agg(arg)`	Returns a bitstring with bits set for each distinct value.	`bitstring_agg(A)`
`bitstring_agg(arg, min, max)`	Returns a bitstring with bits set for each distinct value.	`bitstring_agg(A, 1, 42)`

Bitstring Aggregation The `BITSTRING_AGG` function takes any integer type as input and returns a bitstring with bits set for each distinct value. The left-most bit represents the smallest value in the column and the right-most bit the maximum value. If possible, the min and max are retrieved from the column statistics. Otherwise, it is also possible to provide the min and max values.

The combination of `BIT_COUNT` and `BITSTRING_AGG` could be used as an alternative to `COUNT DISTINCT`, with possible performance improvements in cases of low cardinality and dense values.

Blob Functions

This section describes functions and operators for examining and manipulating blob values.

Function	Description	Example	Result
blob || *blob*	Blob concatenation	`'\xAA'::BLOB || '\xBB'::BLOB`	\xAA\xBB
`decode(`*blob*`)`	Convert blob to varchar. Fails if blob is not valid utf-8.	`decode('\xC3\xBC'::BLOB)`	ü
`encode(`*string*`)`	Convert varchar to blob. Converts utf-8 characters into literal encoding.	`encode('my_string_with_ü')`	my_string_with_\xC3\xBC
`octet_length(`*blob*`)`	Number of bytes in blob	`octet_length('\xAA\xBB'::BLOB)`	2

Date Format

The `strftime` and `strptime` functions can be used to convert between dates/timestamps and strings. This is often required when parsing CSV files, displaying output to the user or transferring information between programs. Because there are many possible date representations, these functions accept a format string that describes how the date or timestamp should be structured.

strftime Examples

`strftime(timestamp, format)` converts timestamps or dates to strings according to the specified pattern.

```
SELECT strftime(DATE '1992-03-02', '%d/%m/%Y');
-- 02/03/1992
SELECT strftime(TIMESTAMP '1992-03-02 20:32:45', '%A, %-d %B %Y - %I:%M:%S
 ↪  %p');
-- Monday, 2 March 1992 - 08:32:45 PM
```

strptime Examples

`strptime(string, format)` converts strings to timestamps according to the specified pattern.

```
SELECT strptime('02/03/1992', '%d/%m/%Y');
-- 1992-03-02 00:00:00
SELECT strptime('Monday, 2 March 1992 - 08:32:45 PM', '%A, %-d %B %Y -
 ↪  %I:%M:%S %p');
-- 1992-03-02 20:32:45
```

CSV Parsing

The date formats can also be specified during CSV parsing, either in the COPY statement or in the `read_csv` function. This can be done by either specifying a `DATEFORMAT` or a `TIMESTAMPFORMAT` (or both). `DATEFORMAT` will be used for converting dates, and `TIMESTAMPFORMAT` will be used for converting timestamps. Below are some examples for how to use this:

```
-- in COPY statement
COPY dates FROM 'test.csv' (DATEFORMAT '%d/%m/%Y', TIMESTAMPFORMAT '%A, %-d
 ↪  %B %Y - %I:%M:%S %p');

-- in read_csv function
SELECT * FROM read_csv('test.csv', dateformat='%m/%d/%Y');
```

Format Specifiers

Below is a full list of all available format specifiers.

Specifier	Description	Example
%a	Abbreviated weekday name.	Sun, Mon, ...
%A	Full weekday name.	Sunday, Monday, ...

Specifier	Description	Example
%w	Weekday as a decimal number.	0, 1, ..., 6
%d	Day of the month as a zero-padded decimal.	01, 02, ..., 31
%-d	Day of the month as a decimal number.	1, 2, ..., 30
%b	Abbreviated month name.	Jan, Feb, ..., Dec
%B	Full month name.	January, February, ...
%m	Month as a zero-padded decimal number.	01, 02, ..., 12
%-m	Month as a decimal number.	1, 2, ..., 12
%y	Year without century as a zero-padded decimal number.	00, 01, ..., 99
%-y	Year without century as a decimal number.	0, 1, ..., 99
%Y	Year with century as a decimal number.	2013, 2019 etc.
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23
%-H	Hour (24-hour clock) as a decimal number.	0, 1, ..., 23
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12
%-I	Hour (12-hour clock) as a decimal number.	1, 2, ... 12
%p	Locale's AM or PM.	AM, PM
%M	Minute as a zero-padded decimal number.	00, 01, ..., 59
%-M	Minute as a decimal number.	0, 1, ..., 59
%S	Second as a zero-padded decimal number.	00, 01, ..., 59
%-S	Second as a decimal number.	0, 1, ..., 59
%g	Millisecond as a decimal number, zero-padded on the left.	000 - 999
%f	Microsecond as a decimal number, zero-padded on the left.	000000 - 999999
%n	Nanosecond as a decimal number, zero-padded on the left.	000000000 - 999999999

Specifier	Description	Example
%z	Time offset from UTC in the form ±HH:MM, ±HHMM, or ±HH.	-0700
%Z	Time zone name.	Europe/Amsterdam
%j	Day of the year as a zero-padded decimal number.	001, 002, ..., 366
%-j	Day of the year as a decimal number.	1, 2, ..., 366
%U	Week number of the year (Sunday as the first day of the week).	00, 01, ..., 53
%W	Week number of the year (Monday as the first day of the week).	00, 01, ..., 53
%c	ISO date and time representation	1992-03-02 10:30:20
%x	ISO date representation	1992-03-02
%X	ISO time representation	10:30:20
%%	A literal '%' character.	%

Date Functions

This section describes functions and operators for examining and manipulating date values.

Date Operators

The table below shows the available mathematical operators for DATE types.

Operator	Description	Example	Result
+	addition of days (integers)	`DATE '1992-03-22' + 5`	1992-03-27
+	addition of an INTERVAL	`DATE '1992-03-22' + INTERVAL 5 DAY`	1992-03-27

Operator	Description	Example	Result
+	addition of a variable INTERVAL	`SELECT DATE '1992-03-22' + INTERVAL 1 DAY * d.days FROM (VALUES (5), (11)) AS d(days)`	1992-03-27 1992-04-02
-	subtraction of DATEs	`DATE '1992-03-27' - DATE '1992-03-22'`	5
-	subtraction of an INTERVAL	`DATE '1992-03-27' - INTERVAL 5 DAY`	1992-03-22
-	subtraction of a variable INTERVAL	`SELECT DATE '1992-03-27' - INTERVAL 1 DAY * d.days FROM (VALUES (5), (11)) AS d(days)`	1992-03-22 1992-03-16

Adding to or subtracting from infinite values produces the same infinite value.

Date Functions

The table below shows the available functions for DATE types. Dates can also be manipulated with the timestamp functions through type promotion.

Function	Description	Example	Result
`current_date`	Current date (at start of current transaction)	`current_date`	`2022-10-08`
`date_diff(`*`part`*`, `*`startdate`*`, `*`enddate`*`)`	The number of partition boundaries between the dates	`date_diff('month', DATE '1992-09-15', DATE '1992-11-14')`	2
`datediff(`*`part`*`, `*`startdate`*`, `*`enddate`*`)`	Alias of date_diff. The number of partition boundaries between the dates	`datediff('month', DATE '1992-09-15', DATE '1992-11-14')`	2

Function	Description	Example	Result
`date_part(`*`part`*`,` *`date`*`)`	Get the subfield (equivalent to `extract`)	`date_part('year', DATE '1992-09-20')`	`1992`
`datepart(`*`part`*`,` *`date`*`)`	Alias of date_part. Get the subfield (equivalent to `extract`)	`datepart('year', DATE '1992-09-20')`	`1992`
`date_sub(`*`part`*`,` *`startdate`*`,` *`enddate`*`)`	The number of complete partitions between the dates	`date_sub('month', DATE '1992-09-15', DATE '1992-11-14')`	`1`
`datesub(`*`part`*`,` *`startdate`*`,` *`enddate`*`)`	Alias of date_sub. The number of complete partitions between the dates	`datesub('month', DATE '1992-09-15', DATE '1992-11-14')`	`1`
`date_trunc(` *`part`*`,` *`date`*`)`	Truncate to specified precision	`date_trunc('month', DATE '1992-03-07')`	`1992-03-01`
`datetrunc(`*`part`*`,` *`date`*`)`	Alias of date_trunc. Truncate to specified precision	`datetrunc('month', DATE '1992-03-07')`	`1992-03-01`
`dayname(` *`date`*`)`	The (English) name of the weekday	`dayname(DATE '1992-09-20')`	`Sunday`
`isfinite(` *`date`*`)`	Returns true if the date is finite, false otherwise	`isfinite(DATE '1992-03-07')`	`true`
`isinf(` *`date`*`)`	Returns true if the date is infinite, false otherwise	`isinf(DATE '-infinity')`	`true`
`extract(` *`part`* `fromdate`*`)`	Get subfield from a date	`extract('year' FROM DATE '1992-09-20')`	`1992`
`greatest(`*`date`*`,` *`date`*`)`	The later of two dates	`greatest(DATE '1992-09-20', DATE '1992-03-07')`	`1992-09-20`

Function	Description	Example	Result
`last_day(`*`date`*`)`	The last day of the corresponding month in the date	`last_day(DATE '1992-09-20')`	`1992-09-30`
`least(`*`date`*`, `*`date`*`)`	The earlier of two dates	`least(DATE '1992-09-20', DATE '1992-03-07')`	`1992-03-07`
`make_date(`*`bigint`*`, `*`bigint`*`, `*`bigint`*`)`	The date for the given parts	`make_date(1992, 9, 20)`	`1992-09-20`
`monthname(`*`date`*`)`	The (English) name of the month	`monthname(DATE '1992-09-20')`	`September`
`strftime(date, format)`	Converts a date to a string according to the format string	`strftime(date '1992-01-01', '%a, %-d %B %Y')`	`Wed, 1 January 1992`
`time_bucket(`*`bucket_width`*`, `*`date`*`[,`*`origin`*`])`	Truncate `date` by the specified interval `bucket_width`. Buckets are aligned relative to `origin` date. `origin` defaults to 2000-01-03 for buckets that don't include a month or year interval, and to 2000-01-01 for month and year buckets.	`time_bucket(INTERVAL '2 weeks', DATE '1992-04-20', DATE '1992-04-01')`	`1992-04-15`
`time_bucket(`*`bucket_width`*`, `*`date`*`[,`*`offset`*`])`	Truncate `date` by the specified interval `bucket_width`. Buckets are offset by `offset` interval.	`time_bucket(INTERVAL '2 months', DATE '1992-04-20', INTERVAL '1 month')`	`1992-04-01`
`today()`	Current date (start of current transaction)	`today()`	`2022-10-08`

There are also dedicated extraction functions to get the subfields. A few examples include extracting the day from a date, or the day of the week from a date.

Functions applied to infinite dates will either return the same infinite dates (e.g, `greatest`) or `NULL` (e.g., `date_part`) depending on what "makes sense". In general, if the function needs to examine the parts of the infinite date, the result will be `NULL`.

Date Parts

The `date_part` and `date_diff` and `date_trunc` functions can be used to manipulate the fields of temporal types. The fields are specified as strings that contain the part name of the field.

Part Specifiers

Below is a full list of all available date part specifiers. The examples are the corresponding parts of the timestamp `2021-08-03 11:59:44.123456`.

Usable as Date Part Specifiers and in Intervals

Specifier	Description	Synonyms	Example
`'century'`	Gregorian century	`'cent'`, `'centuries'`, `'c'`	21
`'day'`	Gregorian day	`'days'`, `'d'`, `'dayofmonth'`	3
`'decade'`	Gregorian decade	`'dec'`, `'decades'`, `'decs'`	202
`'hour'`	Hours	`'hr'`, `'hours'`, `'hrs'`, `'h'`	11
`'microseconds'`	Sub-minute microseconds	`'microsecond'`, `'us'`, `'usec'`, `'usecs'`, `'usecond'`, `'useconds'`	44123456
`'millennium'`	Gregorian millennium	`'mil'`, `'millenniums'`, `'millenia'`, `'mils'`, `'millenium'`	3

Specifier	Description	Synonyms	Example
`'milliseconds'`	Sub-minute milliseconds	`'millisecond'`, `'ms'`, `'msec'`, `'msecs'`, `'msecond'`, `'mseconds'`	44123
`'minute'`	Minutes	`'min'`, `'minutes'`, `'mins'`, `'m'`	59
`'month'`	Gregorian month	`'mon'`, `'months'`, `'mons'`	8
`'quarter'`	Quarter of the year (1-4)	`'quarters'`	3
`'second'`	Seconds	`'sec'`, `'seconds'`, `'secs'`, `'s'`	44
`'year'`	Gregorian year	`'yr'`, `'y'`, `'years'`, `'yrs'`	2021

Usable in Date Part Specifiers Only

Specifier	Description	Synonyms	Example
`'dayofweek'`	Day of the week (Sunday = 0, Saturday = 6)	`'weekday'`, `'dow'`	2
`'dayofyear'`	Day of the year (1-365/366)	`'doy'`	215
`'epoch'`	Seconds since 1970-01-01		1627991984
`'era'`	Gregorian era (CE/AD, BCE/BC)		1
`'isodow'`	ISO day of the week (Monday = 1, Sunday = 7)		2
`'isoyear'`	ISO Year number (Starts on Monday of week containing Jan 4th)		2021

Specifier	Description	Synonyms	Example
`'timezone'`	Time zone offset in seconds		0
`'timezone_hour'`	Time zone offset hour portion		0
`'timezone_minute'`	Time zone offset minute portion		0
`'week'`	Week number	`'weeks'`, `'w'`	31
`'yearweek'`	ISO year and week number in YYYYWW format		202131

Note that the time zone parts are all zero unless a time zone plugin such as ICU has been installed to support `TIMESTAMP WITH TIME ZONE`.

Part Functions There are dedicated extraction functions to get certain subfields:

Function	Description	Example	Result
`century(`*`date`*`)`	Century	`century(date '1992-02-15')`	`20`
`day(`*`date`*`)`	Day	`day(date '1992-02-15')`	`15`
`dayofmonth(`*`date`*`)`	Day (synonym)	`dayofmonth(date '1992-02-15')`	`15`
`dayofweek(`*`date`*`)`	Numeric weekday (Sunday = 0, Saturday = 6)	`dayofweek(date '1992-02-15')`	`6`
`dayofyear(`*`date`*`)`	Day of the year (starts from 1, i.e., January 1 = 1)	`dayofyear(date '1992-02-15')`	`46`
`decade(`*`date`*`)`	Decade (year / 10)	`decade(date '1992-02-15')`	`199`

Function	Description	Example	Result
`epoch(`*`date`*`)`	Seconds since 1970-01-01	`epoch(date '1992-02-15')`	`698112000`
`era(`*`date`*`)`	Calendar era	`era(date '0044-03-15 (BC)')`	`0`
`hour(`*`date`*`)`	Hours	`hour(timestamp '2021-08-03 11:59:44.123456')`	`11`
`isodow(`*`date`*`)`	Numeric ISO weekday (Monday = 1, Sunday = 7)	`isodow(date '1992-02-15')`	`6`
`isoyear(`*`date`*`)`	ISO Year number (Starts on Monday of week containing Jan 4th)	`isoyear(date '2022-01-01')`	`2021`
`microsecond(`*`date`*`)`	Sub-minute microseconds	`microsecond(timestamp '2021-08-03 11:59:44.123456')`	`44123456`
`millennium(`*`date`*`)`	Millennium	`millennium(date '1992-02-15')`	`2`
`millisecond(`*`date`*`)`	Sub-minute milliseconds	`millisecond(timestamp '2021-08-03 11:59:44.123456')`	`44123`
`minute(`*`date`*`)`	Minutes	`minute(timestamp '2021-08-03 11:59:44.123456')`	`59`
`month(`*`date`*`)`	Month	`month(date '1992-02-15')`	`2`
`quarter(`*`date`*`)`	Quarter	`quarter(date '1992-02-15')`	`1`
`second(`*`date`*`)`	Seconds	`second(timestamp '2021-08-03 11:59:44.123456')`	`44`

Function	Description	Example	Result
`timezone(`*`date`*`)`	Time Zone offset in minutes	`timezone(date '1992-02-15')`	`0`
`timezone_hour(`*`date`*`)`	Time zone offset hour portion	`timezone_hour(date '1992-02-15')`	`0`
`timezone_minute(`*`date`*`)`	Time zone offset minutes portion	`timezone_minute(date '1992-02-15')`	`0`
`week(`*`date`*`)`	ISO Week	`week(date '1992-02-15')`	`7`
`weekday(`*`date`*`)`	Numeric weekday synonym (Sunday = 0, Saturday = 6)	`weekday(date '1992-02-15')`	`6`
`weekofyear(`*`date`*`)`	ISO Week (synonym)	`weekofyear(date '1992-02-15')`	`7`
`year(`*`date`*`)`	Year	`year(date '1992-02-15')`	`1992`
`yearweek(`*`date`*`)`	`BIGINT` of combined ISO Year number and 2-digit version of ISO Week number	`yearweek(date '1992-02-15')`	`199207`

Enum Functions

This section describes functions and operators for examining and manipulating `ENUM` values. The examples assume an enum type created as:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy', 'anxious');
```

These functions can take `NULL` or a specific value of the type as argument(s). With the exception of `enum_range_boundary`, the result depends only on the type of the argument and not on its value.

Function	Description	Example	Result
`enum_code(`*`enum_value`*`)`	Returns the numeric value backing the given enum value	`enum_code('happy'::mood)`	2
`enum_first(`*`enum`*`)`	Returns the first value of the input enum type.	`enum_first(null::mood)`	sad
`enum_last(`*`enum`*`)`	Returns the last value of the input enum type.	`enum_last(null::mood)`	anxious
`enum_range(`*`enum`*`)`	Returns all values of the input enum type as an array.	`enum_range(null::mood)`	[sad, ok, happy, anxious]
`enum_range_boundary(`*`enum`*`,`*`enum`*`)`	Returns the range between the two given enum values as an array. The values must be of the same enum type. When the first parameter is `NULL`, the result starts with the first value of the enum type. When the second parameter is `NULL`, the result ends with the last value of the enum type.	`enum_range_boundary(NULL, 'happy'::mood)`	[sad, ok, happy]

Interval Functions

This section describes functions and operators for examining and manipulating `INTERVAL` values.

Interval Operators

The table below shows the available mathematical operators for `INTERVAL` types.

Operator	Description	Example	Result
+	addition of an `INTERVAL`	`INTERVAL 1 HOUR + INTERVAL 5 HOUR`	INTERVAL 6 HOUR
+	addition to a `DATE`	`DATE '1992-03-22' + INTERVAL 5 DAY`	1992-03-27
+	addition to a `TIMESTAMP`	`TIMESTAMP '1992-03-22 01:02:03' + INTERVAL 5 DAY`	1992-03-27 01:02:03
+	addition to a `TIME`	`TIME '01:02:03' + INTERVAL 5 HOUR`	06:02:03
-	subtraction of an `INTERVAL`	`INTERVAL 5 HOUR - INTERVAL 1 HOUR`	INTERVAL 4 HOUR
-	subtraction from a `DATE`	`DATE '1992-03-27' - INTERVAL 5 DAY`	1992-03-22
-	subtraction from a `TIMESTAMP`	`TIMESTAMP '1992-03-27 01:02:03' - INTERVAL 5 DAY`	1992-03-22 01:02:03
-	subtraction from a `TIME`	`TIME '06:02:03' - INTERVAL 5 HOUR`	01:02:03

Interval Functions

The table below shows the available scalar functions for `INTERVAL` types.

Function	Description	Example	Result
`date_part(`*`part`*`,`*`interval`*`)`	Get subfield (equivalent to *extract*)	`date_part('year', INTERVAL '14 months')`	1

Function	Description	Example	Result
`datepart(`*`part`*`,` *`interval`*`)`	Alias of date_part. Get subfield (equivalent to *extract*)	`datepart('year', INTERVAL '14 months')`	1
`extract(`*`part`* `from` *`interval`*`)`	Get subfield from a date	`extract('month' FROM INTERVAL '14 months')`	2
`to_days(` *`integer`*`)`	Construct a day interval	`to_days(5)`	INTERVAL 5 DAY
`to_hours(` *`integer`*`)`	Construct a hour interval	`to_hours(5)`	INTERVAL 5 HOUR
`to_microseconds(` *`integer`*`)`	Construct a microsecond interval	`to_microseconds(5)`	INTERVAL 5 MICROSECOND
`to_milliseconds(` *`integer`*`)`	Construct a millisecond interval	`to_milliseconds(5)`	INTERVAL 5 MILLISECOND
`to_minutes(` *`integer`*`)`	Construct a minute interval	`to_minutes(5)`	INTERVAL 5 MINUTE
`to_months(` *`integer`*`)`	Construct a month interval	`to_months(5)`	INTERVAL 5 MONTH
`to_seconds(` *`integer`*`)`	Construct a second interval	`to_seconds(5)`	INTERVAL 5 SECOND
`to_years(` *`integer`*`)`	Construct a year interval	`to_years(5)`	INTERVAL 5 YEAR

Only the documented date parts are defined for intervals.

Nested Functions

This section describes functions and operators for examining and manipulating nested values. There are three nested data types: lists, structs, and maps.

List Functions

In the descriptions, `l` is the three element list `[4, 5, 6]`.

Function	Aliases	Description	Example	Result
list`[`*index*`]`		Bracket notation serves as an alias for `list_extract`.	`l[3]`	`6`
list`[`*begin*`:`*end*`]`		Bracket notation with colon is an alias for `list_slice`.	`l[2:3]`	`[5, 6]`
list`[`*begin*`:`*end*`:`*step*`]`		`list_slice` in bracket notation with an added `step` feature.	`l[:-:2]`	`[4, 6]`
`array_pop_back(`*list*`)`		Returns the list without the last element.	`array_pop_back(l)`	`[4, 5]`
`array_pop_front(`*list*`)`		Returns the list without the first element.	`array_pop_front(l)`	`[5, 6]`
`flatten(`*list_of_lists*`)`		Concatenate a list of lists into a single list. This only flattens one level of the list (see examples).	`flatten([[1, 2], [3, 4]])`	`[1, 2, 3, 4]`
`len(`*list*`)`	`array_length`	Return the length of the list.	`len([1, 2, 3])`	`3`

Function	Aliases	Description	Example	Result
`list_aggregate(list, name)`	`list_aggr`, `aggregate`, `array_aggregate`, `array_aggr`	Executes the aggregate function name on the elements of `list`. See the List Aggregates section for more details.	`list_aggregate([1, 2, NULL], 'min')`	`1`
`list_any_value(list)`		Returns the first non-null value in the list	`list_any_value([NULL, -3])`	`-3`
`list_append(list, element)`	`array_append`, `array_push_back`	Appends `element` to `list`.	`list_append([2, 3], 4)`	`[2, 3, 4]`
`list_concat(list1, list2)`	`list_cat`, `array_concat`, `array_cat`	Concatenates two lists.	`list_concat([2, 3], [4, 5, 6])`	`[2, 3, 4, 5, 6]`
`list_contains(list, element)`	`list_has`, `array_contains`, `array_has`	Returns true if the list contains the element.	`list_contains([1, 2, NULL], 1)`	`true`
`list_cosine_similarity(list1, list2)`		Compute the cosine similarity between two lists	`list_cosine_similarity([1, 2, 3], [1, 2, 5])`	`0.9759000729485332`
`list_distance(list1, list2)`			`list_distance([1, 2, 3], [1, 2, 5])`	`2.0`

Function	Aliases	Description	Example	Result
`list_distinct(list)`	`array_distinct`	Removes all duplicates and NULLs from a list. Does not preserve the original order.	`list_distinct([1, 1, NULL, -3, 1, 5])`	`[1, 5, -3]`
`list_dot_product(list1, list2)`	`list_inner_product`		`list_dot_product([1, 2, 3], [1, 2, 5])`	`20.0`
`list_extract(list, index)`	`list_element`, `array_extract`	Extract the `index`th (1-based) value from the list.	`list_extract(l, 3)`	`6`
`list_filter(list, lambda)`	`array_filter`, `filter`	Constructs a list from those elements of the input list for which the lambda function returns true. See the Lambda Functions section for more details.	`list_filter(l, x -> x > 4)`	`[5, 6]`
`list_has_all(list, sub-list)`	`array_has_all`	Returns true if all elements of sub-list exist in list.	`list_has_all(l, [4,6])`	`true`
`list_has_any(list1, list2)`	`array_has_any`	Returns true if any elements exist is both lists.	`list_has_any([1,2,3], [2,3,4])`	`true`
`list_intersect(list1, list2)`	`array_intersect`	Returns a list of all the elements that exist in both l1 and l2, without duplicates.	`list_intersect([1,2,3], [2,3,4])`	`[2, 3]`

Function	Aliases	Description	Example	Result
`list_position(list, element)`	`list_indexof`, `array_position`, `array_indexof`	Returns the index of the element if the list contains the element.	`list_contains([1, 2, NULL], 2)`	`2`
`list_prepend(element, list)`	`array_prepend`, `array_push_front`	Prepends `element` to `list`.	`list_prepend(3, [4, 5, 6])`	`[3, 4, 5, 6]`
`list_resize(list, size[,value])`	`array_resize`	Resizes the list to contain `size` elements. Initializes new elements with `value` or `NULL` if `value` is not set.	`list_resize([1, 2, 3], 5, 0)`	`[1, 2, 3, 0, 0]`
`list_reverse_sort(list)`	`array_reverse_sort`	Sorts the elements of the list in reverse order. See the Sorting Lists section for more details about the null sorting order.	`list_reverse_sort([3, 6, 1, 2])`	`[6, 3, 2, 1]`
`list_reverse(list)`	`array_reverse`	Reverses the list.	`list_reverse(l)`	`[6, 5, 4]`
`list_slice(list, begin, end, step)`	`array_slice`	`list_slice` with added `step` feature.	`list_slice(l, 1, 3, 2)`	`[4, 6]`
`list_slice(list, begin, end)`	`array_slice`	Extract a sublist using slice conventions. Negative values are accepted. See slicing.	`list_slice(l, 2, 3)`	`[5, 6]`

Function	Aliases	Description	Example	Result
`list_sort(list)`	`array_sort`	Sorts the elements of the list. See the Sorting Lists section for more details about the sorting order and the null sorting order.	`list_sort([3, 6, 1, 2])`	`[1, 2, 3, 6]`
`list_transform(list, lambda)`	`array_transform`, `apply`, `list_apply`, `array_apply`	Returns a list that is the result of applying the lambda function to each element of the input list. See the Lambda Functions section for more details.	`list_transform(l, x -> x + 1)`	`[5, 6, 7]`
`list_unique(list)`	`array_unique`	Counts the unique elements of a list.	`list_unique([1, 1, NULL, -3, 1, 5])`	`3`
`list_value(any, ...)`	`list_pack`	Create a `LIST` containing the argument values.	`list_value(4, 5, 6)`	`[4, 5, 6]`
`unnest(list)`		Unnests a list by one level. Note that this is a special function that alters the cardinality of the result. See the UNNEST page for more details.	`unnest([1, 2, 3])`	`1, 2, 3`

List Operators

The following operators are supported for lists:

Operator	Description	Example	Result
`&&`	Alias for `list_intersect`	`[1, 2, 3, 4, 5] && [2, 5, 5, 6]`	`[2, 5]`
`@>`	Alias for `list_has_all`, where the list on the **right** of the operator is the sublist.	`[1, 2, 3, 4] @> [3, 4, 3]`	`true`
`<@`	Alias for `list_has_all`, where the list on the **left** of the operator is the sublist.	`[1, 4] <@ [1, 2, 3, 4]`	`true`
`||`	Alias for `list_concat`	`[1, 2, 3] || [4, 5, 6]`	`[1, 2, 3, 4, 5, 6]`
`<=>`	Alias for `list_cosine_similarity`	`[1, 2, 3] <=> [1, 2, 5]`	`0.9759000729485332`
`<->`	Alias for `list_distance`	`[1, 2, 3] <-> [1, 2, 5]`	`2.0`

List Comprehension

Python-style list comprehension can be used to compute expressions over elements in a list. For example:

```
SELECT [lower(x) for x in strings] FROM (VALUES (['Hello', '', 'World'])) t(strings);
-- ['hello', '', 'world']
SELECT [upper(x) for x in strings if len(x)>0] FROM (VALUES (['Hello', '', 'World'])) t(strings);
-- [HELLO, WORLD]
```

Struct Functions

Function	Description	Example	Result
struct.*entry*	Dot notation serves as an alias for struct_extract.	({'i': 3, 's': 'string'}).s	string
struct[*entry*]	Bracket notation serves as an alias for struct_extract.	({'i': 3, 's': 'string'})['s']	string
row(*any*, ...)	Create a STRUCT containing the argument values. If the values are column references, the entry name will be the column name; otherwise it will be the string 'vN' where N is the (1-based) position of the argument.	row(i, i % 4, i / 4)	{'i': 3, 'v2': 3, 'v3': 0}
struct_extract(*struct*, *'entry'*)	Extract the named entry from the struct.	struct_extract({'i': 3, 'v2': 3, 'v3': 0}, 'i')	3
struct_pack(*name* := *any*, ...)	Create a STRUCT containing the argument values. The entry name will be the bound variable name.	struct_pack(i := 4, s := 'string')	{'i': 4, 's': string}
struct_insert(*struct*, *name* := *any*, ...)	Add field(s)/value(s) to an existing STRUCT with the argument values. The entry name(s) will be the bound variable name(s).	struct_insert({'a': 1}, b := 2)	{'a': 1, 'b': 2}

Map Functions

Function	Description	Example	Result
`map[entry]`	Alias for `element_at`	`map([100, 5], ['a', 'b'])[100]`	`[a]`
`element_at(map, key)`	Return a list containing the value for a given key or an empty list if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.	`element_at(map([100, 5], [42, 43]), 100)`	`[42]`
`map_extract(map, key)`	Alias of `element_at`. Return a list containing the value for a given key or an empty list if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.	`map_extract(map([100, 5], [42, 43]), 100)`	`[42]`
`cardinality(map)`	Return the size of the map (or the number of entries in the map).	`cardinality(map([4, 2], ['a', 'b']))`	`2`
`map_from_entries(STRUCT(k, v)[])`	Returns a map created from the entries of the array	`map_from_entries([{k: 5, v: 'val1'}, {k: 3, v: 'val2'}])`	`{5=val1, 3=val2}`
`map()`	Returns an empty map.	`map()`	`{}`
`map_keys(map)`	Return a list of all keys in the map.	`map_keys(map([100, 5], [42, 43]))`	`[100, 5]`

Function	Description	Example	Result
`map_values(map)`	Return a list of all values in the map.	`map_values(map([100, 5], [42, 43]))`	`[42, 43]`
`map_entries(map)`	Return a list of struct(k, v) for each key-value pair in the map.	`map_entries(map([100, 5], [42, 43]))`	`[{'key': 100, 'value': 42}, {'key': 5, 'value': 43}]`

Union Functions

Function	Description	Example	Result
union.tag	Dot notation serves as an alias for `union_extract`.	`(union_value(k := 'hello')).k`	`string`
`union_extract(union, 'tag')`	Extract the value with the named tags from the union. `NULL` if the tag is not currently selected	`union_extract(s, 'k')`	`hello`
`union_value(tag := any)`	Create a single member `UNION` containing the argument value. The tag of the value will be the bound variable name.	`union_value(k := 'hello')`	`'hello'::UNION(k VARCHAR)`
`union_tag(union)`	Retrieve the currently selected tag of the union as an Enum.	`union_tag(union_value(k := 'foo'))`	`'k'`

Range Functions

The functions *range* and *generate_series* create a list of values in the range between `start` and `stop`. The `start` parameter is inclusive. For the `range` function, the `stop` parameter is exclusive, while for `generate_series`, it is inclusive.

Based on the number of arguments, the following variants exist:

- `range(`*start*`,`*stop*`,`*step*`)`
- `range(`*start*`,`*stop*`)`
- `range(`*stop*`)`
- `generate_series(`*start*`,`*stop*`,`*step*`)`
- `generate_series(`*start*`,`*stop*`)`
- `generate_series(`*stop*`)`

The default value of `start` is 0 and the default value of `step` is 1.

```
SELECT range(5);
-- [0, 1, 2, 3, 4]

SELECT range(2, 5);
-- [2, 3, 4]

SELECT range(2, 5, 3);
-- [2]

SELECT generate_series(5);
-- [0, 1, 2, 3, 4, 5]

SELECT generate_series(2, 5);
-- [2, 3, 4, 5]

SELECT generate_series(2, 5, 3);
-- [2, 5]
```

Date ranges are also supported:

```
SELECT * FROM range(date '1992-01-01', date '1992-03-01', interval '1'
↪  month);
```

range
1992-01-01 00:00:00

```
┌─────────────────────┐
│ 1992-02-01 00:00:00 │
└─────────────────────┘
```

Slicing

The function `list_slice` can be used to extract a sublist from a list. The following variants exist:

- `list_slice(`*`list`*`,`*`begin`*`,`*`end`*`)`
- `list_slice(`*`list`*`,`*`begin`*`,`*`end`*`)`
- `array_slice(`*`list`*`,`*`begin`*`,`*`end`*`,`*`step`*`)`
- `array_slice(`*`list`*`,`*`begin`*`,`*`end`*`,`*`step`*`)`
- `list[`*`begin`*`:`*`end`*`]`
- `list[`*`begin`*`:`*`end`*`:`*`step`*`]`

`list`

- Is the list to be sliced

`begin`

- Is the index of the first element to be included in the slice
- When `begin < 0` the index is counted from the end of the list
- When `begin < 0` and `-begin > length`, `begin` is clamped to the beginning of the list
- When `begin > length`, the result is an empty list
- **Bracket Notation:** When `begin` is omitted, it defaults to the beginning of the list

`end`

- Is the index of the last element to be included in the slice
- When `end < 0` the index is counted from the end of the list
- When `end > length`, end is clamped to `length`
- When `end < begin`, the result is an empty list
- **Bracket Notation:** When `end` is omitted, it defaults to the end of the list. When `end` is omitted and a `step` is provided, `end` must be replaced with a `-`

`step` *(optional)*

- Is the step size between elements in the slice
- When `step < 0` the slice is reversed, and `begin` and `end` are swapped
- Must be non-zero

```
SELECT list_slice([1, 2, 3, 4, 5], 2, 4);
-- [2, 3, 4]
```

```
SELECT ([1, 2, 3, 4, 5])[2:4:2];
-- [2, 4]

SELECT([1, 2, 3, 4, 5])[4:2:-2];
-- [4, 2]

SELECT ([1, 2, 3, 4, 5])[:];
-- [1, 2, 3, 4, 5]

SELECT ([1, 2, 3, 4, 5])[:-:2];
-- [1, 3, 5]

SELECT ([1, 2, 3, 4, 5])[:-:-2];
-- [5, 3, 1]
```

List Aggregates

The function `list_aggregate` allows the execution of arbitrary existing aggregate functions on the elements of a list. Its first argument is the list (column), its second argument is the aggregate function name, e.g., `min`, `histogram` or `sum`.

`list_aggregate` accepts additional arguments after the aggregate function name. These extra arguments are passed directly to the aggregate function, which serves as the second argument of `list_aggregate`.

```
SELECT list_aggregate([1, 2, -4, NULL], 'min');
-- -4

SELECT list_aggregate([2, 4, 8, 42], 'sum');
-- 56

SELECT list_aggregate([[1, 2], [NULL], [2, 10, 3]], 'last');
-- [2, 10, 3]

SELECT list_aggregate([2, 4, 8, 42], 'string_agg', '|');
-- 2|4|8|42
```

The following is a list of existing rewrites. Rewrites simplify the use of the list aggregate function by only taking the list (column) as their argument. `list_avg`, `list_var_samp`, `list_var_pop`, `list_stddev_pop`, `list_stddev_samp`, `list_sem`, `list_approx_count_distinct`, `list_bit_xor`, `list_bit_or`, `list_bit_and`, `list_bool_and`, `list_bool_or`, `list_`

`count`, `list_entropy`, `list_last`, `list_first`, `list_kurtosis`, `list_min`, `list_max`, `list_product`, `list_skewness`, `list_sum`, `list_string_agg`, `list_mode`, `list_median`, `list_mad` and `list_histogram`.

```
SELECT list_min([1, 2, -4, NULL]);
-- -4

SELECT list_sum([2, 4, 8, 42]);
-- 56

SELECT list_last([[1, 2], [NULL], [2, 10, 3]]);
-- [2, 10, 3]
```

array_to_string Concatenates list/array elements using an optional delimiter.

```
SELECT array_to_string([1, 2, 3], '-') AS str;
-- 1-2-3

-- this is equivalent to the following SQL
SELECT list_aggr([1, 2, 3], 'string_agg', '-') AS str;
-- 1-2-3
```

Sorting Lists

The function `list_sort` sorts the elements of a list either in ascending or descending order. In addition, it allows to provide whether NULL values should be moved to the beginning or to the end of the list.

By default if no modifiers are provided, DuckDB sorts ASC NULLS FIRST, i.e., the values are sorted in ascending order and NULL values are placed first. This is identical to the default sort order of SQLite. The default sort order can be changed using these PRAGMA statements.

`list_sort` leaves it open to the user whether they want to use the default sort order or a custom order. `list_sort` takes up to two additional optional parameters. The second parameter provides the sort order and can be either ASC or DESC. The third parameter provides the NULL sort order and can be either `NULLS FIRST` or `NULLS LAST`.

```
-- default sort order and default NULL sort order
SELECT list_sort([1, 3, NULL, 5, NULL, -5]);
----
[NULL, NULL, -5, 1, 3, 5]
```

```
-- only providing the sort order
SELECT list_sort([1, 3, NULL, 2], 'ASC');
----
[NULL, 1, 2, 3]

-- providing the sort order and the NULL sort order
SELECT list_sort([1, 3, NULL, 2], 'DESC', 'NULLS FIRST');
----
[NULL, 3, 2, 1]
```

`list_reverse_sort` has an optional second parameter providing the NULL sort order. It can be either `NULLS FIRST` or `NULLS LAST`.

```
-- default NULL sort order
SELECT list_sort([1, 3, NULL, 5, NULL, -5]);
----
[NULL, NULL, -5, 1, 3, 5]

-- providing the NULL sort order
SELECT list_reverse_sort([1, 3, NULL, 2], 'NULLS LAST');
----
[3, 2, 1, NULL]
```

Lambda Functions

`(parameter1, parameter2, ...) -> expression`. If the lambda function has only one parameter, then the brackets can be omitted. The parameters can have any names.

```
param -> param > 1
duck -> CONTAINS(CONCAT(duck, 'DB'), 'duck')
(x, y) -> x + y
```

Transform

`list_transform(list, lambda)`

Returns a list that is the result of applying the lambda function to each element of the input list. The lambda function must have exactly one left-hand side parameter. The return type of the lambda function defines the type of the list elements.

```
-- incrementing each list element by one
SELECT list_transform([1, 2, NULL, 3], x -> x + 1);
----
```

```
[2, 3, NULL, 4]

-- transforming strings
SELECT list_transform(['duck', 'a', 'b'], duck -> CONCAT(duck, 'DB'));
----
[duckDB, aDB, bDB]

-- combining lambda functions with other functions
SELECT list_transform([5, NULL, 6], x -> COALESCE(x, 0) + 1);
----
[6, 1, 7]
```

Filter

`list_filter(list, lambda)`

Constructs a list from those elements of the input list for which the lambda function returns true. The lambda function must have exactly one left-hand side parameter and its return type must be of type BOOLEAN.

```
-- filter out negative values
SELECT list_filter([5, -6, NULL, 7], x -> x > 0);
----
[5, 7]

-- divisible by 2 and 5
SELECT list_filter(list_filter([2, 4, 3, 1, 20, 10, 3, 30], x -> x % 2 ==
 ↪  0), y -> y % 5 == 0);
----
[20, 10, 30]

-- in combination with range(...) to construct lists
SELECT list_filter([1, 2, 3, 4], x -> x > #1) FROM range(4);
----
[1, 2, 3, 4]
[2, 3, 4]
[3, 4]
[4]
[]
```

Lambda functions can be arbitrarily nested.

```
-- nested lambda functions to get all squares of even list elements
```

```
SELECT list_transform(list_filter([0, 1, 2, 3, 4, 5], x -> x % 2 = 0), y ->
 ↪  y * y);
----
[0, 4, 16]
```

Flatten

The flatten function is a scalar function that converts a list of lists into a single list by concatenating each sub-list together. Note that this only flattens one level at a time, not all levels of sub-lists.

```
-- Convert a list of lists into a single list
SELECT
    flatten([
        [1, 2],
        [3, 4]
    ]);
----
[1, 2, 3, 4]

-- If the list has multiple levels of lists,
-- only the first level of sub-lists is concatenated into a single list
SELECT
    flatten([
        [
            [1, 2],
            [3, 4],
        ],
        [
            [5, 6],
            [7, 8],
        ]
    ]);
----
[[1, 2], [3, 4], [5, 6], [7, 8]]
```

In general, the input to the flatten function should be a list of lists (not a single level list). However, the behavior of the flatten function has specific behavior when handling empty lists and NULL values.

```
-- If the input list is empty, return an empty list
SELECT flatten([]);
----
[]
```

```
-- If the entire input to flatten is NULL, return NULL
SELECT flatten(NULL);
----
NULL

-- If a list whose only entry is NULL is flattened, return an empty list
SELECT flatten([NULL]);
----
[]

-- If the sub-list in a list of lists only contains NULL,
-- do not modify the sub-list
-- (Note the extra set of parentheses vs. the prior example)
SELECT flatten([[NULL]]);
----
[NULL]

-- Even if the only contents of each sub-list is NULL,
-- still concatenate them together
-- Note that no de-duplication occurs when flattening.
-- See list_distinct function for de-duplication.
SELECT flatten([[NULL],[NULL]]);
----
[NULL, NULL]
```

generate_subscripts

The `generate_subscript(`*`arr`*`,`*`dim`*`)` function generates indexes along the `dim`th dimension of array `arr`.

```
SELECT generate_subscripts([4, 5, 6], 1) AS i;
```

i
1
2
3

Related Functions

There are also aggregate functions `list` and `histogram` that produces lists and lists of structs. UNNEST is used to unnest a list by one level.

Numeric Functions

Numeric Operators

The table below shows the available mathematical operators for numeric types.

Operator	Description	Example	Result
+	addition	`2 + 3`	5
−	subtraction	`2 - 3`	-1
*	multiplication	`2 * 3`	6
/	float division	`5 / 2`	2.5
//	division	`5 // 2`	2
%	modulo (remainder)	`5 % 4`	1
**	exponent	`3 ** 4`	81
^	exponent (alias for **)	`3 ^ 4`	81
&	bitwise AND	`91 & 15`	11
|	bitwise OR	`32 | 3`	35
<<	bitwise shift left	`1 << 4`	16
>>	bitwise shift right	`8 >> 2`	2
~	bitwise negation	`~15`	-16
!	factorial of x. Computes the product of the current integer and all integers below it	`4!`	24

There are two division operators: `/` and `//`. They are equivalent when at least one of the operands is a FLOAT or a DOUBLE. When both operands are integers, `/` performs floating points division (`5 / 2 = 2.5`) while `//` performs integer division (`5 // 2 = 2`).

The modulo, bitwise, and negation and factorial operators work only on integral data types, whereas the others are available for all numeric data types.

Numeric Functions

The table below shows the available mathematical functions.

Function	Description	Example	Result
`abs(x)`	absolute value	`abs(-17.4)`	17.4
`acos(x)`	computes the arccosine of x	`acos(0.5)`	1.0471975511965976
`asin(x)`	computes the arcsine of x	`asin(0.5)`	0.5235987755982989
`atan(x)`	computes the arctangent of x	`atan(0.5)`	0.4636476090008061
`atan2(y, x)`	computes the arctangent (y, x)	`atan2(0.5, 0.5)`	0.7853981633974483
`bit_count(x)`	returns the number of bits that are set	`bit_count(31)`	5
`cbrt(x)`	returns the cube root of the number	`cbrt(8)`	2
`ceil(x)`	rounds the number up	`ceil(17.4)`	18
`ceiling(x)`	rounds the number up. Alias of `ceil`.	`ceiling(17.4)`	18
`cos(x)`	computes the cosine of x	`cos(90)`	-0.4480736161291701
`cot(x)`	computes the cotangent of x	`cot(0.5)`	1.830487721712452
`degrees(x)`	converts radians to degrees	`degrees(pi())`	180
`even(x)`	round to next even number by rounding away from zero.	`even(2.9)`	4

Function	Description	Example	Result
`exp(x)`	computes `e ** x`	`exp(0.693)`	2
`factorial(x)`	See `!` operator. Computes the product of the current integer and all integers below it	`factorial(4)`	24
`floor(x)`	rounds the number down	`floor(17.4)`	17
`gamma(x)`	interpolation of (x-1) factorial (so decimal inputs are allowed)	`gamma(5.5)`	52.34277778455352
`gcd(x, y)`	computes the greatest common divisor of x and y	`gcd(42, 57)`	3
`greatest_common_divisor(x, y)`	computes the greatest common divisor of x and y	`greatest_common_divisor(42, 57)`	3
`greatest(x1, x2, ...)`	selects the largest value	`greatest(3, 2, 4, 4)`	4
`isfinite(x)`	Returns true if the floating point value is finite, false otherwise	`isfinite(5.5)`	`true`
`isinf(x)`	Returns true if the floating point value is infinite, false otherwise	`isinf('Infinity'::float)`	`true`
`isnan(x)`	Returns true if the floating point value is not a number, false otherwise	`isnan('NaN'::float)`	`true`
`lcm(x, y)`	computes the least common multiple of x and y	`lcm(42, 57)`	798

Function	Description	Example	Result
`least_common_multiple(x, y)`	computes the least common multiple of x and y	`least_common_multiple(42, 57)`	798
`least(x1, x2, ...)`	selects the smallest value	`least(3, 2, 4, 4)`	2
`lgamma(x)`	computes the log of the gamma function.	`lgamma(2)`	0
`ln(x)`	computes the natural logarithm of *x*	`ln(2)`	0.693
`log(x)`	computes the 10-log of *x*	`log(100)`	2
`log2(x)`	computes the 2-log of *x*	`log2(8)`	3
`log10(x)`	alias of `log`. computes the 10-log of *x*	`log10(1000)`	3
`nextafter(x, y)`	return the next floating point value after *x* in the direction of *y*	`nextafter(1::float, 2::float)`	1.0000001
`pi()`	returns the value of pi	`pi()`	3.141592653589793
`pow(x, y)`	computes x to the power of y	`pow(2, 3)`	8
`power(x, y)`	Alias of pow. computes x to the power of y	`power(2, 3)`	8
`radians(x)`	converts degrees to radians	`radians(90)`	1.5707963267948966
`random()`	returns a random number between 0 and 1	`random()`	various
`round(v numeric, s int)`	round to *s* decimal places, values *s < 0* are allowed	`round(42.4332, 2)`	42.43
`setseed(x)`	sets the seed to be used for the random function	`setseed(0.42)`	

Function	Description	Example	Result
`sin(x)`	computes the sin of x	`sin(90)`	0.8939966636005579
`sign(x)`	returns the sign of x as -1, 0 or 1	`sign(-349)`	-1
`signbit(x)`	returns whether the signbit is set or not	`signbit(-0.0)`	`true`
`sqrt(x)`	returns the square root of the number	`sqrt(9)`	3
`xor(x)`	bitwise XOR	`xor(17, 5)`	20
`tan(x)`	computes the tangent of x	`tan(90)`	-1.995200412208242
`@`	absolute value (parentheses optional if operating on a column)	`@(-2)`	2

Pattern Matching

There are four separate approaches to pattern matching provided by DuckDB: the traditional SQL `LIKE` operator, the more recent `SIMILAR TO` operator (added in SQL:1999), a `GLOB` operator, and POSIX-style regular expressions.

LIKE

The `LIKE` expression returns `true` if the string matches the supplied pattern. (As expected, the `NOT LIKE` expression returns `false` if `LIKE` returns `true`, and vice versa. An equivalent expression is `NOT (string LIKE pattern)`.)

If pattern does not contain percent signs or underscores, then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore (_) in pattern stands for (matches) any single character; a percent sign (%) matches any sequence of zero or more characters.

`LIKE` pattern matching always covers the entire string. Therefore, if it's desired to match a sequence anywhere within a string, the pattern must start and end with a percent sign.

Some examples:

```
'abc' LIKE 'abc' -- true
'abc' LIKE 'a%'  -- true
'abc' LIKE '_b_' -- true
'abc' LIKE 'c'   -- false
'abc' LIKE 'c%'  -- false
'abc' LIKE '%c'  -- true
'abc' NOT LIKE '%c'  -- false
```

The keyword `ILIKE` can be used instead of `LIKE` to make the match case-insensitive according to the active locale.

```
'abc' ILIKE '%C' -- true
'abc' NOT ILIKE '%C' -- false
```

To search within a string for a character that is a wildcard (% or _), the pattern must use an `ESCAPE` clause and an escape character to indicate the wildcard should be treated as a literal character instead of a wildcard. See an example below.

Additionally, the function `like_escape` has the same functionality as a `LIKE` expression with an `ESCAPE` clause, but using function syntax. See the Text Functions Docs for details.

```
--Search for strings with 'a' then a literal percent sign then 'c'
'a%c' LIKE 'a$%c' ESCAPE '$'        -- true
'azc' LIKE 'a$%c' ESCAPE '$'        -- false

--Case insensitive ILIKE with ESCAPE
'A%c' ILIKE 'a$%c' ESCAPE '$';      --true
```

There are also alternative characters that can be used as keywords in place of `LIKE` expressions. These enhance PostgreSQL compatibility.

LIKE-style	PostgreSQL-style
`LIKE`	`~~`
`NOT LIKE`	`!~~`
`ILIKE`	`~~*`
`NOT ILIKE`	`!~~*`

SIMILAR TO

The `SIMILAR TO` operator returns true or false depending on whether its pattern matches the given string. It is similar to `LIKE`, except that it interprets the pattern using a regular expression. Like `LIKE`,

the `SIMILAR TO` operator succeeds only if its pattern matches the entire string; this is unlike common regular expression behavior where the pattern can match any part of the string.

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a regular set). A string is said to match a regular expression if it is a member of the regular set described by the regular expression. As with `LIKE`, pattern characters match string characters exactly unless they are special characters in the regular expression language — but regular expressions use different special characters than `LIKE` does.

Some examples:

```
'abc' SIMILAR TO 'abc'       -- true
'abc' SIMILAR TO 'a'         -- false
'abc' SIMILAR TO '.*(b|d).*' -- true
'abc' SIMILAR TO '(b|c).*'   -- false
'abc' NOT SIMILAR TO 'abc'   -- false
```

There are also alternative characters that can be used as keywords in place of `SIMILAR TO` expressions. These follow POSIX syntax.

SIMILAR TO-style	POSIX-style
`SIMILAR TO`	`~`
`NOT SIMILAR TO`	`!~`

GLOB

The `GLOB` operator returns `true` or `false` if the string matches the `GLOB` pattern. The `GLOB` operator is most commonly used when searching for filenames that follow a specific pattern (for example a specific file extension). Use the question mark (`?`) wildcard to match any single character, and use the asterisk (`*`) to match zero or more characters. In addition, use bracket syntax (`[ ]`) to match any single character contained within the brackets, or within the character range specified by the brackets. An exclamation mark (`!`) may be used inside the first bracket to search for a character that is not contained within the brackets. To learn more, visit the Glob Programming Wikipedia page.

Some examples:

```
'best.txt' GLOB '*.txt'           -- true
'best.txt' GLOB '????.txt'        -- true
'best.txt' GLOB '?.txt'           -- false
'best.txt' GLOB '[abc]est.txt'    -- true
'best.txt' GLOB '[a-z]est.txt'    -- true
```

```
-- The bracket syntax is case sensitive
'Best.txt' GLOB '[a-z]est.txt'     -- false
'Best.txt' GLOB '[a-zA-Z]est.txt'  -- true

-- The ! applies to all characters within the brackets
'Best.txt' GLOB '[!a-zA-Z]est.txt' -- false

-- To negate a GLOB operator, negate the entire expression
-- (NOT GLOB is not valid syntax)
NOT 'best.txt' GLOB '*.txt'        -- false
```

Three tildes (~~~) may also be used in place of the GLOB keyword.

GLOB-style	Symbolic-style
GLOB	~~~

Glob Function to Find Filenames The glob pattern matching syntax can also be used to search for filenames using the `glob` table function. It accepts one parameter: the path to search (which may include glob patterns).

```
-- Search the current directory for all files
SELECT * FROM glob('*');
```

file
duckdb.exe
test.csv
test.json
test.parquet
test2.csv
test2.parquet
todos.json

Regular Expressions

Function	Description	Example	Result
`regexp_full_match(string, regex)`	Returns `true` if the entire *string* matches the *regex*	`regexp_full_match('anabanana', '(an)*')`	`false`
`regexp_matches(string, pattern)`	Returns `true` if *string* contains the regexp *pattern*, `false` otherwise	`regexp_matches('anabanana', '(an)*')`	`true`
`regexp_replace(string, pattern, replacement);`	If *string* contains the regexp *pattern*, replaces the matching part with *replacement*	`select regexp_replace('hello', '[lo]', '-')`	`he-lo`
`regexp_split_to_array(string, regex)`	Alias of `string_split_regex`. Splits the *string* along the *regex*	`regexp_split_to_array('hello␣world;42', ';?␣')`	`['hello', 'world', '42']`
`regexp_extract(string, pattern[,idx]);`	If *string* contains the regexp *pattern*, returns the capturing group specified by optional parameter *idx*	`regexp_extract('hello_world', '([a-z]+)_?', 1)`	`hello`
`regexp_extract(string, pattern, name_list);`	If *string* contains the regexp *pattern*, returns the capturing groups as a struct with corresponding names from *name_list*	`regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd'])`	`{'y':'2023', 'm':'04', 'd':'15'}`
`regexp_extract_all(string, regex[,group= 0])`	Split the *string* along the *regex* and extract all occurrences of *group*	`regexp_extract_all('hello_world', '([a-z]+)_?', 1)`	`[hello, world]`

The `regexp_matches` function is similar to the `SIMILAR TO` operator, however, it does not require the entire string to match. Instead, `regexp_matches` returns `true` if the string merely contains the

pattern (unless the special tokens ^ and $ are used to anchor the regular expression to the start and end of the string). Below are some examples:

```
regexp_matches('abc', 'abc')       -- true
regexp_matches('abc', '^abc$')     -- true
regexp_matches('abc', 'a')         -- true
regexp_matches('abc', '^a$')       -- false
regexp_matches('abc', '.*(b|d).*') -- true
regexp_matches('abc', '(b|c).*')   -- true
regexp_matches('abc', '^(b|c).*')  -- false
regexp_matches('abc', '(?i)A')     -- true
```

The `regexp_matches` function also supports the following options.

Option	Description
`'c'`	case-sensitive matching
`'i'`	case-insensitive matching
`'l'`	match literals instead of regular expression tokens
`'m','n','p'`	newline sensitive matching
`'s'`	non-newline sensitive matching
`'g'`	global replace, only available for regexp_replace

```
regexp_matches('abcd', 'ABC', 'c')-- false
regexp_matches('abcd', 'ABC', 'i') -- true
regexp_matches('ab^/$cd', '^/$', 'l') -- true
regexp_matches('hello\nworld', 'hello.world', 'p') -- false
regexp_matches('hello\nworld', 'hello.world', 's') -- true
```

The `regexp_matches` operator will be optimized to the `LIKE` operator when possible. To achieve the best results, the `'s'` option should be passed. By default the RE2 library doesn't match '.' to newline.

Original	Optimized equivalent
`regexp_matches('hello world', '^hello', 's')`	`prefix('hello world', 'hello')`
`regexp_matches('hello world', 'world$', 's')`	`suffix('hello world', 'world')`

Original	Optimized equivalent
`regexp_matches('hello world', 'hello.world', 's')`	`LIKE 'hello_world'`
`regexp_matches('hello world', 'he.*rld', 's')`	`LIKE '%he%rld'`

The `regexp_replace` function can be used to replace the part of a string that matches the regexp pattern with a replacement string. The notation `\d` (where d is a number indicating the group) can be used to refer to groups captured in the regular expression in the replacement string. Below are some examples:

```
regexp_replace('abc', '(b|c)', 'X')        -- aXc
regexp_replace('abc', '(b|c)', '\1\1\1\1') -- abbbbc
regexp_replace('abc', '(.*)c', '\1e')      -- abe
regexp_replace('abc', '(a)(b)', '\2\1')    -- bac
```

The `regexp_extract` function is used to extract a part of a string that matches the regexp pattern. A specific capturing group within the pattern can be extracted using the *idx* parameter. If *idx* is not specified, it defaults to 0, extracting the first match with the whole pattern.

```
regexp_extract('abc', '.b.')     -- abc
regexp_extract('abc', '.b.', 0)  -- abc
regexp_extract('abc', '.b.', 1)  -- (empty)
regexp_extract('abc', '([a-z])(b)', 1) -- a
regexp_extract('abc', '([a-z])(b)', 2) -- b
```

If *ids* is a `LIST` of strings, then `regexp_extract` will return the corresponding capture groups as fields of a `STRUCT`:

```
regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd']) --
↪ {'y':'2023', 'm':'04', 'd':'15'}
regexp_extract('2023-04-15 07:59:56', '^(\d+)-(\d+)-(\d+)
↪ (\d+):(\d+):(\d+)', ['y', 'm', 'd']) -- {'y':'2023', 'm':'04', 'd':'15'}
regexp_extract('duckdb_0_7_1', '^(\w+)_(\d+)_(\d+)', ['tool', 'major',
↪ 'minor', 'fix']) -- error
```

If the number of column names is less than the number of capture groups, then only the first groups are returned. If the number of column names is greater, then an error is generated.

DuckDB uses RE2 as its regex engine. For more information see the RE2 docs

Text Functions

This section describes functions and operators for examining and manipulating string values. The ␣ symbol denotes a space character.

Function	Description	Example	Result	Alias
string ^@ *search_string*	Alias for starts_with.	'abc' ^@ 'a'	true	
string || *string*	String concatenation	'Duck' || 'DB'	DuckDB	
string[*index*]	Alias for array_extract.	'DuckDB'[4]	'k'	
string[*begin*:*end*]	Alias for array_slice. Missing begin or end arguments are interpreted as the beginning or end of the list respectively.	'DuckDB'[:4]	'Duck'	
array_extract(*list*, *index*)	Extract a single character using a (1-based) index.	array_extract('DuckDB', 2)	'u'	list_element, list_extract
array_slice(*list*, *begin*, *end*)	Extract a string using slice conventions. Negative values are accepted.	array_slice('DuckDB', 5, NULL)	'DB'	

Function	Description	Example	Result	Alias
`ascii(`*`string`*`)`	Returns an integer that represents the Unicode code point of the first character of the *string*	`ascii('Ω')`	`937`	
`bar(`*`x`*`,`*`min`*`,`*`max`*`[,`*`width`*`])`	Draw a band whose width is proportional to (*x* - *min*) and equal to *width* characters when *x* = *max*. *width* defaults to 80.	`bar(5, 0, 20, 10)`		
`bit_length(`*`string`*`)`	Number of bits in a string.	`bit_length('abc')`	`24`	
`chr(`*`x`*`)`	returns a character which is corresponding the ASCII code value or Unicode code point	`chr(65)`	`A`	
`concat(`*`string`*`, ...)`	Concatenate many strings together	`concat('Hello', ' ', 'World')`	`Hello World`	
`concat_ws(`*`separator`*`,`*`string`*`, ...)`	Concatenate strings together separated by the specified separator	`concat_ws(', ', 'Banana', 'Apple', 'Melon')`	`Banana, Apple, Melon`	

Function	Description	Example	Result	Alias
`contains(`*`string`*`,` *`search_string`*`)`	Return true if *search_string* is found within *string*	`contains('abc', 'a')`	`true`	
`ends_with(`*`string`*`,` *`search_string`*`)`	Return true if *string* ends with *search_string*	`ends_with('abc', 'c')`	`true`	`suffix`
`format(`*`format`*`,` *`parameters...`*`)`	Formats a string using the fmt syntax	`format('Benchmark "{}" took {} seconds', 'CSV', 42)`	`Benchmark "CSV" took 42 seconds`	
`from_base64(`*`string`*`)`	Convert a base64 encoded string to a character string.	`from_base64('QQ==')`	`'A'`	
`hash(`*`value`*`)`	Returns an integer with the hash of the *value*	`hash('🦆')`	`2595805878642663834`	
`instr(`*`string`*`,` *`search_string`*`)`	Return location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.	`instr('test test', 'es')`	`2`	
`left(`*`string`*`,` *`count`*`)`	Extract the left-most count characters	`left('Hello🦆', 2)`	`He`	

Function	Description	Example	Result	Alias
`left_grapheme(string, count)`	Extract the left-most grapheme clusters	`left_grapheme('🤦🏼‍♂️🤦🏾‍♀️', 1)`	🤦🏼‍♂️	
`length(string)`	Number of characters in *string*	`length('Hello🦆')`	6	
`length_grapheme(string)`	Number of grapheme clusters in *string*	`length_grapheme('🤦🏼‍♂️🤦🏾‍♀️')`	2	
string`LIKE`*target*	Returns true if the *string* matches the like specifier (see Pattern Matching)	`'hello' LIKE '%lo'`	`true`	
`like_escape(string, like_specifier, escape_character)`	Returns true if the *string* matches the *like_specifier* (see Pattern Matching). *escape_character* is used to search for wildcard characters in the *string*.	`like_escape('a%c', 'a$%c', '$')`	`true`	
`lower(string)`	Convert *string* to lower case	`lower('Hello')`	`hello`	`lcase`

Function	Description	Example	Result	Alias
`lpad(`*`string`*`,`*`count`*`,`*`character`*`)`	Pads the *string* with the character from the left until it has count characters	`lpad('hello', 10, '>')`	`>>>>>hello`	
`ltrim(`*`string`*`)`	Removes any spaces from the left side of the *string*	`ltrim('␣␣␣␣test␣␣')`	`test␣␣`	
`ltrim(`*`string`*`,`*`characters`*`)`	Removes any occurrences of any of the *characters* from the left side of the *string*	`ltrim('>>>>test<<', '><')`	`test<<`	
`md5(`*`value`*`)`	Returns the MD5 hash of the *value*	`md5('123')`	`'202cb962ac59075b964b07152d234b70'`	
`nfc_normalize(`*`string`*`)`	Convert string to Unicode NFC normalized string. Useful for comparisons and ordering if text data is mixed between NFC normalized and not.	`nfc_normalize('ardèch')`	`arde`ch`	

Function	Description	Example	Result	Alias
`not_like_escape(string, like_specifier, escape_character)`	Returns false if the *string* matches the *like_specifier* (see Pattern Matching). *escape_character* is used to search for wildcard characters in the *string*.	`like_escape('a%c', 'a$%c', '$')`	`true`	
`ord(string)`	Return ASCII character code of the leftmost character in a string.	`ord('ü')`	`252`	
`position(search_stringinstring)`	Return location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.	`position('b' in 'abc')`	`2`	
`printf(format, parameters...)`	Formats a *string* using printf syntax	`printf('Benchmark "%s" took %d seconds', 'CSV', 42)`	`Benchmark "CSV" took 42 seconds`	

Function	Description	Example	Result	Alias
`regexp_full_match(string, regex)`	Returns true if the entire *string* matches the *regex* (see Pattern Matching)	`regexp_full_match('anabanana', '(an)*')`	`false`	
`regexp_matches(string, regex)`	Returns true if a part of *string* matches the *regex* (see Pattern Matching)	`regexp_matches('anabanana', '(an)*')`	`true`	
`regexp_replace(string, regex, replacement, modifiers)`	Replaces the first occurrence of *regex* with the *replacement*, use `'g'` modifier to replace all occurrences instead (see Pattern Matching)	`select regexp_replace('hello', '[lo]', '-')`	`he-lo`	
`regexp_extract(string, regex[,group=0])`	Split the *string* along the *regex* and extract first occurrence of *group*	`regexp_extract('hello_world', '([a-z]+)_?', 1)`	`hello`	
`regexp_extract_all(string, regex[,group=0])`	Split the *string* along the *regex* and extract all occurrences of *group*	`regexp_extract_all('hello_world', '([a-z]+)_?', 1)`	`[hello, world]`	

Function	Description	Example	Result	Alias
`repeat(string, count)`	Repeats the *string count* number of times	`repeat('A', 5)`	AAAAA	
`replace(string, source, target)`	Replaces any occurrences of the *source* with *target* in *string*	`replace('hello', 'l', '-')`	he--o	
`reverse(string)`	Reverses the *string*	`reverse('hello')`	olleh	
`right(string, count)`	Extract the right-most *count* characters	`right('Hello🦆', 3)`	lo🦆	
`right_grapheme(string, count)`	Extract the right-most *count* grapheme clusters	`right_grapheme('🤦🏼‍♂️🤦🏾‍♀️', 1)`	🤦🏾‍♀️	
`rpad(string, count, character)`	Pads the *string* with the character from the right until it has *count* characters	`rpad('hello', 10, '<')`	hello<<<<<	
`rtrim(string)`	Removes any spaces from the right side of the *string*	`rtrim('␣␣␣␣test␣␣ ␣␣␣␣test`	␣␣␣␣test␣␣ ␣␣␣␣test	
`rtrim(string, characters)`	Removes any occurrences of any of the *characters* from the right side of the *string*	`rtrim('>>>>test<< somethingwithmanyx>>>test', '><')`	somethingwithmanyx>>>test	

Function	Description	Example	Result	Alias
`split_part(`*`string`*`,`*`separator`*`,`*`index`*`)`	Split the *string* along the *separator* and return the data at the (1-based) *index* of the list. If the *index* is outside the bounds of the list, return an empty string (to match PostgreSQL's behavior).	`split_part('a|b|c', '|', 2)`	b	
`starts_with(`*`string`*`,`*`search_string`*`)`	Return true if *string* begins with *search_string*	`starts_with('abc', 'a')`	`true`	
`string``SIMILAR TO`*`regex`*	Returns `true` if the *string* matches the *regex*; identical to `regexp_full_match` (see Pattern Matching)	`'hello' SIMILAR TO 'l+'`	`false`	
`strlen(`*`string`*`)`	Number of bytes in *string*	`strlen('🦆')`	4	

Function	Description	Example	Result	Alias
`strpos(`*`string`*`,`*`search_string`*`)`	Alias of `instr`. Return location of first occurrence of *search_string* in *string*, counting from 1. Returns 0 if no match found.	`strpos('test test', 'es')`	2	
`strip_accents(`*`string`*`)`	Strips accents from *string*	`strip_accents('mühleisen')`	muhleisen	
`string_split(`*`string`*`,`*`separator`*`)`	Splits the *string* along the *separator*	`string_split('hello␣world', '␣')`	['hello', 'world']	`str_split`, `string_to_array`
`string_split_regex(`*`string`*`,`*`regex`*`)`	Splits the *string* along the *regex*	`string_split_regex('hello␣world; 42', ';?␣')`	['hello', 'world', '42']	`regexp_split_to_array`, `str_split_regex`
`substring(`*`string`*`,`*`start`*`,`*`length`*`)`	Extract substring of *length* characters starting from character *start*. Note that a *start* value of 1 refers to the *first* character of the string.	`substring('Hello', 2, 2)`	el	`substr`

Function	Description	Example	Result	Alias
`substring_grapheme(`*`string`*`,`*`start`*`,`*`length`*`)`	Extract substring of *length* grapheme clusters starting from character *start*. Note that a *start* value of 1 refers to the *first* character of the string.	`substring_grapheme('🏃🤦🏻‍♂️🤦🏾‍♀️🏃', 3, 2)`	🤦🏾‍♀️🏃	
`to_base64(`*`blob`*`)`	Convert a blob to a base64 encoded string.	`to_base64('A'::blob)`	QQ==	base64
`trim(`*`string`*`)`	Removes any spaces from either side of the *string*	`trim('␣␣␣␣test␣␣')`	test	
`trim(`*`string`*`,`*`characters`*`)`	Removes any occurrences of any of the *characters* from either side of the *string*	`trim('>>>>test<<', '><')`	test	
`unicode(`*`string`*`)`	Returns the unicode code of the first character of the *string*	`unicode('ü')`	252	
`upper(`*`string`*`)`	Convert *string* to upper case	`upper('Hello')`	HELLO	ucase

Text Similarity Functions

These functions are used to measure the similarity of two strings using various metrics.

Function	Description	Example	Result
`editdist3(`*`string`*`,`*`string`*`)`	Alias of `levenshtein` for SQLite compatibility. The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Different case is considered different.	`editdist3('duck', 'db')`	3
`hamming(`*`string`*`,`*`string`*`)`	The number of positions with different characters for 2 strings of equal length. Different case is considered different.	`hamming('duck', 'luck')`	1
`jaccard(`*`string`*`,`*`string`*`)`	The Jaccard similarity between two strings. Different case is considered different. Returns a number between 0 and 1.	`jaccard('duck', 'luck')`	0.6
`jaro_similarity(`*`string`*`,`*`string`*`)`	The Jaro similarity between two strings. Different case is considered different. Returns a number between 0 and 1.	`jaro_similarity('duck', 'duckdb')`	0.88

Function	Description	Example	Result
`jaro_winkler_similarity(`*`string`*`,`*`string`*`)`	The Jaro-Winkler similarity between two strings. Different case is considered different. Returns a number between 0 and 1.	`jaro_winkler_similarity('duck', 'duckdb')`	0.93
`levenshtein(`*`string`*`,`*`string`*`)`	The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Different case is considered different.	`levenshtein('duck', 'db')`	3
`damerau_levenshtein(`*`string`*`,`*`string`*`)`	Extension of Levenshtein distance to also include transposition of adjacent characters as an allowed edit operation. In other words, the minimum number of edit operations (insertions, deletions, substitutions or transpositions) required to change one string to another. Different case is considered different.	`damerau_levenshtein('duckdb', 'udckbd')`	2

Function	Description	Example	Result
`mismatches(`*`string`*`,`*`string`*`)`	The number of positions with different characters for 2 strings of equal length. Different case is considered different.	`mismatches('duck', 'luck')`	1

Time Functions

This section describes functions and operators for examining and manipulating `TIME` values.

Time Operators

The table below shows the available mathematical operators for `TIME` types.

Operator	Description	Example	Result
+	addition of an `INTERVAL`	`TIME '01:02:03' + INTERVAL 5 HOUR`	06:02:03
-	subtraction of an `INTERVAL`	`TIME '06:02:03' - INTERVAL 5 HOUR'`	01:02:03

Time Functions

The table below shows the available scalar functions for `TIME` types.

Function	Description	Example	Result
`current_time`/`get_current_time()`	Current time (start of current transaction)		

Function	Description	Example	Result
`date_diff(`*`part`*`,`*`starttime`*`,`*`endtime`*`)`	The number of partition boundaries between the times	`date_diff('hour', TIME '01:02:03', TIME '06:01:03')`	`5`
`datediff(`*`part`*`,`*`starttime`*`,`*`endtime`*`)`	Alias of date_diff. The number of partition boundaries between the times	`datediff('hour', TIME '01:02:03', TIME '06:01:03')`	`5`
`date_part(`*`part`*`,`*`time`*`)`	Get subfield (equivalent to *extract*)	`date_part('minute', TIME '14:21:13')`	`21`
`datepart(`*`part`*`,`*`time`*`)`	Alias of date_part. Get subfield (equivalent to *extract*)	`datepart('minute', TIME '14:21:13')`	`21`
`date_sub(`*`part`*`,`*`starttime`*`,`*`endtime`*`)`	The number of complete partitions between the times	`date_sub('hour', TIME '01:02:03', TIME '06:01:03')`	`4`
`datesub(`*`part`*`,`*`starttime`*`,`*`endtime`*`)`	Alias of date_sub. The number of complete partitions between the times	`datesub('hour', TIME '01:02:03', TIME '06:01:03')`	`4`
`extract(`*`part`* `FROM` *`time`*`)`	Get subfield from a time	`extract('hour' FROM TIME '14:21:13')`	`14`
`make_time(`*`bigint`*`,`*`bigint`*`,`*`double`*`)`	The time for the given parts	`make_time(13, 34, 27.123456)`	`13:34:27.123456`

The only date parts that are defined for times are `epoch`, `hours`, `minutes`, `seconds`, `milliseconds` and `microseconds`.

Timestamp Functions

This section describes functions and operators for examining and manipulating `TIMESTAMP` values.

Timestamp Operators

The table below shows the available mathematical operators for `TIMESTAMP` types.

Operator	Description	Example	Result
`+`	addition of an `INTERVAL`	`TIMESTAMP '1992-03-22 01:02:03' + INTERVAL 5 DAY`	`1992-03-27 01:02:03`
`-`	subtraction of `TIMESTAMP`s	`TIMESTAMP '1992-03-27' - TIMESTAMP '1992-03-22'`	`5 days`
`-`	subtraction of an `INTERVAL`	`TIMESTAMP '1992-03-27 01:02:03' - INTERVAL 5 DAY`	`1992-03-22 01:02:03`

Adding to or subtracting from infinite values produces the same infinite value.

Timestamp Functions

The table below shows the available scalar functions for `TIMESTAMP` values.

Function	Description	Example	Result
`age(`*`timestamp`*`,` *`timestamp`*`)`	Subtract arguments, resulting in the time difference between the two timestamps	`age(TIMESTAMP '2001-04-10', TIMESTAMP '1992-09-20')`	`8 years 6 months 20 days`
`age(`*`timestamp`*`)`	Subtract from current_date	`age(TIMESTAMP '1992-09-20')`	`29 years 1 month 27 days 12:39:00.844`

Function	Description	Example	Result
`century(`*`timestamp`*`)`	Extracts the century of a timestamp	`century(TIMESTAMP '1992-03-22')`	`20`
`date_diff(`*`part`*`,`*`startdate`*`,`*`enddate`*`)`	The number of partition boundaries between the timestamps	`date_diff('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`	`2`
`datediff(`*`part`*`,`*`startdate`*`,`*`enddate`*`)`	Alias of date_diff. The number of partition boundaries between the timestamps	`datediff('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`	`2`
`date_part(`*`part`*`,`*`timestamp`*`)`	Get subfield (equivalent to *extract*)	`date_part('minute', TIMESTAMP '1992-09-20 20:38:40')`	`38`
`datepart(`*`part`*`,`*`timestamp`*`)`	Alias of date_part. Get subfield (equivalent to *extract*)	`datepart('minute', TIMESTAMP '1992-09-20 20:38:40')`	`38`
`date_part([`*`part`*`, ...],`*`timestamp`*`)`	Get the listed subfields as a `struct`. The list must be constant.	`date_part(['year', 'month', 'day'], TIMESTAMP '1992-09-20 20:38:40')`	`{year: 1992, month: 9, day: 20}`
`datepart([`*`part`*`, ...],`*`timestamp`*`)`	Alias of date_part. Get the listed subfields as a `struct`. The list must be constant.	`datepart(['year', 'month', 'day'], TIMESTAMP '1992-09-20 20:38:40')`	`{year: 1992, month: 9, day: 20}`

Function	Description	Example	Result
`date_sub(`*`part`*`,` *`startdate`*`,` *`enddate`*`)`	The number of complete partitions between the timestamps	`date_sub('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`	`1`
`datesub(`*`part`*`,` *`startdate`*`,` *`enddate`*`)`	Alias of date_sub. The number of complete partitions between the timestamps	`datesub('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`	`1`
`date_trunc(` *`part`*`,` *`timestamp`*`)`	Truncate to specified precision	`date_trunc('hour', TIMESTAMP '1992-09-20 20:38:40')`	`1992-09-20 20:00:00`
`datetrunc(` *`part`*`,` *`timestamp`*`)`	Alias of date_trunc. Truncate to specified precision	`datetrunc('hour', TIMESTAMP '1992-09-20 20:38:40')`	`1992-09-20 20:00:00`
`dayname(` *`timestamp`*`)`	The (English) name of the weekday	`dayname(TIMESTAMP '1992-03-22')`	`Sunday`
`epoch(` *`timestamp`*`)`	Converts a timestamp to seconds since the epoch	`epoch('2022-11-07 08:43:04'::TIMESTAMP);`	`1667810584`
`epoch_ms(` *`timestamp`*`)`	Converts a timestamp to milliseconds since the epoch	`epoch_ms('2022-11-07 08:43:04.123456'::TIMESTAMP);`	`1667810584123`

Function	Description	Example	Result
`epoch_ms(ms)`	Converts ms since epoch to a timestamp	`epoch_ms(701222400000)`	`1992-03-22 00:00:00`
`epoch_ms(timestamp)`	Return the total number of milliseconds since the epoch	`epoch_ms(timestamp '2021-08-03 11:59:44.123456')`	`1627991984123`
`epoch_us(timestamp)`	Return the total number of microseconds since the epoch	`epoch_ms(timestamp '2021-08-03 11:59:44.123456')`	`1627991984123456`
`epoch_ns(timestamp)`	Return the total number of nanoseconds since the epoch	`epoch_ns(timestamp '2021-08-03 11:59:44.123456')`	`1627991984123456000`
`extract(field from timestamp)`	Get subfield from a timestamp	`extract('hour' FROM TIMESTAMP '1992-09-20 20:38:48')`	`20`
`greatest(timestamp, timestamp)`	The later of two timestamps	`greatest(TIMESTAMP '1992-09-20 20:38:48', TIMESTAMP '1992-03-22 01:02:03.1234')`	`1992-09-20 20:38:48`
`isfinite(timestamp)`	Returns true if the timestamp is finite, false otherwise	`isfinite(TIMESTAMP '1992-03-07')`	`true`
`isinf(timestamp)`	Returns true if the timestamp is infinite, false otherwise	`isinf(TIMESTAMP '-infinity')`	`true`

Function	Description	Example	Result
`last_day(`*`timestamp`*`)`	The last day of the month.	`last_day(TIMESTAMP '1992-03-22 01:02:03.1234')`	`1992-03-31`
`least(`*`timestamp`*`,`*`timestamp`*`)`	The earlier of two timestamps	`least(TIMESTAMP '1992-09-20 20:38:48', TIMESTAMP '1992-03-22 01:02:03.1234')`	`1992-03-22 01:02:03.1234`
`make_timestamp(`*`bigint`*`,`*`bigint`*`,`*`bigint`*`,`*`bigint`*`,`*`bigint`*`,`*`double`*`)`	The timestamp for the given parts	`make_timestamp(1992, 9, 20, 13, 34, 27.123456)`	`1992-09-20 13:34:27.123456`
`make_timestamp(`*`microseconds`*`)`	The timestamp for the given number of µs since the epoch	`make_timestamp(1667810584123456)`	`2022-11-07 08:43:04.123456`
`monthname(`*`timestamp`*`)`	The (English) name of the month.	`monthname(TIMESTAMP '1992-09-20')`	`September`
`strftime(`*`timestamp`*`,`*`format`*`)`	Converts timestamp to string according to the format string	`strftime(timestamp '1992-01-01 20:38:40', '%a, %-d %B %Y - %I:%M:%S %p')`	`Wed, 1 January 1992 - 08:38:40 PM`
`strptime(`*`text`*`,`*`format`*`)`	Converts string to timestamp according to the format string. Throws on failure.	`strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %-d %B %Y - %I:%M:%S %p')`	`1992-01-01 20:38:40`

Function	Description	Example	Result
`strptime(`*`text`*`,` *`format-list`*`)`	Converts string to timestamp applying the format strings in the list until one succeeds. Throws on failure.	`strptime('4/15/2023 10:56:00', ['%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S'])`	`2023-04-15 10:56:00`
`time_bucket(` *`bucket_width`*`,` *`times-tamp`*`[,`*`origin`*`])`	Truncate `timestamp` by the specified interval `bucket_width`. Buckets are aligned relative to `origin` timestamp. `origin` defaults to 2000-01-03 00:00:00 for buckets that don't include a month or year interval, and to 2000-01-01 00:00:00 for month and year buckets.	`time_bucket(INTERVAL '2 weeks', TIMESTAMP '1992-04-20 15:26:00', TIMESTAMP '1992-04-01 00:00:00')`	`1992-04-15 00:00:00`
`time_bucket(` *`bucket_width`*`,` *`times-tamp`*`[,`*`offset`*`])`	Truncate `timestamp` by the specified interval `bucket_width`. Buckets are offset by `offset` interval.	`time_bucket(INTERVAL '10 minutes', TIMESTAMP '1992-04-20 15:26:00-07', INTERVAL '5 minutes')`	`1992-04-20 15:25:00`
`to_timestamp(` *`double`*`)`	Converts seconds since the epoch to a timestamp with time zone	`to_timestamp(1284352323.5)`	`2010-09-13 04:32:03.5+00`

Function	Description	Example	Result
`try_strptime(text, format)`	Converts string to timestamp according to the format string. Returns NULL on failure.	`try_strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %-d %B %Y - %I:%M:%S %p')`	`1992-01-01 20:38:40`
`try_strptime(text, format-list)`	Converts string to timestamp applying the format strings in the list until one succeeds. Returns NULL on failure.	`try_strptime('4/15/2023 10:56:00', ['%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S'])`	`2023-04-15 10:56:00`

There are also dedicated extraction functions to get the subfields.

Functions applied to infinite dates will either return the same infinite dates (e.g, `greatest`) or `NULL` (e.g., `date_part`) depending on what "makes sense". In general, if the function needs to examine the parts of the infinite date, the result will be `NULL`.

Timestamp Table Functions

The table below shows the available table functions for `TIMESTAMP` types.

Function	Description	Example
`generate_series(timestamp, timestamp, interval)`	Generate a table of timestamps in the closed range, stepping by the interval	`generate_series(TIMESTAMP '2001-04-10', TIMESTAMP '2001-04-11', INTERVAL 30 MINUTE)`
`range(timestamp, timestamp, interval)`	Generate a table of timestamps in the half open range, stepping by the interval	`range(TIMESTAMP '2001-04-10', TIMESTAMP '2001-04-11', INTERVAL 30 MINUTE)`

Infinite values are not allowed as table function bounds.

Timestamp with Time Zone Functions

This section describes functions and operators for examining and manipulating `TIMESTAMP WITH TIME ZONE` values.

Despite the name, these values do not store a time zone - just an instant like `TIMESTAMP`. Instead, they request that the instant be binned and formatted using the current time zone.

Time zone support is not built in but can be provided by an extension, such as the ICU extension that ships with DuckDB.

In the examples below, the current time zone is presumed to be America/Los_Angeles using the Gregorian calendar.

Built-in Timestamp With Time Zone Functions

The table below shows the available scalar functions for `TIMESTAMP WITH TIME ZONE` values. Since these functions do not involve binning or display, they are always available.

Function	Description	Example	Result
`current_timestamp`	Current date and time (start of current transaction)	`current_timestamp`	`2022-10-08 12:44:46.122-07`
`get_current_timestamp()`	Current date and time (start of current transaction)	`get_current_timestamp()`	`2022-10-08 12:44:46.122-07`
`greatest(`*`timestamptz`*`,`*`timestamptz`*`)`	The later of two timestamps	`greatest(TIMESTAMPTZ '1992-09-20 20:38:48', TIMESTAMPTZ '1992-03-22 01:02:03.1234')`	`1992-09-20 20:38:48-07`
`isfinite(`*`timestamptz`*`)`	Returns true if the timestamp with time zone is finite, false otherwise	`isfinite(TIMESTAMPTZ '1992-03-07')`	`true`

Function	Description	Example	Result
`isinf(`*`timestamptz`*`)`	Returns true if the timestamp with time zone is infinite, false otherwise	`isinf(TIMESTAMPTZ '-infinity')`	`true`
`least(`*`timestamptz`*`,`*`timestamptz`*`)`	The earlier of two timestamps	`least(TIMESTAMPTZ '1992-09-20 20:38:48', TIMESTAMPTZ '1992-03-22 01:02:03.1234')`	`1992-03-22 01:02:03.1234-08`
`now()`	Current date and time (start of current transaction)	`now()`	`2022-10-08 12:44:46.122-07`
`transaction_timestamp()`	Current date and time (start of current transaction)	`transaction_timestamp()`	`2022-10-08 12:44:46.122-07`

Timestamp With Time Zone Strings

With no time zone extension loaded, `TIMESTAMPTZ` values will be cast to and from strings using offset notation. This will let you specify an instant correctly without access to time zone information. For portability, `TIMESTAMPTZ` values will always be displayed using GMT offsets:

```
SELECT '2022-10-08 13:13:34-07'::TIMESTAMPTZ;
-- 2022-10-08 20:13:34+00
```

If a time zone extension such as ICU is loaded, then a time zone can be parsed from a string and cast to a representation in the local time zone:

```
SELECT '2022-10-08 13:13:34 Europe/Amsterdam'::TIMESTAMPTZ::VARCHAR;
-- 2022-10-08 04:13:34-07 -- the offset will differ based on your local time
 ↪  zone
```

ICU Timestamp With Time Zone Operators

The table below shows the available mathematical operators for `TIMESTAMP WITH TIME ZONE` values provided by the ICU extension.

Operator	Description	Example	Result
`+`	addition of an `INTERVAL`	`TIMESTAMPTZ '1992-03-22 01:02:03' + INTERVAL 5 DAY`	`1992-03-27 01:02:03`
`-`	subtraction of `TIMESTAMPTZ`s	`TIMESTAMPTZ '1992-03-27' - TIMESTAMPTZ '1992-03-22'`	`5 days`
`-`	subtraction of an `INTERVAL`	`TIMESTAMPTZ '1992-03-27 01:02:03' - INTERVAL 5 DAY`	`1992-03-22 01:02:03`

Adding to or subtracting from infinite values produces the same infinite value.

ICU Timestamp With Time Zone Functions

The table below shows the ICU provided scalar functions for `TIMESTAMP WITH TIME ZONE` values.

Function	Description	Example	Result
`age(`*`timestamptz`*`,`*`timestamptz`*`)`	Subtract arguments, resulting in the time difference between the two timestamps	`age(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '1992-09-20')`	`8 years 6 months 20 days`
`age(`*`timestamptz`*`)`	Subtract from current_date	`age(TIMESTAMP '1992-09-20')`	`29 years 1 month 27 days 12:39:00.844`

Function	Description	Example	Result
`date_diff(`*`part`*`, `*`startdate`*`, `*`enddate`*`)`	The number of partition boundaries between the timestamps	`date_diff('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`	`2`
`datediff(`*`part`*`, `*`startdate`*`, `*`enddate`*`)`	Alias of date_diff. The number of partition boundaries between the timestamps	`datediff('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`	`2`
`date_part(`*`part`*`, `*`timestamptz`*`)`	Get subfield (equivalent to *extract*)	`date_part('minute', TIMESTAMPTZ '1992-09-20 20:38:40')`	`38`
`datepart(`*`part`*`, `*`timestamptz`*`)`	Alias of date_part. Get subfield (equivalent to *extract*)	`datepart('minute', TIMESTAMPTZ '1992-09-20 20:38:40')`	`38`
`date_part([`*`part`*`, ...],`*`timestamptz`*`)`	Get the listed subfields as a `struct`. The list must be constant.	`date_part(['year', 'month', 'day'], TIMESTAMPTZ '1992-09-20 20:38:40-07')`	`{year: 1992, month: 9, day: 20}`

Function	Description	Example	Result
`datepart([part, ...],timestamptz)`	Alias of date_part. Get the listed subfields as a `struct`. The list must be constant.	`datepart(['year', 'month', 'day'], TIMESTAMPTZ '1992-09-20 20:38:40-07')`	`{year: 1992, month: 9, day: 20}`
`date_sub(part, startdate, enddate)`	The number of complete partitions between the timestamps	`date_sub('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`	`1`
`datesub(part, startdate, enddate)`	Alias of date_sub. The number of complete partitions between the timestamps	`datesub('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`	`1`
`date_trunc(part, timestamptz)`	Truncate to specified precision	`date_trunc('hour', TIMESTAMPTZ '1992-09-20 20:38:40')`	`1992-09-20 20:00:00`
`datetrunc(part, timestamptz)`	Alias of date_trunc. Truncate to specified precision	`datetrunc('hour', TIMESTAMPTZ '1992-09-20 20:38:40')`	`1992-09-20 20:00:00`
`extract(field from timestamptz)`	Get subfield from a timestamp with time zone	`extract('hour' FROM TIMESTAMPTZ '1992-09-20 20:38:48')`	`20`

Function	Description	Example	Result
`epoch_ms(`*`timestamptz`*`)`	Converts a timestamptz to milliseconds since the epoch	`epoch_ms('2022-11-07 08:43:04.123456+00'::TIMESTAMPTZ);`	`1667810584123`
`epoch_us(`*`timestamptz`*`)`	Converts a timestamptz to microseconds since the epoch	`epoch_us('2022-11-07 08:43:04.123456+00'::TIMESTAMPTZ);`	`1667810584123456`
`epoch_ns(`*`timestamptz`*`)`	Converts a timestamptz to nanoseconds since the epoch	`epoch_ns('2022-11-07 08:43:04.123456+00'::TIMESTAMPTZ);`	`1667810584123456000`
`last_day(`*`timestamptz`*`)`	The last day of the month.	`last_day(TIMESTAMPTZ '1992-03-22 01:02:03.1234')`	`1992-03-31`
`make_timestamptz(`*`bigint`*`,`*`bigint`*`,`*`bigint`*`,`*`bigint`*`,`*`bigint`*`,`*`double`*`)`	The timestamp with time zone for the given parts in the current time zone	`make_timestamptz(1992, 9, 20, 13, 34, 27.123456)`	`1992-09-20 13:34:27.123456-07`
`make_timestamptz(`*`microseconds`*`)`	The timestamp with time zone for the given µs since the epoch	`make_timestamptz(1667810584123456)`	`2022-11-07 08:43:04.123456-08`
`make_timestamptz(`*`bigint`*`,`*`bigint`*`,`*`bigint`*`,`*`bigint`*`,`*`bigint`*`,`*`double`*`,`*`string`*`)`	The timestamp with time zone for the given parts and time zone	`make_timestamptz(1992, 9, 20, 15, 34, 27.123456, 'CET')`	`1992-09-20 06:34:27.123456-07`

Function	Description	Example	Result
`strftime(`*`timestamptz`*`,`*`format`*`)`	Converts timestamp with time zone to string according to the format string	`strftime(timestamptz '1992-01-01 20:38:40', '%a, %-d %B %Y - %I:%M:%S %p')`	`Wed, 1 January 1992 - 08:38:40 PM`
`strptime(`*`text`*`,`*`format`*`)`	Converts string to timestamp with time zone according to the format string if %Z is specified.	`strptime('Wed, 1 January 1992 - 08:38:40 PST', '%a, %-d %B %Y - %H:%M:%S %Z')`	`1992-01-01 08:38:40-08`
`time_bucket(`*`bucket_width`*`,`*`times-tamptz`*`[,`*`origin`*`])`	Truncate `timestamptz` by the specified interval `bucket_width`. Buckets are aligned relative to `origin` timestamptz. `origin` defaults to 2000-01-03 00:00:00+00 for buckets that don't include a month or year interval, and to 2000-01-01 00:00:00+00 for month and year buckets.	`time_bucket(INTERVAL '2 weeks', TIMESTAMPTZ '1992-04-20 15:26:00-07', TIMESTAMPTZ '1992-04-01 00:00:00-07')`	`1992-04-15 00:00:00-07`
`time_bucket(`*`bucket_width`*`,`*`times-tamptz`*`[,`*`offset`*`])`	Truncate `timestamptz` by the specified interval `bucket_width`. Buckets are offset by `offset` interval.	`time_bucket(INTERVAL '10 minutes', TIMESTAMPTZ '1992-04-20 15:26:00-07', INTERVAL '5 minutes')`	`1992-04-20 15:25:00-07`

Function	Description	Example	Result
`time_bucket(`*`bucket_width`*`,`*`times-tamptz`*`[,`*`timezone`*`])`	Truncate `timestamptz` by the specified interval *`bucket_width`*. Bucket starts and ends are calculated using `timezone`. `timezone` is a varchar and defaults to UTC.	`time_bucket(INTERVAL '2 days', TIMESTAMPTZ '1992-04-20 15:26:00-07', 'Europe/Berlin')`	`1992-04-19 15:00:00-07`

There are also dedicated extraction functions to get the subfields.

ICU Timestamp Table Functions

The table below shows the available table functions for `TIMESTAMP WITH TIME ZONE` types.

Function	Description	Example
`generate_series(`*`timestamptz`*`,`*`timestamptz`*`,`*`interval`*`)`	Generate a table of timestamps in the closed range (including both the starting timestamp and the ending timestamp), stepping by the interval	`generate_series(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '2001-04-11', INTERVAL 30 MINUTE)`
`range(`*`timestamptz`*`,`*`timestamptz`*`,`*`interval`*`)`	Generate a table of timestamps in the half open range (including the starting timestamp, but stopping before the ending timestamp) , stepping by the interval	`range(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '2001-04-11', INTERVAL 30 MINUTE)`

Infinite values are not allowed as table function bounds.

ICU Timestamp Without Time Zone Functions

The table below shows the ICU provided scalar functions that operate on plain `TIMESTAMP` values. These functions assume that the `TIMESTAMP` is a "local timestamp".

A local timestamp is effectively a way of encoding the part values from a time zone into a single value. They should be used with caution because the produced values can contain gaps and ambiguities thanks to daylight savings time. Often the same functionality can be implemented more reliably using the `struct` variant of the `date_part` function.

Function	Description	Example	Result
`current_localtime()`	Returns a `TIME` whose GMT bin values correspond to local time in the current time zone.	`current_localtime()`	`08:47:56.497`
`current_localtimestamp()`	Returns a `TIMESTAMP` whose GMT bin values correspond to local date and time in the current time zone.	`current_localtimestamp()`	`2022-12-17 08:47:56.497`
`localtime`	Synonym for the `current_localtime()` function call.	`localtime`	`2022-12-17 08:47:56.497`
`localtimestamp`	Synonym for the `current_localtimestamp()` function call.	`localtimestamp`	`2022-12-17 08:47:56.497`

Function	Description	Example	Result
`timezone(`*`text`*`,` *`timestamp`*`)`	Use the date parts of the timestamp in GMT to construct a timestamp in the given time zone. Effectively, the argument is a ”local” time.	`timezone('America/Denver', TIMESTAMP '2001-02-16 20:38:40')`	`2001-02-16 19:38:40-08`
`timezone(`*`text`*`,` *`timestamptz`*`)`	Use the date parts of the timestamp in the given time zone to construct a timestamp. Effectively, the result is a ”local” time.	`timezone('America/Denver', TIMESTAMPTZ '2001-02-16 20:38:40-05')`	`2001-02-16 18:38:40`

At Time Zone The `AT TIME ZONE` syntax is syntactic sugar for the (two argument) `timezone` function listed above:

```
timestamp '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';
-- 2001-02-16 19:38:40-08
timestamp with time zone '2001-02-16 20:38:40-05' AT TIME ZONE
 ↪  'America/Denver';
-- 2001-02-16 18:38:40
```

Infinities

Functions applied to infinite dates will either return the same infinite dates (e.g, `greatest`) or `NULL` (e.g., `date_part`) depending on what ”makes sense”. In general, if the function needs to examine the parts of the infinite temporal value, the result will be `NULL`.

Calendars

The ICU extension also supports non-Gregorian calendars. If such a calendar is current, then the display and binning operations will use that calendar.

Utility Functions

Utility Functions

The functions below are difficult to categorize into specific function types and are broadly useful.

Function	Description	Example	Result
`alias(`*`column`*`)`	Return the name of the column	`alias(column1)`	`'column1'`
`checkpoint(`*`database`*`)`	Synchronize WAL with file for (optional) database without interrupting transactions.	`checkpoint(my_db)`	success boolean
`coalesce(`*`expr`*`, ...)`	Return the first expression that evaluates to a non-NULL value. Accepts 1 or more parameters. Each expression can be a column, literal value, function result, or many others.	`coalesce(NULL, NULL, 'default_string')`	`'default_string'`
`error(`*`message`*`)`	Throws the given error *message*	`error('access_mode')`	
`ifnull(`*`expr`*`, `*`other`*`)`	A two-argument version of coalesce	`ifnull(NULL, 'default_string')`	`'default_string'`
`nullif(`*`a`*`, `*`b`*`)`	Return null if a = b, else return a. Equivalent to `CASE WHEN a=b THEN NULL ELSE a END`.	`nullif(1+1, 2)`	`NULL`

Function	Description	Example	Result
`current_schema()`	Return the name of the currently active schema. Default is main.	`current_schema()`	`'main'`
`current_schemas(`*`boolean`*`)`	Return list of schemas. Pass a parameter of `true` to include implicit schemas.	`current_schemas(true)`	`['temp', 'main', 'pg_catalog']`
`current_setting(`*`'setting_name'`*`)`	Return the current value of the configuration setting	`current_setting('access_mode')`	`'automatic'`
`currval(`*`'sequence_name'`*`)`	Return the current value of the sequence. Note that `nextval` must be called at least once prior to calling `currval`.	`currval('my_sequence_name')`	`1`
`force_checkpoint(`*`database`*`)`	Synchronize WAL with file for (optional) database interrupting transactions.	`force_checkpoint(my_db)`	success boolean
`gen_random_uuid()`	Alias of `uuid`. Return a random uuid similar to this: eeccb8c5-9943-b2bb-bb5e-222f4e14b687.	`gen_random_uuid()`	various
`hash(`*`value`*`)`	Returns an integer with the hash of the *value*	`hash('🦆')`	`2595805878642663834`

Function	Description	Example	Result
`icu_sort_key(`*`string`*`,`*`collator`*`)`	Surrogate key used to sort special characters according to the specific locale. Collator parameter is optional. Valid only when ICU extension is installed.	`icu_sort_key('ö', 'DE')`	`460145960106`
`md5(`*`string`*`)`	Return an md5 one-way hash of the *string*.	`md5('123')`	`'202cb962ac59075b964b07152`
`nextval(`*`'sequence_name'`*`)`	Return the following value of the sequence.	`nextval('my_sequence_name')`	`2`
`pg_typeof(`*`expression`*`)`	Returns the lower case name of the data type of the result of the expression. For PostgreSQL compatibility.	`pg_typeof('abc')`	`'varchar'`
`stats(`*`expression`*`)`	Returns a string with statistics about the expression. Expression can be a column, constant, or SQL expression.	`stats(5)`	`'[Min: 5, Max: 5][Has Null: false]'`
`txid_current()`	Returns the current transaction's ID (a `BIGINT`). It will assign a new one if the current transaction does not have one already.	`txid_current()`	various

Function	Description	Example	Result
`typeof(`*`expression`*`)`	Returns the name of the data type of the result of the expression.	`typeof('abc')`	`'VARCHAR'`
`uuid()`	Return a random uuid similar to this: eeccb8c5-9943-b2bb-bb5e-222f4e14b687.	`uuid()`	various
`version()`	Return the currently active version of DuckDB in this format: `v0.3.2`	`version()`	various

Utility Table Functions

A table function is used in place of a table in a `FROM` clause.

Function	Description	Example
`glob(`*`search_path`*`)`	Return filenames found at the location indicated by the *search_path* in a single column named `file`. The *search_path* may contain glob pattern matching syntax.	`glob('*')`

Aggregate Functions

Examples

```
-- produce a single row containing the sum of the "amount" column
SELECT SUM(amount) FROM sales;
-- produce one row per unique region, containing the sum of "amount" for
 ↪  each group
SELECT region, SUM(amount) FROM sales GROUP BY region;
```

```
-- return only the regions that have a sum of "amount" higher than 100
SELECT region FROM sales GROUP BY region HAVING SUM(amount) > 100;
-- return the number of unique values in the "region" column
SELECT COUNT(DISTINCT region) FROM sales;
-- return two values, the total sum of "amount" and the sum of "amount"
 ↪  minus columns where the region is "north"
SELECT SUM(amount), SUM(amount) FILTER (region != 'north') FROM sales;
-- returns a list of all regions in order of the "amount" column
SELECT LIST(region ORDER BY amount DESC) FROM sales;
```

Syntax

Aggregates are functions that *combine* multiple rows into a single value. Aggregates are different from scalar functions and window functions because they change the cardinality of the result. As such, aggregates can only be used in the `SELECT` and `HAVING` clauses of a SQL query.

When the `DISTINCT` clause is provided, only distinct values are considered in the computation of the aggregate. This is typically used in combination with the `COUNT` aggregate to get the number of distinct elements; but it can be used together with any aggregate function in the system.

When the `ORDER BY` clause is provided, the values being aggregated are sorted before applying the function. Usually this is not important, but there are some order-sensitive aggregates that can have indeterminate results (e.g., `first`, `last`, `list` and `string_agg`). These can be made deterministic by ordering the arguments. For order-insensitive aggregates, this clause is parsed and applied, which is inefficient, but still produces the same result.

General Aggregate Functions

The table below shows the available general aggregate functions.

Function	Description	Example	Alias(es)
`any_value(arg)`	Returns the first *non-null* value from arg.	`any_value(A)`	-
`arg_max(arg, val)`	Finds the row with the maximum `val`. Calculates the `arg` expression at that row.	`arg_max(A, B)`	`argMax(A, B)`, `max_by(A, b)`

Function	Description	Example	Alias(es)
`arg_min(arg, val)`	Finds the row with the minimum `val`. Calculates the `arg` expression at that row.	`arg_min(A, B)`	`argMin(A, B)`, `min_by(A, B)`
`avg(arg)`	Calculates the average value for all tuples in arg.	`avg(A)`	-
`bit_and(arg)`	Returns the bitwise AND of all bits in a given expression.	`bit_and(A)`	-
`bit_or(arg)`	Returns the bitwise OR of all bits in a given expression.	`bit_or(A)`	-
`bit_xor(arg)`	Returns the bitwise XOR of all bits in a given expression.	`bit_xor(A)`	-
`bitstring_agg(arg)`	Returns a bitstring with bits set for each distinct value.	`bitstring_agg(A)`	-
`bool_and(arg)`	Returns `true` if every input value is `true`, otherwise `false`.	`bool_and(A)`	-
`bool_or(arg)`	Returns `true` if any input value is `true`, otherwise `false`.	`bool_or(A)`	-
`count(arg)`	Calculates the number of tuples in arg.	`count(A)`	-
`favg(arg)`	Calculates the average using a more accurate floating point summation (Kahan Sum).	`favg(A)`	-
`first(arg)`	Returns the first value of a column.	`first(A)`	`arbitrary(A)`
`fsum(arg)`	Calculates the sum using a more accurate floating point summation (Kahan Sum).	`fsum(A)`	`sumKahan`, `kahan_sum`

Function	Description	Example	Alias(es)
`geomean(arg)`	Calculates the geometric mean for all tuples in arg.	`geomean(A)`	`geometric_mean(A)`
`histogram(arg)`	Returns a `LIST` of STRUCTs with the fields `bucket` and `count`.	`histogram(A)`	-
`last(arg)`	Returns the last value of a column.	`last(A)`	-
`list(arg)`	Returns a `LIST` containing all the values of a column.	`list(A)`	`array_agg`
`max(arg)`	Returns the maximum value present in arg.	`max(A)`	-
`min(arg)`	Returns the minimum value present in arg.	`min(A)`	-
`product(arg)`	Calculates the product of all tuples in arg	`product(A)`	-
`string_agg(arg, sep)`	Concatenates the column string values with a separator	`string_agg(S, ',')`	`group_concat`
`sum(arg)`	Calculates the sum value for all tuples in arg.	`sum(A)`	-

Approximate Aggregates

The table below shows the available approximate aggregate functions.

Function	Description	Example
`approx_count_distinct(x)`	Gives the approximate count of distinct elements using HyperLogLog.	`approx_count_distinct(A)`
`approx_quantile(x, pos)`	Gives the approximate quantile using T-Digest.	`approx_quantile(A, 0.5)`

Function	Description	Example
`reservoir_quantile(x, quantile, sample_size=8192)`	Gives the approximate quantile using reservoir sampling, the sample size is optional and uses 8192 as a default size.	`reservoir_quantile(A, 0.5, 1024)`

Statistical Aggregates

The table below shows the available statistical aggregate functions.

Function	Description	Formula	Alias
`corr(y, x)`	Returns the correlation coefficient for non-null pairs in a group.	`COVAR_POP(y, x) / (STDDEV_POP(x) * STDDEV_POP(y))`	-
`covar_pop(y, x)`	Returns the population covariance of input values.	`(SUM(x*y) - SUM(x) * SUM(y) / COUNT(*)) / COUNT(*)`	-
`covar_samp(y, x)`	Returns the sample covariance for non-null pairs in a group.	`(SUM(x*y) - SUM(x) * SUM(y) / COUNT(*)) / (COUNT(*) - 1)`	-
`entropy(x)`	Returns the log-2 entropy of count input-values.	-	-
`kurtosis(x)`	Returns the excess kurtosis (Fisher's definition) of all input values, with a bias correction according to the sample size.	-	-
`mad(x)`	Returns the median absolute deviation for the values within x. NULL values are ignored. Temporal types return a positive `INTERVAL`.	`MEDIAN(ABS(x-MEDIAN(x)))`	-

Function	Description	Formula	Alias
`median(x)`	Returns the middle value of the set. NULL values are ignored. For even value counts, quantitative values are averaged and ordinal values return the lower value.	`QUANTILE_CONT(x, 0.5)`	-
`mode(x)`	Returns the most frequent value for the values within x. NULL values are ignored.	-	-
`quantile_cont(x, pos)`	Returns the interpolated quantile number between 0 and 1. If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding interpolated quantiles.	-	-
`quantile_disc(x, pos)`	Returns the exact quantile number between 0 and 1. If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding exact quantiles.	-	`quantile`
`regr_avgx(y, x)`	Returns the average of the independent variable for non-null pairs in a group, where x is the independent variable and y is the dependent variable.	-	-
`regr_avgy(y, x)`	Returns the average of the dependent variable for non-null pairs in a group, where x is the independent variable and y is the dependent variable.	-	-

Function	Description	Formula	Alias
`regr_count(y, x)`	Returns the number of non-null number pairs in a group.	`(SUM(x*y) - SUM(x) * SUM(y) / COUNT(*)) / COUNT(*)`	-
`regr_intercept(y, x)`	Returns the intercept of the univariate linear regression line for non-null pairs in a group.	`AVG(y)-REGR_SLOPE(y, x)*AVG(x)`	-
`regr_r2(y, x)`	Returns the coefficient of determination for non-null pairs in a group.	-	-
`regr_slope(y, x)`	Returns the slope of the linear regression line for non-null pairs in a group.	`COVAR_POP(x, y) / VAR_POP(x)`	-
`regr_sxx(y, x)`	-	`REGR_COUNT(y, x) * VAR_POP(x)`	-
`regr_sxy(y, x)`	Returns the population covariance of input values.	`REGR_COUNT(y, x) * COVAR_POP(y, x)`	-
`regr_syy(y, x)`	-	`REGR_COUNT(y, x) * VAR_POP(y) f`	-
`skewness(x)`	Returns the skewness of all input values.	-	-
`stddev_pop(x)`	Returns the population standard deviation.	`sqrt(var_pop(x))`	-
`stddev_samp(x)`	Returns the sample standard deviation.	`sqrt(var_samp(x))`	`stddev(x)`
`var_pop(x)`	Returns the population variance.	-	-

Function	Description	Formula	Alias
`var_samp(x)`	Returns the sample variance of all input values.	`(SUM(x^2) - SUM(x)^2 / COUNT(x)) / (COUNT(x) - 1)`	`variance(arg, val)`

Ordered Set Aggregate Functions

The table below shows the available "ordered set" aggregate functions. These functions are specified using the `WITHIN GROUP(ORDER BY sort_expression)` syntax, and they are converted to an equivalent aggregate function that takes the ordering expression as the first argument.

Function	Equivalent
`mode() WITHIN GROUP (ORDER BY sort_expression)`	`mode(sort_expression)`
`percentile_cont(fraction) WITHIN GROUP (ORDER BY sort_expression)`	`quantile_cont(sort_expression, fraction)`
`percentile_cont(fractions) WITHIN GROUP (ORDER BY sort_expression)`	`quantile_cont(sort_expression, fractions)`
`percentile_disc(fraction) WITHIN GROUP (ORDER BY sort_expression)`	`quantile_disc(sort_expression, fraction)`
`percentile_disc(fractions) WITHIN GROUP (ORDER BY sort_expression)`	`quantile_disc(sort_expression, fractions)`

Configuration

DuckDB has a number of configuration options that can be used to change the behavior of the system. The configuration options can be set using either the `SET` statement or the `PRAGMA` statement. They can also be reset to their original values using the `RESET` statement.

Examples

```sql
-- set the memory limit of the system to 10GB
SET memory_limit='10GB';
-- configure the system to use 1 thread
SET threads TO 1;
-- enable printing of a progress bar during long-running queries
SET enable_progress_bar=true;
-- set the default null order to NULLS LAST
PRAGMA default_null_order='nulls_last';

-- show a list of all available settings
SELECT * FROM duckdb_settings();

-- return the current value of a specific setting
-- this example returns 'automatic'
SELECT current_setting('access_mode');

-- reset the memory limit of the system back to the default
RESET memory_limit;
```

Configuration Reference

Below is a list of all available settings.

Name	Description	Input type	Default value
`Calendar`	The current calendar	`VARCHAR`	System (locale) calendar
`TimeZone`	The current time zone	`VARCHAR`	System (locale) timezone
`access_mode`	Access mode of the database (**AUTOMATIC**, **READ_ONLY** or **READ_WRITE**)	`VARCHAR`	`automatic`
`allocator_flush_threshold`	Peak allocation threshold at which to flush the allocator after completing a task.	`VARCHAR`	`134.2MB`
`allow_unsigned_extensions`	Allow to load extensions with invalid or missing signatures	`BOOLEAN`	`false`

Name	Description	Input type	Default value
`arrow_large_buffer_size`	If arrow buffers for strings, blobs, uuids and bits should be exported using large buffers	`BOOLEAN`	`false`
`autoinstall_extension_repository`	Overrides the custom endpoint for extension installation on autoloading	`VARCHAR`	
`autoinstall_known_extensions`	Whether known extensions are allowed to be automatically installed when a query depends on them	`BOOLEAN`	`true`
`autoload_known_extensions`	Whether known extensions are allowed to be automatically loaded when a query depends on them	`BOOLEAN`	`true`
`binary_as_string`	In Parquet files, interpret binary data as a string.	`BOOLEAN`	
`checkpoint_threshold`, `wal_autocheckpoint`	The WAL size threshold at which to automatically trigger a checkpoint (e.g., 1GB)	`VARCHAR`	`16.7MB`
`custom_extension_repository`	Overrides the custom endpoint for remote extension installation	`VARCHAR`	
`custom_user_agent`	Metadata from DuckDB callers	`VARCHAR`	
`default_collation`	The collation setting used when none is specified	`VARCHAR`	
`default_null_order`, `null_order`	Null ordering used when none is specified (**`NULLS_FIRST`** or **`NULLS_LAST`**)	`VARCHAR`	`NULLS_LAST`
`default_order`	The order type used when none is specified (**`ASC`** or **`DESC`**)	`VARCHAR`	`ASC`
`disabled_filesystems`	Disable specific file systems preventing access (e.g., LocalFileSystem)	`VARCHAR`	

Name	Description	Input type	Default value
duckdb_api	DuckDB API surface	VARCHAR	duckdb/v0.9.2-dev385(linux_amd64_gcc4)
enable_external_access	Allow the database to access external state (through e.g., loading/installing modules, COPY TO/FROM, CSV readers, pandas replacement scans, etc)	BOOLEAN	true
enable_fsst_vectors	Allow scans on FSST compressed segments to emit compressed vectors to utilize late decompression	BOOLEAN	false
enable_http_metadata_cache	Whether or not the global http metadata is used to cache HTTP metadata	BOOLEAN	false
enable_object_cache	Whether or not object cache is used to cache e.g., Parquet metadata	BOOLEAN	false
enable_profiling	Enables profiling, and sets the output format (**JSON**, **QUERY_TREE**, **QUERY_TREE_OPTIMIZER**)	VARCHAR	NULL
enable_progress_bar_print	Controls the printing of the progress bar, when 'enable_progress_bar' is true	BOOLEAN	true
enable_progress_bar	Enables the progress bar, printing progress to the terminal for long queries	BOOLEAN	false
explain_output	Output of EXPLAIN statements (**ALL**, **OPTIMIZED_ONLY**, **PHYSICAL_ONLY**)	VARCHAR	physical_only
extension_directory	Set the directory to store extensions in	VARCHAR	
external_threads	The number of external threads that work on DuckDB tasks.	BIGINT	0

Name	Description	Input type	Default value
file_search_path	A comma separated list of directories to search for input files	VARCHAR	
force_download	Forces upfront download of file	BOOLEAN	false
home_directory	Sets the home directory used by the system	VARCHAR	
http_retries	HTTP retries on I/O error (default 3)	UBIGINT	3
http_retry_backoff	Backoff factor for exponentially increasing retry wait time (default 4)	FLOAT	4
http_retry_wait_ms	Time between retries (default 100ms)	UBIGINT	100
http_timeout	HTTP timeout read/write/connection/retry (default 30000ms)	UBIGINT	30000
immediate_transaction_mode	Whether transactions should be started lazily when needed, or immediately when BEGIN TRANSACTION is called	BOOLEAN	false
integer_division	Whether or not the / operator defaults to integer division, or to floating point division	BOOLEAN	false
lock_configuration	Whether or not the configuration can be altered	BOOLEAN	false
log_query_path	Specifies the path to which queries should be logged (default: empty string, queries are not logged)	VARCHAR	NULL
max_expression_depth	The maximum expression depth limit in the parser. WARNING: increasing this setting and using very deep expressions might lead to stack overflow errors.	UBIGINT	1000
max_memory, memory_limit	The maximum memory of the system (e.g., 1GB)	VARCHAR	80% of RAM

Name	Description	Input type	Default value
`ordered_aggregate_threshold`	The number of rows to accumulate before sorting, used for tuning	UBIGINT	262144
`password`	The password to use. Ignored for legacy compatibility.	VARCHAR	NULL
`perfect_ht_threshold`	Threshold in bytes for when to use a perfect hash table (default: 12)	BIGINT	12
`pivot_filter_threshold`	The threshold to switch from using filtered aggregates to LIST with a dedicated pivot operator	BIGINT	10
`pivot_limit`	The maximum number of pivot columns in a pivot statement (default: 100000)	BIGINT	100000
`prefer_range_joins`	Force use of range joins with mixed predicates	BOOLEAN	false
`preserve_identifier_case`	Whether or not to preserve the identifier case, instead of always lowercasing all non-quoted identifiers	BOOLEAN	true
`preserve_insertion_order`	Whether or not to preserve insertion order. If set to false the system is allowed to re-order any results that do not contain ORDER BY clauses.	BOOLEAN	true
`profile_output`, `profiling_output`	The file to which profile output should be saved, or empty to print to the terminal	VARCHAR	
`profiler_history_size`	Sets the profiler history size	BIGINT	NULL
`profiling_mode`	The profiling mode (**STANDARD** or **DETAILED**)	VARCHAR	NULL
`progress_bar_time`	Sets the time (in milliseconds) how long a query needs to take before we start printing a progress bar	BIGINT	2000

Name	Description	Input type	Default value
s3_access_key_id	S3 Access Key ID	VARCHAR	
s3_endpoint	S3 Endpoint (default 's3.amazonaws.com')	VARCHAR	s3.amazonaws.com
s3_region	S3 Region	VARCHAR	
s3_secret_access_key	S3 Access Key	VARCHAR	
s3_session_token	S3 Session Token	VARCHAR	
s3_uploader_max_filesize	S3 Uploader max filesize (between 50GB and 5TB, default 800GB)	VARCHAR	800GB
s3_uploader_max_parts_per_file	S3 Uploader max parts per file (between 1 and 10000, default 10000)	UBIGINT	10000
s3_uploader_thread_limit	S3 Uploader global thread limit (default 50)	UBIGINT	50
s3_url_compatibility_mode	Disable Globs and Query Parameters on S3 urls	BOOLEAN	0
s3_url_style	S3 url style ('vhost' (default) or 'path')	VARCHAR	vhost
s3_use_ssl	S3 use SSL (default true)	BOOLEAN	1
schema	Sets the default search schema. Equivalent to setting search_path to a single value.	VARCHAR	main
search_path	Sets the default catalog search path as a comma-separated list of values	VARCHAR	

Name	Description	Input type	Default value
`temp_directory`	Set the directory to which to write temp files	VARCHAR	
`threads`, `worker_threads`	The number of total threads used by the system.	BIGINT	# Cores
`username`, `user`	The username to use. Ignored for legacy compatibility.	VARCHAR	NULL

Constraints

In SQL, constraints can be specified for tables. Constraints enforce certain properties over data that is inserted into a table. Constraints can be specified along with the schema of the table as part of the CREATE TABLE statement. In certain cases, constraints can also be added to a table using the ALTER TABLE statement, but this is not currently supported for all constraints.

Syntax

Check

Check constraints allow you to specify an arbitrary boolean expression. Any columns that *do not* satisfy this expression violate the constraint. For example, we could enforce that the name column does not contain spaces using the following CHECK constraint.

```
CREATE TABLE students(name VARCHAR CHECK(NOT CONTAINS(name, ' ')));
INSERT INTO students VALUES ('this name contains spaces');
-- Constraint Error: CHECK constraint failed: students
```

Not Null

A not-null constraint specifies that the column cannot contain any NULL values. By default, all columns in tables are nullable. Adding NOT NULL to a column definition enforces that a column cannot contain NULL values.

```
CREATE TABLE students(name VARCHAR NOT NULL);
INSERT INTO students VALUES (NULL);
-- Constraint Error: NOT NULL constraint failed: students.name
```

Primary Key/Unique

Primary key or unique constraints define a column, or set of columns, that are a unique identifier for a row in the table. The constraint enforces that the specified columns are *unique* within a table, i.e., that at most one row contains the given values for the set of columns.

```
CREATE TABLE students(id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO students VALUES (1, 'Student 1');
INSERT INTO students VALUES (1, 'Student 2');
-- Constraint Error: Duplicate key "id: 1" violates primary key constraint
```

In order to enforce this property efficiently, an ART index is automatically created for every primary key or unique constraint that is defined in the table.

Primary key constraints and unique constraints are identical except for two points:

- A table can only have one primary key constraint defined, but many unique constraints
- A primary key constraint also enforces the keys to not be `NULL`.

> **Note.** Indexes have certain limitations that might result in constraints being evaluated too eagerly, see the indexes section for more details

Foreign Key

Foreign keys define a column, or set of columns, that refer to a primary key or unique constraint from *another* table. The constraint enforces that the key exists in the other table.

```
CREATE TABLE students(id INTEGER PRIMARY KEY, name VARCHAR);
CREATE TABLE exams(exam_id INTEGER REFERENCES students(id), grade INTEGER);
INSERT INTO students VALUES (1, 'Student 1');
INSERT INTO exams VALUES (1, 10);
INSERT INTO exams VALUES (2, 10);
-- Constraint Error: Violates foreign key constraint because key "id: 2"
 ↪  does not exist in the referenced table
```

In order to enforce this property efficiently, an ART index is automatically created for every foreign key constraint that is defined in the table.

> **Note.** Indexes have certain limitations that might result in constraints being evaluated too eagerly, see the indexes section for more details

Indexes

Index Types

DuckDB currently uses two index types:

- A min-max index (also known as zonemap and block range index) is automatically created for columns of all general-purpose data types.
- An Adaptive Radix Tree (ART) is mainly used to ensure primary key constraints and to speed up point and very highly selective (i.e., < 0.1%) queries. Such an index is automatically created for columns with a `UNIQUE` or `PRIMARY KEY` constraint and can be defined using `CREATE INDEX`.

Joins on columns with an ART index can make use of the index join algorithm. Index joins are disabled by default, forcing them is possible by issuing the following PRAGMA:

```
PRAGMA force_index_join;
```

> **Note.** ART indexes must currently be able to fit in-memory. Avoid creating ART indexes if the index does not fit in memory.

Persistence

Both min-max indexes and ART indexes are persisted on disk.

Create Index

`CREATE INDEX` constructs an index on the specified column(s) of the specified table. Compound indexes on multiple columns/expressions are supported.

> **Note.** Unidimensional indexes are supported, while multidimensional indexes are not yet supported.

Parameters

Name	Description
`UNIQUE`	Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data that would result in duplicate entries will generate an error.
`name`	The name of the index to be created.
`table`	The name of the table to be indexed.
`column`	The name of the column to be indexed.
`expression`	An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses can be omitted if the expression has the form of a function call.

Examples

```
-- Create a unique index 'films_id_idx' on the column id of table films.
CREATE UNIQUE INDEX films_id_idx ON films (id);
-- Create index 's_idx' that allows for duplicate values on column revenue
 ↪  of table films.
CREATE INDEX s_idx ON films (revenue);
-- Create compound index 'gy_idx' on genre and year columns.
CREATE INDEX gy_idx ON films (genre, year);
-- Create index 'i_index' on the expression of the sum of columns j and k
 ↪  from table integers.
CREATE INDEX i_index ON integers ((j+k));
```

Drop Index

`DROP INDEX` drops an existing index from the database system.

Parameters

Name	Description
`IF EXISTS`	Do not throw an error if the index does not exist.

Name	Description
name	The name of an index to remove.

Examples

```
-- Remove the index title_idx.
DROP INDEX title_idx;
```

Index Limitations

ART indexes create a secondary copy of the data in a second location - this complicates processing, particularly when combined with transactions. Certain limitations apply when it comes to modifying data that is also stored in secondary indexes.

Updates Become Deletes and Inserts When an update statement is executed on a column that is present in an index - the statement is transformed into a *delete* of the original row followed by an *insert*. This has certain performance implications, particularly for wide tables, as entire rows are rewritten instead of only the affected columns.

Over-Eager Unique Constraint Checking Due to the presence of transactions, data can only be removed from the index after (1) the transaction that performed the delete is committed, and (2) no further transactions exist that refer to the old entry still present in the index. As a result of this - transactions that perform *deletions followed by insertions* may trigger unexpected unique constraint violations, as the deleted tuple has not actually been removed from the index yet. For example:

```
CREATE TABLE students(id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO students VALUES (1, 'Student 1');
BEGIN;
DELETE FROM students WHERE id=1;
INSERT INTO students VALUES (1, 'Student 2');
-- Constraint Error: Duplicate key "id: 1" violates primary key constraint
```

This, combined with the fact that updates are turned into deletions and insertions within the same transaction, means that updating rows in the presence of unique or primary key constraints can often lead to unexpected unique constraint violations.

```
CREATE TABLE students(id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO students VALUES (1, 'Student 1');
```

```
UPDATE students SET name='Student 2', id=1 WHERE id=1;
-- Constraint Error: Duplicate key "id: 1" violates primary key constraint
```

Currently this is an expected limitation of the system - although we aim to resolve this in the future.

Information Schema

The views in the `information_schema` are SQL-standard views that describe the catalog entries of the database. These views can be filtered to obtain information about a specific column or table.

Database, Catalog and Schema

The top level catalog view is `information_schema.schemata`. It lists the catalogs and the schemas present in the database and has the following layout:

Column	Description	Type	Example
`catalog_name`	Name of the database that the schema is contained in.	`VARCHAR`	`NULL`
`schema_name`	Name of the schema.	`VARCHAR`	`'main'`
`schema_owner`	Name of the owner of the schema. Not yet implemented.	`VARCHAR`	`NULL`
`default_character_set_catalog`	Applies to a feature not available in DuckDB.	`VARCHAR`	`NULL`
`default_character_set_schema`	Applies to a feature not available in DuckDB.	`VARCHAR`	`NULL`
`default_character_set_name`	Applies to a feature not available in DuckDB.	`VARCHAR`	`NULL`
`sql_path`	The file system location of the database. Currently unimplemented.	`VARCHAR`	`NULL`

Tables and Views

The view that describes the catalog information for tables and views is `information_schema.tables`. It lists the tables present in the database and has the following layout:

Column	Description	Type	Example
`table_catalog`	The catalog the table or view belongs to.	`VARCHAR`	`NULL`
`table_schema`	The schema the table or view belongs to.	`VARCHAR`	`'main'`
`table_name`	The name of the table or view.	`VARCHAR`	`'widgets'`
`table_type`	The type of table. One of: `BASE TABLE`, `LOCAL TEMPORARY`, `VIEW`.	`VARCHAR`	`'BASE TABLE'`
`self_referencing_column_name`	Applies to a feature not available in DuckDB.	`VARCHAR`	`NULL`
`reference_generation`	Applies to a feature not available in DuckDB.	`VARCHAR`	`NULL`
`user_defined_type_catalog`	If the table is a typed table, the name of the database that contains the underlying data type (always the current database), else null. Currently unimplemented.	`VARCHAR`	`NULL`
`user_defined_type_schema`	If the table is a typed table, the name of the schema that contains the underlying data type, else null. Currently unimplemented.	`VARCHAR`	`NULL`
`user_defined_type_name`	If the table is a typed table, the name of the underlying data type, else null. Currently unimplemented.	`VARCHAR`	`NULL`

Column	Description	Type	Example
`is_insertable_into`	YES if the table is insertable into, NO if not (Base tables are always insertable into, views not necessarily.)	`VARCHAR`	`'YES'`
`is_typed`	YES if the table is a typed table, NO if not.	`VARCHAR`	`'NO'`
`commit_action`	Not yet implemented.	`VARCHAR`	`'NO'`

Columns

The view that describes the catalog information for columns is `information_schema.columns`. It lists the column present in the database and has the following layout:

Column	Description	Type	Example
`table_catalog`	Name of the database containing the table.	`VARCHAR`	`NULL`
`table_schema`	Name of the schema containing the table.	`VARCHAR`	`'main'`
`table_name`	Name of the table.	`VARCHAR`	`'widgets'`
`column_name`	Name of the column.	`VARCHAR`	`'price'`
`ordinal_position`	Ordinal position of the column within the table (count starts at 1).	`INTEGER`	`5`
`column_default`	Default expression of the column.	`VARCHAR`	`1.99`
`is_nullable`	YES if the column is possibly nullable, NO if it is known not nullable.	`VARCHAR`	`'YES'`
`data_type`	Data type of the column.	`VARCHAR`	`'DECIMAL(18, 2)'`

Column	Description	Type	Example
`character_maximum_length`	If `data_type` identifies a character or bit string type, the declared maximum length; null for all other data types or if no maximum length was declared.	`INTEGER`	`255`
`character_octet_length`	If data_type identifies a character type, the maximum possible length in octets (bytes) of a datum; null for all other data types. The maximum octet length depends on the declared character maximum length (see above) and the character encoding.	`INTEGER`	`1073741824`
`numeric_precision`	If data_type identifies a numeric type, this column contains the (declared or implicit) precision of the type for this column. The precision indicates the number of significant digits. For all other data types, this column is null.	`INTEGER`	`18`
`numeric_scale`	If data_type identifies a numeric type, this column contains the (declared or implicit) scale of the type for this column. The precision indicates the number of significant digits. For all other data types, this column is null.	`INTEGER`	`2`

Column	Description	Type	Example
`datetime_precision`	If data_type identifies a date, time, timestamp, or interval type, this column contains the (declared or implicit) fractional seconds precision of the type for this column, that is, the number of decimal digits maintained following the decimal point in the seconds value. No fractional seconds are currently supported in DuckDB. For all other data types, this column is null.	`INTEGER`	`0`

Catalog Functions

Several functions are also provided to see details about the schemas that are configured in the database.

Function	Description	Example	Result
`current_schema()`	Return the name of the currently active schema. Default is main.	`current_schema()`	`'main'`
`current_schemas(boolean)`	Return list of schemas. Pass a parameter of `true` to include implicit schemas.	`current_schemas(true)`	`['temp', 'main', 'pg_catalog']`

DuckDB_% Metadata Functions

DuckDB offers a collection of table functions that provide metadata about the current database. These functions reside in the `main` schema and their names are prefixed with `duckdb_`.

The resultset returned by a `duckdb_` table function may be used just like an ordinary table or view. For example, you can use a `duckdb_` function call in the `FROM` clause of a `SELECT` statement, and

you may refer to the columns of its returned resultset elsewhere in the statement, for example in the WHERE clause.

Table functions are still functions, and you should write parenthesis after the function name to call it to obtain its returned resultset:

```
SELECT * FROM duckdb_settings();
```

Alternatively, you may execute table functions also using the CALL-syntax:

```
CALL duckdb_settings();
```

In this case too, the parentheses are mandatory.

> **Note.** For some of the duckdb_% functions, there is also an identically named view available, which also resides in the main schema. Typically, these views do a SELECT on the duckdb_ table function with the same name, while filtering out those objects that are marked as internal. We mention it here, because if you accidentally omit the parentheses in your duckdb_ table function call, you might still get a result, but from the identically named view.

Example:

```
-- duckdb_views table function: returns all views, including those marked
 ↪  internal
SELECT * FROM duckdb_views();
-- duckdb_views view: returns views that are not marked as internal
SELECT * FROM duckdb_views;
```

duckdb_columns

The duckdb_columns() function provides metadata about the columns available in the DuckDB instance.

Column	Description	Type
database_name	The name of the database that contains the column object.	VARCHAR
database_oid	Internal identifier of the database that contains the column object.	BIGINT
schema_name	The SQL name of the schema that contains the table object that defines this column.	VARCHAR

Column	Description	Type
`schema_oid`	Internal identifier of the schema object that contains the table of the column.	`BIGINT`
`table_name`	The SQL name of the table that defines the column.	`VARCHAR`
`table_oid`	Internal identifier (name) of the table object that defines the column.	`BIGINT`
`column_name`	The SQL name of the column.	`VARCHAR`
`column_index`	The unique position of the column within its table.	`INTEGER`
`internal`	`true` if this column built-in, `false` if it is user-defined.	`BOOLEAN`
`column_default`	The default value of the column (expressed in SQL)	`VARCHAR`
`is_nullable`	`true` if the column can hold `NULL` values; `false` if the column cannot hold `NULL`-values.	`BOOLEAN`
`data_type`	The name of the column datatype.	`VARCHAR`
`data_type_id`	The internal identifier of the column data type	`BIGINT`
`character_maximum_length`	Always `NULL`. DuckDB text types do not enforce a value length restriction based on a length type parameter.	`INTEGER`
`numeric_precision`	The number of units (in the base indicated by `numeric_precision_radix`) used for storing column values. For integral and approximate numeric types, this is the number of bits. For decimal types, this is the number of digits positions.	`INTEGER`

Column	Description	Type
`numeric_precision_radix`	The number-base of the units in the `numeric_precision` column. For integral and approximate numeric types, this is `2`, indicating the precision is expressed as a number of bits. For the `decimal` type this is `10`, indicating the precision is expressed as a number of decimal positions.	`INTEGER`
`numeric_scale`	Applicable to `decimal` type. Indicates the maximum number of fractional digits (i.e., the number of digits that may appear after the decimal separator).	`INTEGER`

The `information_schema.columns` system view provides a more standardized way to obtain metadata about database columns, but the `duckdb_columns` function also returns metadata about DuckDB internal objects. (In fact, `information_schema.columns` is implemented as a query on top of `duckdb_columns()`)

`duckdb_constraints`

The `duckdb_constraints()` function provides metadata about the constraints available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains the constraint.	`VARCHAR`
`database_oid`	Internal identifier of the database that contains the constraint.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the table on which the constraint is defined.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object that contains the table on which the constraint is defined.	`BIGINT`

Column	Description	Type
`table_name`	The SQL name of the table on which the constraint is defined.	`VARCHAR`
`table_oid`	Internal identifier (name) of the table object on which the constraint is defined.	`BIGINT`
`constraint_index`	Indicates the position of the constraint as it appears in its table definition.	`BIGINT`
`constraint_type`	Indicates the type of constraint. Applicable values are `CHECK`, `FOREIGN KEY`, `PRIMARY KEY`, `NOT NULL`, `UNIQUE`.	`VARCHAR`
`constraint_text`	The definition of the constraint expressed as a SQL-phrase. (Not necessarily a complete or syntactically valid DDL-statement.)	`VARCHAR`
`expression`	If constraint is a check constraint, the definition of the condition being checked, otherwise `NULL`.	`VARCHAR`
`constraint_column_indexes`	An array of table column indexes referring to the columns that appear in the constraint definition	`BIGINT[]`
`constraint_column_names`	An array of table column names appearing in the constraint definition	`VARCHAR[]`

`duckdb_databases`

The `duckdb_databases()` function lists the databases that are accessible from within the current DuckDB process. Apart from the database associated at startup, the list also includes databases that were attached later on to the duckdb process

Column	Description	Type
`database_name`	The name of the database, or the alias if the database was attached using an ALIAS-clause.	`VARCHAR`
`database_oid`	The internal identifier of the database.	`VARCHAR`

Column	Description	Type
`path`	The file path associated with the database.	VARCHAR
`internal`	`true` indicates a system or built-in database. False indicates a user-defined database.	BOOLEAN
`type`	The type indicates the type of RDBMS implemented by the attached database. For DuckDB databases, that value is `duckdb`.	

`duckdb_dependencies`

The `duckdb_dependencies()` function provides metadata about the dependencies available in the DuckDB instance.

Column	Description	Type
`classid`	Always 0	`BIGINT`
`objid`	The internal id of the object.	`BIGINT`
`objsubid`	Always 0	`INTEGER`
`refclassid`	Always 0	`BIGINT`
`refobjid`	The internal id of the dependent object.	`BIGINT`
`refobjsubid`	Always 0	`INTEGER`
`deptype`	The type of dependency. Either regular (n) or automatic (a).	VARCHAR

`duckdb_extensions`

The `duckdb_extensions()` function provides metadata about the extensions available in the DuckDB instance.

Column	Description	Type
`extension_name`	The name of the extension.	VARCHAR

Column	Description	Type
`loaded`	`true` if the extension is loaded, `false` if it's not loaded.	`BOOLEAN`
`installed`	`true` if the extension is installed, `false` if it's not installed.	`BOOLEAN`
`install_path`	`(BUILT-IN)` if the extension is built-in, otherwise, the filesystem path where binary that implements the extension resides.	`VARCHAR`
`description`	Human readable text that describes the extension's functionality.	`VARCHAR`
`aliases`	List of alternative names for this extension.	`VARCHAR[]`

`duckdb_functions`

The `duckdb_functions()` function provides metadata about the functions available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this function.	`VARCHAR`
`schema_name`	The SQL name of the schema where the function resides.	`VARCHAR`
`function_name`	The SQL name of the function.	`VARCHAR`
`function_type`	The function kind. Value is one of: `table`,`scalar`,`aggregate`,`pragma`,`macro`	`VARCHAR`
`description`	Description of this function (always `NULL`)	`VARCHAR`
`return_type`	The logical data type name of the returned value. Applicable for scalar and aggregate functions.	`VARCHAR`
`parameters`	If the function has parameters, the list of parameter names.	`VARCHAR[]`

Column	Description	Type
`parameter_types`	If the function has parameters, a list of logical data type names corresponding to the parameter list.	`VARCHAR[]`
`varargs`	The name of the data type in case the function has a variable number of arguments, or `NULL` if the function does not have a variable number of arguments.	`VARCHAR`
`macro_definition`	If this is a macro, the SQL expression that defines it.	`VARCHAR`
`has_side_effects`	`false` if this is a pure function. `true` if this function changes the database state (like sequence functions `nextval()` and `curval()`).	`BOOLEAN`
`function_oid`	The internal identifier for this function	`BIGINT`

`duckdb_indexes`

The `duckdb_indexes()` function provides metadata about secondary indexes available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this index.	`VARCHAR`
`database_oid`	Internal identifier of the database containing the index.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the table with the secondary index.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object.	`BIGINT`
`index_name`	The SQL name of this secondary index	`VARCHAR`
`index_oid`	The object identifier of this index.	`BIGINT`
`table_name`	The name of the table with the index	`VARCHAR`

Column	Description	Type
`table_oid`	Internal identifier (name) of the table object.	`BIGINT`
`is_unique`	`true` if the index was created with the `UNIQUE` modifier, `false` if it was not.	`BOOLEAN`
`is_primary`	Always `false`	`BOOLEAN`
`expressions`	Always `NULL`	`VARCHAR`
`sql`	The definition of the index, expressed as a `CREATE INDEX` SQL statement.	`VARCHAR`

Note that `duckdb_indexes` only provides metadata about secondary indexes - i.e., those indexes created by explicit `CREATE INDEX` statements. Primary keys are maintained using indexes, but their details are included in the `duckdb_constraints()` function.

`duckdb_keywords`

The `duckdb_keywords()` function provides metadata about DuckDB's keywords and reserved words.

Column	Description	Type
`keyword_name`	The keyword.	`VARCHAR`
`keyword_category`	Indicates the category of the keyword. Values are `column_name`, `reserved`, `type_function` and `unreserved`.	`VARCHAR`

`duckdb_schemas`

The `duckdb_schemas()` function provides metadata about the schemas available in the DuckDB instance.

Column	Description	Type
`oid`	Internal identifier of the schema object.	`BIGINT`

Column	Description	Type
`database_name`	The name of the database that contains this schema.	`VARCHAR`
`database_oid`	Internal identifier of the database containing the schema.	`BIGINT`
`schema_name`	The SQL name of the schema.	`VARCHAR`
`internal`	`true` if this is an internal (built-in) schema, `false` if this is a user-defined schema.	`BOOLEAN`
`sql`	Always `NULL`	`VARCHAR`

The `information_schema.schemata` system view provides a more standardized way to obtain metadata about database schemas.

`duckdb_sequences`

The `duckdb_sequences()` function provides metadata about the sequences available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this sequence	`VARCHAR`
`database_oid`	Internal identifier of the database containing the sequence.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the sequence object.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object that contains the sequence object.	`BIGINT`
`sequence_name`	The SQL name that identifies the sequence within the schema.	`VARCHAR`
`sequence_oid`	The internal identifier of this sequence object.	`BIGINT`

Column	Description	Type
`temporary`	Whether this sequence is temporary. Temporary sequences are transient and only visible within the current connection.	`BOOLEAN`
`start_value`	The initial value of the sequence. This value will be returned when `nextval()` is called for the very first time on this sequence.	`BIGINT`
`min_value`	The minimum value of the sequence.	`BIGINT`
`max_value`	The maximum value of the sequence.	`BIGINT`
`increment_by`	The value that is added to the current value of the sequence to draw the next value from the sequence.	`BIGINT`
`cycle`	Whether the sequence should start over when drawing the next value would result in a value outside the range.	`BOOLEAN`
`last_value`	`null` if no value was ever drawn from the sequence using `nextval(...)`. `1` if a value was drawn.	`BIGINT`
`sql`	The definition of this object, expressed as SQL DDL-statement.	`VARCHAR`

Attributes like `temporary`, `start_value` etc. correspond to the various options available in the `CREATE SEQUENCE` statement and are documented there in full. Note that the attributes will always be filled out in the `duckdb_sequences` resultset, even if they were not explicitly specified in the `CREATE SEQUENCE` statement.

> **Note.**
>
> 1. The column name `last_value` suggests that it contains the last value that was drawn from the sequence, but that is not the case. It's either `null` if a value was never drawn from the sequence, or `1` (when there was a value drawn, ever, from the sequence).
> 2. If the sequence cycles, then the sequence will start over from the boundary of its range, not necessarily from the value specified as start value.

duckdb_settings

The `duckdb_settings()` function provides metadata about the settings available in the DuckDB instance.

Column	Description	Type
`name`	Name of the setting.	`VARCHAR`
`value`	Current value of the setting.	`VARCHAR`
`description`	A description of the setting.	`VARCHAR`
`input_type`	The logical datatype of the setting's value.	`VARCHAR`

The various settings are described in the configuration page.

duckdb_tables

The `duckdb_tables()` function provides metadata about the base tables available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this table	`VARCHAR`
`database_oid`	Internal identifier of the database containing the table.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the base table.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object that contains the base table.	`BIGINT`
`table_name`	The SQL name of the base table.	`VARCHAR`
`table_oid`	Internal identifier of the base table object.	`BIGINT`
`internal`	`false` if this is a user-defined table.	`BOOLEAN`
`temporary`	Whether this is a temporary table. Temporary tables are not persisted and only visible within the current connection.	`BOOLEAN`

Column	Description	Type
has_primary_key	true if this table object defines a PRIMARY KEY.	BOOLEAN
estimated_size	The estimated number of rows in the table.	BIGINT
column_count	The number of columns defined by this object	BIGINT
index_count	The number of indexes associated with this table. This number includes all secondary indexes, as well as internal indexes generated to maintain PRIMARY KEY and/or UNIQUE constraints.	BIGINT
check_constraint_count	The number of check constraints active on columns within the table.	BIGINT
sql	The definition of this object, expressed as SQL CREATE TABLE-statement.	VARCHAR

The information_schema.tables system view provides a more standardized way to obtain metadata about database tables that also includes views. But the resultset returned by duckdb_tables contains a few columns that are not included in information_schema.tables.

duckdb_types

The duckdb_types() function provides metadata about the data types available in the DuckDB instance.

Column	Description	Type
database_name	The name of the database that contains this schema.	VARCHAR
database_oid	Internal identifier of the database that contains the data type.	BIGINT
schema_name	The SQL name of the schema containing the type definition. Always main.	VARCHAR
schema_oid	Internal identifier of the schema object.	BIGINT

Column	Description	Type
`type_name`	The name or alias of this data type.	`VARCHAR`
`type_oid`	The internal identifier of the data type object. If `NULL`, then this is an alias of the type (as identified by the value in the `logical_type` column).	`BIGINT`
`type_size`	The number of bytes required to represent a value of this type in memory.	`BIGINT`
`logical_type`	The 'canonical' name of this data type. The same `logical_type` may be referenced by several types having different `type_name`s.	`VARCHAR`
`type_category`	The category to which this type belongs. Data types within the same category generally expose similar behavior when values of this type are used in expression. For example, the `NUMERIC` type_category includes integers, decimals, and floating point numbers.	`VARCHAR`
`internal`	Whether this is an internal (built-in) or a user object.	`BOOLEAN`

`duckdb_views`

The `duckdb_views()` function provides metadata about the views available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this view	`VARCHAR`
`database_oid`	Internal identifier of the database that contains this view.	`BIGINT`
`schema_name`	The SQL name of the schema where the view resides.	`VARCHAR`

Column	Description	Type
`schema_oid`	Internal identifier of the schema object that contains the view.	`BIGINT`
`view_name`	The SQL name of the view object.	`VARCHAR`
`view_oid`	The internal identifier of this view object.	`BIGINT`
`internal`	`true` if this is an internal (built-in) view, `false` if this is a user-defined view.	`BOOLEAN`
`temporary`	`true` if this is a temporary view. Temporary views are not persistent and are only visible within the current connection.	`BOOLEAN`
`column_count`	The number of columns defined by this view object.	`BIGINT`
`sql`	The definition of this object, expressed as SQL DDL-statement.	`VARCHAR`

The `information_schema.tables` system view provides a more standardized way to obtain metadata about database views that also includes base tables. But the resultset returned by `duckdb_views` contains also definitions of internal view objects as well as a few columns that are not included in `information_schema.tables`.

`duckdb_temporary_files`

The `duckdb_temporary_files()` function provides metadata about the temporary files DuckDB has written to disk, to offload data from memory. This function mostly exists for debugging and testing purposes.

Column	Description	Type
`path`	The name of the temporary file	`VARCHAR`
`size`	The size in bytes of the temporary file	`INT64`

Pragmas

The PRAGMA statement is an SQL extension adopted by DuckDB from SQLite. PRAGMA statements can be issued in a similar manner to regular SQL statements. PRAGMA commands may alter the internal state of the database engine, and can influence the subsequent execution or behavior of the engine.

List of Supported PRAGMA statements

Below is a list of supported PRAGMA statements.

database_list, show_tables, show_tables_expanded, table_info, show, functions

```
-- List all databases, usually one
PRAGMA database_list;
-- List all tables
PRAGMA show_tables;
-- List all tables, with extra information, similar to DESCRIBE
PRAGMA show_tables_expanded;
-- Get info for a specific table
PRAGMA table_info('table_name');
CALL pragma_table_info('table_name');
-- Also show table structure, but slightly different format (for
 ↪  compatibility)
PRAGMA show('table_name');
-- List all functions
PRAGMA functions;
```

`table_info` returns information about the columns of the table with name *table_name*. The exact format of the table returned is given below:

```
cid INTEGER,         -- cid of the column
name VARCHAR,        -- name of the column
type VARCHAR,        -- type of the column
notnull BOOLEAN,     -- if the column is marked as NOT NULL
dflt_value VARCHAR, -- default value of the column, or NULL if not specified
pk BOOLEAN           -- part of the primary key or not
```

memory_limit, threads

```
-- set the memory limit
PRAGMA memory_limit='1GB';
```

```
-- set the amount of threads for parallel query execution
PRAGMA threads=4;
```

database_size

```
-- get the file and memory size of each database
PRAGMA database_size;
CALL pragma_database_size();
```

`database_size` returns information about the file and memory size of each database. The column types of the returned results are given below:

```
database_name VARCHAR, -- database name
database_size VARCHAR, -- total block count times the block size
block_size BIGINT,     -- database block size
total_blocks BIGINT,   -- total blocks in the database
used_blocks BIGINT,    -- used blocks in the database
free_blocks BIGINT,    -- free blocks in the database
wal_size VARCHAR,      -- write ahead log size
memory_usage VARCHAR,  -- memory used by the database buffer manager
memory_limit VARCHAR   -- maximum memory allowed for the database
```

collations, default_collation

```
-- list all available collations
PRAGMA collations;
-- set the default collation to one of the available ones
PRAGMA default_collation='nocase';
```

default_null_order, default_order

```
-- set the ordering for NULLs to be either NULLS FIRST or NULLS LAST
PRAGMA default_null_order='NULLS LAST';
-- set the default result set ordering direction to ASCENDING or DESCENDING
PRAGMA default_order='DESCENDING';
```

version

```
-- show DuckDB version
PRAGMA version;
CALL pragma_version();
```

platform `platform` returns an identifier for the platform the current DuckDB executable has been compiled for. This matches the platform_name as described on the extension loading explainer.

```
-- show platform of current DuckDB executable
PRAGMA platform;
CALL pragma_platform();
```

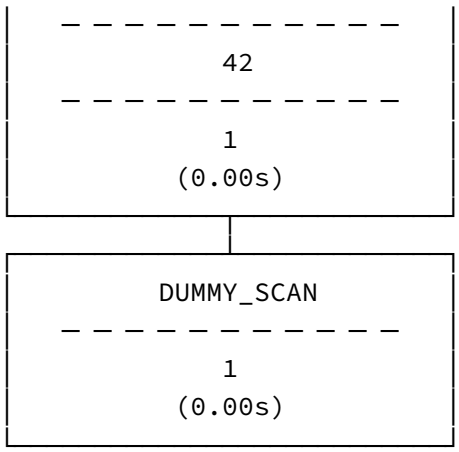
enable_progress_bar, disable_progress_bar, enable_profiling, disable_profiling, profiling_output

```
-- Show progress bar when running queries
PRAGMA enable_progress_bar;
-- Don't show a progress bar for running queries
PRAGMA disable_progress_bar;
-- Enable profiling
PRAGMA enable_profiling;
-- Enable profiling in a specified format
PRAGMA enable_profiling=[json, query_tree, query_tree_optimizer]
-- Disable profiling
PRAGMA disable_profiling;
-- Specify a file to save the profiling output to
PRAGMA profiling_output='/path/to/file.json';
PRAGMA profile_output='/path/to/file.json';
```

Enable the gathering and printing of profiling information after the execution of a query. Optionally, the format of the resulting profiling information can be specified as either *json*, *query_tree*, or *query_tree_optimizer*. The default format is *query_tree*, which prints the physical operator tree together with the timings and cardinalities of each operator in the tree to the screen.

Below is an example output of the profiling information for the simple query `SELECT 42`:

```
┌─────────────────────────────────────┐
│┌───────────────────────────────────┐│
│└───────────────────────────────────┘│
│    Query Profiling Information      │
└─────────────────────────────────────┘
SELECT 42;
┌─────────────────────────────────────┐
│┌───────────────────────────────────┐│
│└───────────────────────────────────┘│
│        Total Time: 0.0001s          │
└─────────────────────────────────────┘

┌─────────────────────────┐
│         PROJECTION      │
```



The printing of profiling information can be disabled again using *disable_profiling*.

By default, profiling information is printed to the console. However, if you prefer to write the profiling information to a file the PRAGMA `profiling_output` can be used to write to a specified file. **Note that the file contents will be overwritten for every new query that is issued, hence the file will only contain the profiling information of the last query that is run.**

disable_optimizer, enable_optimizer

```
-- disables the query optimizer
PRAGMA disable_optimizer;
-- enables the query optimizer
PRAGMA enable_optimizer;
```

log_query_path, explain_output, enable_verification, disable_verification, verify_parallelism, disable_verify_parallelism

```
-- Set a path for query logging
PRAGMA log_query_path='/tmp/duckdb_log/';
-- Disable query logging again
PRAGMA log_query_path='';
-- either show 'all' or only 'optimized' plans in the EXPLAIN output
PRAGMA explain_output='optimized';
-- Enable query verification (for development)
PRAGMA enable_verification;
-- Disable query verification (for development)
PRAGMA disable_verification;
-- Enable force parallel query processing (for development)
PRAGMA verify_parallelism;
-- Disable force parallel query processing (for development)
```

```
PRAGMA disable_verify_parallelism;
-- Force index joins where applicable
PRAGMA force_index_join;
```

These are PRAGMAs mostly used for development and internal testing.

create_fts_index, drop_fts_index Only available when the FTS extension is built, documented here.

verify_external, disable_verify_external

```
-- Enable verification of external operators
PRAGMA verify_external;
-- Disable verification of external operators
PRAGMA disable_verify_external;
```

verify_serializer, disable_verify_serializer

```
-- Enable verification of round-trip capabilities for supported Logical
 ↪  Plans
PRAGMA verify_serializer;
-- Disable verification of round-trip capabilities
PRAGMA disable_verify_serializer;
```

enable_object_cache, disable_object_cache

```
-- Enable caching of objects for e.g., Parquet metadata
PRAGMA enable_object_cache;
-- Disable caching of objects
PRAGMA disable_object_cache;
```

force_checkpoint

```
-- When CHECKPOINT is called when no changes are made, force a CHECKPOINT
 ↪  regardless.
PRAGMA force_checkpoint;
```

enable_print_progress_bar, disable_print_progress_bar

```
-- Enable printing of the progress bar, if it's enabled
PRAGMA enable_print_progress_bar;
```

```
-- Disable printing of the progress bar
PRAGMA disable_print_progress_bar;
```

enable_checkpoint_on_shutdown, disable_checkpoint_on_shutdown

```
-- Run a CHECKPOINT on successful shutdown and delete the WAL, to leave only
 ↪  a single database file behind
PRAGMA enable_checkpoint_on_shutdown;
-- Don't run a CHECKPOINT on shutdown
PRAGMA disable_checkpoint_on_shutdown;
```

temp directory for spilling data to disk -- defaults to .tmp

```
PRAGMA temp_directory='/path/to/temp.tmp'
```

storage_info

```
PRAGMA storage_info('table_name');
CALL pragma_storage_info('table_name');
```

This call returns the following information for the given table:

name	type	description
row_group_id	BIGINT	
column_name	VARCHAR	
column_id	BIGINT	
column_path	VARCHAR	
segment_id	BIGINT	
segment_type	VARCHAR	
start	BIGINT	The start row id of this chunk
count	BIGINT	The amount of entries in this storage chunk
compression	VARCHAR	Compression type used for this column - see blog post
stats	VARCHAR	
has_updates	BOOLEAN	
persistent	BOOLEAN	false if temporary table
block_id	BIGINT	empty unless persistent

name	type	description
`block_offset`	`BIGINT`	empty unless persistent

See Storage for more information.

Rules for Case Sensitivity

Keywords and Function Names

SQL keywords and function names are case-insensitive in DuckDB.

Examples The following two queries are equivalent:

```
select COS(Pi()) as CosineOfPi;
SELECT cos(pi()) AS CosineOfPi;
```

```
┌──────────────┐
│ CosineOfPi   │
│   double     │
├──────────────┤
│         -1.0 │
└──────────────┘
```

Identifiers

Following the convention of the SQL standard, identifiers in DuckDB are case-insensitive. However, each character's case (uppercase/lowercase) is maintained as entered by the user.

Examples The case entered by the user is preserved even if a query uses different cases when referring to the identifier:

```
CREATE TABLE CosPi AS SELECT cos(pi()) AS CosineOfPi;
SELECT cosineofpi FROM CosPi;
```

```
┌──────────────┐
│ CosineOfPi   │
│   double     │
├──────────────┤
```

```
┌──────────┐
│     -1.0 │
└──────────┘
```

In case of a conflict, when the same identifier is spelt with different cases, one will be selected randomly. For example:

```
CREATE TABLE t1(idfield int, x int);
CREATE TABLE t2(IdField int, y int);
SELECT * FROM t1 NATURAL JOIN t2;
```

```
┌─────────┬─────────┬─────────┐
│ idfield │    x    │    y    │
│  int32  │  int32  │  int32  │
├─────────┴─────────┴─────────┤
│          0 rows             │
└─────────────────────────────┘
```

Samples

Samples are used to randomly select a subset of a dataset.

Examples

```
-- select a sample of 5 rows from "tbl" using reservoir sampling
SELECT * FROM tbl USING SAMPLE 5;
-- select a sample of 10% of the table using system sampling (cluster
 ↪  sampling)
SELECT * FROM tbl USING SAMPLE 10%;
-- select a sample of 10% of the table using bernoulli sampling
SELECT * FROM tbl USING SAMPLE 10 PERCENT (bernoulli);
-- select a sample of 50 rows of the table using reservoir sampling with a
 ↪  fixed seed (100)
SELECT * FROM tbl USING SAMPLE reservoir(50 ROWS) REPEATABLE (100);
-- select a sample of 20% of the table using system sampling with a fixed
 ↪  seed (377)
SELECT * FROM tbl USING SAMPLE 10% (system, 377);
-- select a sample of 10% of "tbl" BEFORE the join with tbl2
SELECT * FROM tbl TABLESAMPLE RESERVOIR(20%), tbl2 WHERE tbl.i=tbl2.i;
-- select a sample of 10% of "tbl" AFTER the join with tbl2
SELECT * FROM tbl, tbl2 WHERE tbl.i=tbl2.i USING SAMPLE RESERVOIR(20%);
```

Syntax Samples allow you to randomly extract a subset of a dataset. Samples are useful for exploring a dataset faster, as often you might not be interested in the exact answers to queries, but only in rough indications of what the data looks like and what is in the data. Samples allow you to get approximate answers to queries faster, as they reduce the amount of data that needs to pass through the query engine.

DuckDB supports three different types of sampling methods: `reservoir`, `bernoulli` and `system`. By default, DuckDB uses `reservoir` sampling when an exact number of rows is sampled, and `system` sampling when a percentage is specified. The sampling methods are described in detail below.

Samples require a *sample size*, which is an indication of how many elements will be sampled from the total population. Samples can either be given as a percentage (`10%`) or as a fixed number of rows (`10 rows`). All three sampling methods support sampling over a percentage, but **only** reservoir sampling supports sampling a fixed number of rows.

Samples are probablistic, that is to say, samples can be different between runs *unless* the seed is specifically specified. Specifying the seed *only* guarantees that the sample is the same if multi-threading is not enabled (i.e., `PRAGMA threads=1`). In the case of multiple threads running over a sample, samples are not necessarily consistent even with a fixed seed.

reservoir Reservoir sampling is a stream sampling technique that selects a random sample by keeping a *reservoir* of size equal to the sample size, and randomly replacing elements as more elements come in. Reservoir sampling allows us to specify *exactly* how many elements we want in the resulting sample (by selecting the size of the reservoir). As a result, reservoir sampling *always* outputs the same amount of elements, unlike system and bernoulli sampling.

Reservoir sampling is only recommended for small sample sizes, and is not recommended for use with percentages. That is because reservoir sampling needs to materialize the entire sample and randomly replace tuples within the materialized sample. The larger the sample size, the higher the performance hit incurred by this process.

Reservoir sampling also incurs an additional performance penalty when multi-processing is used, since the reservoir is to be shared amongst the different threads to ensure unbiased sampling. This is not a big problem when the reservoir is very small, but becomes costly when the sample is large.

> **Note.** Avoid using Reservoir Sample with large sample sizes if possible. Reservoir sampling requires the entire sample to be materialized in memory.

bernoulli Bernoulli sampling can only be used when a sampling percentage is specified. It is rather straightforward: every tuple in the underlying table is included with a chance equal to the specified

percentage. As a result, bernoulli sampling can return a different number of tuples even if the same percentage is specified. The amount of rows will generally be more or less equal to the specified percentage of the table, but there will be some variance.

Because bernoulli sampling is completely independent (there is no shared state), there is no penalty for using bernoulli sampling together with multiple threads.

system System sampling is a variant of bernoulli sampling with one crucial difference: every *vector* is included with a chance equal to the sampling percentage. This is a form of cluster sampling. System sampling is more efficient than bernoulli sampling, as no per-tuple selections have to be performed. There is almost no extra overhead for using system sampling, whereas bernoulli sampling can add additional cost as it has to perform random number generation for every single tuple.

System sampling is not suitable for smaller data sets as the granularity of the sampling is on the order of ~1000 tuples. That means that if system sampling is used for small data sets (e.g., 100 rows) either all the data will be filtered out, or all the data will be included.

Table Samples

The `TABLESAMPLE` and `USING SAMPLE` clauses are identical in terms of syntax and effect, with one important difference: tablesamples sample directly from the table for which they are specified, whereas the sample clause samples after the entire from clause has been resolved. This is relevant when there are joins present in the query plan.

The `TABLESAMPLE` clause is essentially equivalent to creating a subquery with the `USING SAMPLE` clause, i.e., the following two queries are identical:

```
-- sample 20% of tbl BEFORE the join
SELECT * FROM tbl TABLESAMPLE RESERVOIR(20%), tbl2 WHERE tbl.i=tbl2.i;
-- sample 20% of tbl BEFORE the join
SELECT * FROM (SELECT * FROM tbl USING SAMPLE RESERVOIR(20%)) tbl, tbl2
 ↪  WHERE tbl.i=tbl2.i;
-- sample 20% AFTER the join (i.e., sample 20% of the join result)
SELECT * FROM tbl, tbl2 WHERE tbl.i=tbl2.i USING SAMPLE RESERVOIR(20%);
```

Window Functions

Examples

```
-- generate a "row_number" column containing incremental identifiers for
 ↪  each row
SELECT row_number() OVER () FROM sales;
-- generate a "row_number" column, by order of time
SELECT row_number() OVER (ORDER BY time) FROM sales;
-- generate a "row_number" column, by order of time partitioned by region
SELECT row_number() OVER (PARTITION BY region ORDER BY time) FROM sales;
-- compute the difference between the current amount, and the previous
 ↪  amount, by order of time
SELECT amount - lag(amount) OVER (ORDER BY time) FROM sales;
-- compute the percentage of the total amount of sales per region for each
 ↪  row
SELECT amount / SUM(amount) OVER (PARTITION BY region) FROM sales;
```

Syntax

Window functions can only be used in the SELECT clause. To share OVER specifications between functions, use the statement's WINDOW clause and use the `OVER window-name` syntax.

General-Purpose Window Functions

The table below shows the available general window functions.

Function	Return Type	Description	Example
`row_number()`	`bigint`	The number of the current row within the partition, counting from 1.	`row_number()`
`rank()`	`bigint`	The rank of the current row *with gaps*; same as `row_number` of its first peer.	`rank()`
`dense_rank()`	`bigint`	The rank of the current row *without gaps*; this function counts peer groups.	`dense_rank()`

Function	Return Type	Description	Example
`rank_dense()`	`bigint`	Alias for `dense_rank`.	`rank_dense()`
`percent_rank()`	`double`	The relative rank of the current row: `(rank() - 1) / (total partition rows - 1)`.	`percent_rank()`
`cume_dist()`	`double`	The cumulative distribution: (number of partition rows preceding or peer with current row) / total partition rows.	`cume_dist()`
`ntile(num_buckets integer)`	`bigint`	An integer ranging from 1 to the argument value, dividing the partition as equally as possible.	`ntile(4)`
`lag(expr any [, offset integer [, default any ]])`	same type as **expr**	Returns `expr` evaluated at the row that is `offset` rows before the current row within the partition; if there is no such row, instead return `default` (which must be of the same type as `expr`). Both `offset` and `default` are evaluated with respect to the current row. If omitted, `offset` defaults to 1 and default to `null`.	`lag(column, 3, 0)`

Function	Return Type	Description	Example
`lead(expr any [, offset integer [, default any ]])`	same type as **expr**	Returns `expr` evaluated at the row that is `offset` rows after the current row within the partition; if there is no such row, instead return `default` (which must be of the same type as `expr`). Both `offset` and `default` are evaluated with respect to the current row. If omitted, `offset` defaults to `1` and default to `null`.	`lead(column, 3, 0)`
`first_value(expr any)`	same type as **expr**	Returns `expr` evaluated at the row that is the first row of the window frame.	`first_value(column)`
`last_value(expr any)`	same type as **expr**	Returns `expr` evaluated at the row that is the last row of the window frame.	`last_value(column)`
`nth_value(expr any, nth integer)`	same type as **expr**	Returns `expr` evaluated at the nth row of the window frame (counting from 1); null if no such row.	`nth_value(column, 2)`
`first(expr any)`	same type as **expr**	Alias for `first_value`.	`first(column)`
`last(expr any)`	same type as **expr**	Alias for `last_value`.	`last(column)`

Aggregate Window Functions

All aggregate functions can be used in a windowing context.

Ignoring NULLs

The following functions support the `IGNORE NULLS` specification:

Function	Description	Example
`lag(expr any [, offset integer [, default any ]])`	Skips `NULL` values when counting.	`lag(column, 3 IGNORE NULLS)`
`lead(expr any [, offset integer [, default any ]])`	Skips `NULL` values when counting.	`lead(column, 3 IGNORE NULLS)`
`first_value(expr any)`	Skips leading `NULL`s	`first_value(column IGNORE NULLS)`
`last_value(expr any)`	Skips trailing `NULL`s	`last_value(column IGNORE NULLS)`
`nth_value(expr any, nth integer)`	Skips `NULL` values when counting.	`nth_value(column, 2 IGNORE NULLS)`

Note that there is no comma separating the arguments from the `IGNORE NULLS` specification.

The inverse of `IGNORE NULLS` is `RESPECT NULLS`, which is the default for all functions.

Evaluation

Windowing works by breaking a relation up into independent *partitions*, *ordering* those partitions, and then computing a new column for each row as a function of the nearby values. 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.

The full syntax is shown in the diagram at the top of the page, and this diagram visually illustrates computation environment:

Partition and Ordering Partitioning breaks the relation up into independent, unrelated pieces. Partitioning is optional, and if none is specified then the entire relation is treated as a single partition.

Window functions cannot access values outside of the partition containing the row they are being evaluated at.

Ordering is also optional, but without it the results are not well-defined. Each partition is ordered using the same ordering clause.

Here is a table of power generation data. After partitioning by plant and ordering by date, it will have this layout:

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

Plant	Date	MWh
Worcester	2019-01-12	98963
Worcester	2019-01-13	107170

In what follows, we shall use this table (or small sections of it) to illustrate various pieces of window function evaluation.

The simplest window function is `ROW_NUMBER()`. This function just computes the 1-based row number within the partition using the query:

```
SELECT "Plant", "Date", row_number() OVER (PARTITION BY "Plant" ORDER  BY
 ↪  "Date") AS "Row"
FROM "History"
ORDER BY 1, 2;
```

The result will be

Plant	Date	Row
Boston	2019-01-02	1
Boston	2019-01-03	2
Boston	2019-01-04	3
…	…	…
Worcester	2019-01-02	1
Worcester	2019-01-03	2
Worcester	2019-01-04	3
…	…	…

Note that even though the function is computed with an `ORDER BY` clause, the result does not have to be sorted, so the `SELECT` also needs to be explicitly sorted if that is desired.

Framing Framing specifies a set of rows relative to each row where the function is evaluated. The distance from the current row is given as an expression either `PRECEDING` or `FOLLOWING` the current row. This distance can either be specified as an integral number of `ROWS` or as a `RANGE` delta expression from the value of the ordering expression. For a `RANGE` specification, there must be only

one ordering expression, and it has to support addition and subtraction (i.e., numbers or INTERVALs). The default values for frames are from UNBOUNDED PRECEDING to CURRENT ROW. It is invalid for a frame to start after it ends.

ROW Framing Here is a simple ROW frame query, using an aggregate function:

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

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

Notice that at the edge of the partition, there are only two values added together. This is because frames are cropped to the edge of the partition.

RANGE Framing Returning to the power data, suppose the data is noisy. We might want to compute a 7 day moving average for each plant to smooth out the noise. 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 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). This is the result:

Plant	Date	MWh 7-dayMoving Average
Boston	2019-01-02	517450.75
Boston	2019-01-03	508793.20
Boston	2019-01-04	508529.83

Plant	Date	MWh 7-dayMoving Average
...	...	...
Boston	2019-01-13	499793.00
Worcester	2019-01-02	104768.25
Worcester	2019-01-03	102713.00
Worcester	2019-01-04	102249.50
...	...	...

`WINDOW` Clauses Multiple different `OVER` clauses can be specified in the same `SELECT`, and each will be computed separately. Often, however, we want to use the same layout for multiple window functions. The `WINDOW` clause can be used to define a *named* window that can be shared between multiple window functions:

```
SELECT "Plant", "Date",
    MIN("MWh") OVER seven AS "MWh 7-day Moving Minimum",
    AVG("MWh") OVER seven AS "MWh 7-day Moving Average",
    MAX("MWh") OVER seven AS "MWh 7-day Moving Maximum"
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;
```

The three window functions will also share the data layout, which will improve performance.

Multiple windows can be defined in the same `WINDOW` clause by comma-separating them:

```
SELECT "Plant", "Date",
    MIN("MWh") OVER seven AS "MWh 7-day Moving Minimum",
    AVG("MWh") OVER seven AS "MWh 7-day Moving Average",
    MAX("MWh") OVER seven AS "MWh 7-day Moving Maximum",
    MIN("MWh") OVER three AS "MWh 3-day Moving Minimum",
    AVG("MWh") OVER three AS "MWh 3-day Moving Average",
    MAX("MWh") OVER three AS "MWh 3-day Moving Maximum"
FROM "Generation History"
WINDOW
    seven AS (
```

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

The queries above do not use a number of clauses commonly found in select statements, like `WHERE`, `GROUP BY`, etc. For more complex queries you can find where `WINDOW` clauses fall in the canonical order of a select statement here.

Box and Whisker Queries All aggregates can be used as windowing functions, including the complex statistical functions. These function implementations have been optimised for windowing, and we can use the window syntax to write queries that generate the data for moving box-and-whisker plots:

```
SELECT "Plant", "Date",
    MIN("MWh") OVER seven AS "MWh 7-day Moving Minimum",
    QUANTILE_CONT("MWh", [0.25, 0.5, 0.75]) OVER seven
        AS "MWh 7-day Moving IQR",
    MAX("MWh") OVER seven AS "MWh 7-day Moving Maximum",
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;
```

Extensions

Extensions

Overview

DuckDB has a flexible extension mechanism that allows for dynamically loading extension. These may extend DuckDB's functionality by providing support for additional file formats, introducing new types, and domain-specific functionality.

> **Note.** Extensions are loadable on all clients (e.g., Python and R). Extensions distributed via the official repository are built and tested on MacOS (AMD64 and ARM64), Windows (AMD64) and Linux (AMD64 and ARM64).

We maintain a list of official extensions.

Using Extensions

Listing Extensions To get a list of extensions, run:

```
FROM duckdb_extensions();
```

```
┌──────────────────┬──────────┬──────────┬──────────────┬──────────────────────────────────────────
│ extension_name ┆ loaded   ┆ installed ┆ install_path ┆
↪  description                 ┆  aliases  ┆
┆    varchar     ┆ boolean  ┆  boolean  ┆   varchar    ┆
↪  varchar                     ┆ varchar[] ┆
╞══════════════════╪══════════╪══════════╪══════════════╪══════════════════════════════════════════
┆ autocomplete   ┆ true     ┆ true      ┆ (BUILT-IN)   ┆ Add supports for
↪  autocomplete in the shell ┆ []        ┆
┆ ...            ┆ ...      ┆ ...       ┆ ...          ┆ ...
↪  ┆           ┆
└──────────────────┴──────────┴──────────┴──────────────┴──────────────────────────────────────────
```

Extension Types DuckDB has three types of extensions.

Built-In Extensions Built-in extensions are loaded at startup and are immediately available for use.

```
SELECT * FROM 'test.json';
```

This will use the `json` extension to read the JSON file.

> **Note.** To make the DuckDB distribution lightweight, it only contains a few fundamental built-in extensions (e.g., `autocomplete`, `json`, `parquet`), which are loaded automatically.

Autoloadable Extensions Autoloadable extensions are loaded on first use.

```
SELECT * FROM 'https://raw.githubusercontent.com/duckdb/duckdb-
↪ web/main/data/weather.csv';
```

To access files via the HTTPS protocol, DuckDB will automatically load the `httpfs` extension. Similarly, other autoloadable extensions (`aws`, `fts`) will be loaded on-demand. If an extension is not already available locally, it will be installed from the official extension repository (`extensions.duckdb.org`).

Explicitly Loadable Extensions Some extensions make several changes to the running DuckDB instance, hence, autoloading them may not be possible. These extensions have to be installed and loaded using the following SQL statements:

```
INSTALL spatial;
LOAD spatial;
```

```
CREATE TABLE tbl(geom GEOMETRY);
```

Extension Handling through the Python API If you are using the Python API client, you can install and load them with the `install_extension(name: str)` and `load_extension(name: str)` methods.

> **Note.** Autoloadable extensions can also be installed explicitly.

Ensuring the Integrity of Extensions Extensions are signed with a cryptographic key, which also simplifies distribution (this is why they are served over HTTP and not HTTPS). By default, DuckDB uses its built-in public keys to verify the integrity of extension before loading them. All extensions provided by the DuckDB core team are signed.

If you wish to load your own extensions or extensions from third-parties you will need to enable the `allow_unsigned_extensions` flag. To load unsigned extensions using the CLI client, pass the `-unsigned` flag to it on startup.

Installation Location and Sharing Extensions between Clients

Extensions are by default installed under the user's home directory, to `~/.duckdb/extensions/{DuckDB version}/{Platform name}`. For example, the extensions for DuckDB version 0.9.0 on macOS ARM64 (Apple Silicon) are installed to `~/.duckdb/extensions/v0.9.0/osx_arm64`.

The shared installation location allows extensions to be shared between the client APIs *of the same DuckDB version*. For example, if an extension is installed with version 0.9.0 of the CLI client, it is available from the Python, R, etc. client libraries provided that they have access to the user's home directory and use DuckDB version 0.9.0.

To specify a different extension directory, use the `extension_directory` configuration option:

```
SET extension_directory=/path/to/your/extension/directory
```

> **Note.** For development builds, the directory of the extensions corresponds to the Git hash of the build, e.g., `~/.duckdb/extensions/fc2e4b26a6/linux_amd64_gcc4`.

Developing Extensions

The same API that the official extensions use is available for developing extensions. This allows users to extend the functionaly of DuckDB such to suit their domain the best. A template for creating extensions is available in the `extension-template` repository.

Working with Extensions

For more details, see the Working with Extensions page.

Official Extensions

Extension name	Description	Aliases
arrow GitHub	A zero-copy data integration between Apache Arrow and DuckDB	
autocomplete	Adds support for autocomplete in the shell	
aws GitHub	Provides features that depend on the AWS SDK	

Extension name	Description	Aliases
azure GitHub	Adds a filesystem abstraction for Azure blob storage to DuckDB	
excel	Adds support for Excel-like format strings	
fts	Adds support for Full-Text Search Indexes	
httpfs	Adds support for reading and writing files over a HTTP(S) connection	http, https, s3
iceberg GitHub	Adds support for Apache Iceberg	
icu	Adds support for time zones and collations using the ICU library	
inet	Adds support for IP-related data types and functions	
jemalloc	Overwrites system allocator with jemalloc	
json	Adds support for JSON operations	
mysql_scanner GitHub	Adds support for reading from and writing to a MySQL database	
parquet	Adds support for reading and writing Parquet files	
postgres_scanner GitHub	Adds support for reading from a Postgres database	postgres
spatial GitHub	Geospatial extension that adds support for working with spatial data and functions	
sqlite_scanner GitHub	Adds support for reading SQLite database files	sqlite, sqlite3
substrait GitHub	Adds support for the Substrait integration	
tpcds	Adds TPC-DS data generation and query support	
tpch	Adds TPC-H data generation and query support	

Working with Extensions

Downloading Extensions Directly from S3

Downloading an extension directly could be helpful when building a lambda or container that uses DuckDB. DuckDB extensions are stored in public S3 buckets, but the directory structure of those buckets is not searchable. As a result, a direct URL to the file must be used. To directly download an extension file, use the following format:

```
http://extensions.duckdb.org/v{release_version_number}/{platform_
↪  name}/{extension_name}.duckdb_extension.gz
```

For example:

```
http://extensions.duckdb.org/v{{ site.currentduckdbversion }}/windows_
↪  amd64/json.duckdb_extension.gz
```

The list of supported platforms may increase over time, but the current list of platforms includes:

- linux_amd64_gcc4
- linux_amd64
- linux_arm64
- osx_amd64
- osx_arm64
- wasm_eh DuckDB-Wasm's extensions
- wasm_mvp DuckDB-Wasm's extensions
- windows_amd64
- windows_amd64_rtools

See above for a list of extension names and how to pull the latest list of extensions.

Loading an Extension from Local Storage

Extensions are stored in gzip format, so they must be unzipped prior to use. There are many methods to decompress gzip. Here is a Python example:

```
import gzip
import shutil

with gzip.open('httpfs.duckdb_extension.gz','rb') as f_in:
   with open('httpfs.duckdb_extension', 'wb') as f_out:
     shutil.copyfileobj(f_in, f_out)
```

After unzipping, the install and load commands can be used with the path to the `.duckdb_extension` file. For example, if the file was unzipped into the same directory as where DuckDB is being executed:

```
INSTALL 'httpfs.duckdb_extension';
LOAD 'httpfs.duckdb_extension';
```

Arrow Extension

The `arrow` extension implements provides features for using Apache Arrow, a cross-language development platform for in-memory analytics.

Installing and Loading

The `arrow` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL arrow;
LOAD arrow;
```

Functions

Function	Type	Description
`to_arrow_ipc`	Table in-out-function	Serializes a table into a stream of blobs containing Arrow IPC buffers
`scan_arrow_ipc`	Table function	Scan a list of pointers pointing to Arrow IPC buffers

AutoComplete Extension

This extension adds supports for autocomplete in the CLI client.

Function	Description
`sql_auto_complete(query_string)`	Attempts autocompletion on the given `query_string`.

Example

```
SELECT * FROM sql_auto_complete('SEL');
```

Returns:

suggestion	suggestion_start
SELECT	0
DELETE	0
INSERT	0
CALL	0
LOAD	0
CALL	0
ALTER	0
BEGIN	0
EXPORT	0
CREATE	0
PREPARE	0
EXECUTE	0
EXPLAIN	0
ROLLBACK	0
DESCRIBE	0
SUMMARIZE	0
CHECKPOINT	0
DEALLOCATE	0
UPDATE	0

suggestion	suggestion_start
DROP	0

AWS Extension

The aws extension provides features that depend on the AWS SDK.

> **Note.** This extension is currently in an experimental state. Feel free to try it out, but be aware some things may not work as expected.

Features

function	type	description
`load_aws_credentials`	PRAGMA function	Automatically loads the AWS credentials through the AWS Default Credentials Provider Chain

Usage

Load AWS Credentials To load the AWS credentials, run:

```
CALL load_aws_credentials();
```

```
┌──────────────────────┬──────────────────────────┬──────────────────────┬──────────────────────┐
│ loaded_access_key_id │ loaded_secret_access_key │ loaded_session_token │ loaded_region │
│       varchar        │         varchar          │       varchar        │       varchar        │
├──────────────────────┼──────────────────────────┼──────────────────────┼──────────────────────┤
│ AKIAIOSFODNN7EXAMPLE │ <redacted>               │                      │ eu-west-1            │
└──────────────────────┴──────────────────────────┴──────────────────────┴──────────────────────┘
```

The function takes a string parameter to specify a specific profile:

```
CALL load_aws_credentials('minio-testing-2');
```

```
┌──────────────────────┬──────────────────────────┬──────────────────────┬──────────
│ loaded_access_key_id │ loaded_secret_access_key │ loaded_session_token │
↪  loaded_region │
│       varchar        │         varchar          │       varchar        │
↪  varchar     │
├──────────────────────┼──────────────────────────┼──────────────────────┼──────────
│ minio_duckdb_user_2  │ <redacted>               │                      │
↪  eu-west-2      │
└──────────────────────┴──────────────────────────┴──────────────────────┴──────────
```

There are several parameters to tweak the behavior of the call:

```
CALL load_aws_credentials('minio-testing-2', set_region=false, redact_
↪  secret=false);
```

```
┌──────────────────────┬──────────────────────────┬──────────────────────┬──────────
│ loaded_access_key_id │   loaded_secret_access_key   │ loaded_session_token
↪  │ loaded_region │
│       varchar        │          varchar         │       varchar        │
↪  varchar     │
├──────────────────────┼──────────────────────────┼──────────────────────┼──────────
│ minio_duckdb_user_2  │ minio_duckdb_user_password_2 │
↪  │               │
└──────────────────────┴──────────────────────────┴──────────────────────┴──────────
```

Related Extensions

aws depends on `httpfs` extension capablities, and both will be autoloaded on the first call to `load_aws_credentials`. If autoinstall or autoload are disabled, you can always explicitly install and load httpfs and aws like:

```
INSTALL aws;
LOAD aws;
INSTALL httpfs;
LOAD httpfs;
```

See also the S3 API capabilities of the `httpfs` extension.

GitHub Repository

GitHub

Azure Extension

The `azure` extension is a loadable extension that adds a filesystem abstraction for the Azure Blob storage to DuckDB.

> **Note.** This extension is currently in an experimental state. Feel free to try it out, but be aware some things may not work as expected.

Installing and Loading

To install and load the `azure` extension, run:

```
INSTALL azure;
LOAD azure;
```

Usage

Authentication is done by setting the connection string:

```
SET azure_storage_connection_string = '<your_connection_string>';
```

After setting the connection string, the Azure Blob Storage can be queried:

```
SELECT count(*) FROM 'azure://<my_container>/<my_file>.<parquet_or_csv>';
```

Blobs are also supported:

```
SELECT * FROM 'azure://<my_container>/*.csv';
```

GitHub Repository

GitHub

Excel Extension

This extension, contrary to its name, does not provide support for reading Excel files. It instead provides a function that wraps the number formatting functionality of the i18npool library, which formats numbers per Excel's formatting rules.

Excel files can be handled through the `spatial` extension: see the Excel Import and Excel Export pages for instructions.

Functions

Function	Description	Example	Result
`text(`*`number`*`,` *`format_string`*`)`	Format the given number per the rules given in the `format_string`	`text(1234567.897, 'h AM/PM')`	`9 PM`
`excel_text(`*`number`*`,` *`format_string`*`)`	Alias for `text`.	`text(1234567.897, 'h:mm AM/PM')`	`9:31 PM`

Full Text Search Extension

Full Text Search is an extension to DuckDB that allows for search through strings, similar to SQLite's FTS5 extension.

API

The extension adds two PRAGMA statements to DuckDB: one to create, and one to drop an index. Additionally, a scalar macro `stem` is added, which is used internally by the extension.

PRAGMA create_fts_index

```
create_fts_index(input_table, input_id, *input_values, stemmer='porter',
 ↪  stopwords='english',
                ignore='(\\.|[^a-z])+', strip_accents=1, lower=1,
 ↪  overwrite=0)
```

PRAGMA that creates a FTS index for the specified table.

Name	Type	Description
`input_table`	`VARCHAR`	Qualified name of specified table, e.g., `'table_name'` or `'main.table_name'`
`input_id`	`VARCHAR`	Column name of document identifier, e.g., `'document_identifier'`

Name	Type	Description
`input_values…`	VARCHAR	Column names of the text fields to be indexed (vararg), e.g., `'text_field_1'`, `'text_field_2'`, …, `'text_field_N'`, or `'\*'` for all columns in input_table of type VARCHAR
`stemmer`	VARCHAR	The type of stemmer to be used. One of `'arabic'`, `'basque'`, `'catalan'`, `'danish'`, `'dutch'`, `'english'`, `'finnish'`, `'french'`, `'german'`, `'greek'`, `'hindi'`, `'hungarian'`, `'indonesian'`, `'irish'`, `'italian'`, `'lithuanian'`, `'nepali'`, `'norwegian'`, `'porter'`, `'portuguese'`, `'romanian'`, `'russian'`, `'serbian'`, `'spanish'`, `'swedish'`, `'tamil'`, `'turkish'`, or `'none'` if no stemming is to be used. Defaults to `'porter'`
`stopwords`	VARCHAR	Qualified name of table containing a single VARCHAR column containing the desired stopwords, or `'none'` if no stopwords are to be used. Defaults to `'english'` for a pre-defined list of 571 English stopwords
`ignore`	VARCHAR	Regular expression of patterns to be ignored. Defaults to `'(\\.|[^a-z])+'`, ignoring all escaped and non-alphabetic lowercase characters
`strip_accents`	BOOLEAN	Whether to remove accents (e.g., convert á to a). Defaults to `1`
`lower`	BOOLEAN	Whether to convert all text to lowercase. Defaults to `1`
`overwrite`	BOOLEAN	Whether to overwrite an existing index on a table. Defaults to `0`

This PRAGMA builds the index under a newly created schema. The schema will be named after the input table: if an index is created on table `'main.table_name'`, then the schema will be named `'fts_main_table_name'`.

PRAGMA drop_fts_index

```
drop_fts_index(input_table)
```

Drops a FTS index for the specified table.

Name	Type	Description
input_table	VARCHAR	Qualified name of input table, e.g., `'table_name'` or `'main.table_name'`

match_bm25

```
match_bm25(input_id, query_string, fields := NULL, k := 1.2, b:= 0.75,
↪  conjunctive := 0)
```

When an index is built, this retrieval macro is created that can be used to search the index.

Name	Type	Description
input_id	VARCHAR	Column name of document identifier, e.g., `'document_identifier'`
query_string	VARCHAR	The string to search the index for
fields	VARCHAR	Comma-separarated list of fields to search in, e.g., `'text_field_2, text_field_N'`. Defaults to `NULL` to search all indexed fields
k	DOUBLE	Parameter *k1* in the Okapi BM25 retrieval model. Defaults to `1.2`
b	DOUBLE	Parameter *b* in the Okapi BM25 retrieval model. Defaults to `0.75`
conjunctive	BOOLEAN	Whether to make the query conjunctive i.e., all terms in the query string must be present in order for a document to be retrieved

stem

```
stem(input_string, stemmer)
```

Reduces words to their base. Used internally by the extension.

Name	Type	Description
input_string	VARCHAR	The column or constant to be stemmed
stemmer	VARCHAR	The type of stemmer to be used. One of `'arabic'`, `'basque'`, `'catalan'`, `'danish'`, `'dutch'`, `'english'`, `'finnish'`, `'french'`, `'german'`, `'greek'`, `'hindi'`, `'hungarian'`, `'indonesian'`, `'irish'`, `'italian'`, `'lithuanian'`, `'nepali'`, `'norwegian'`, `'porter'`, `'portuguese'`, `'romanian'`, `'russian'`, `'serbian'`, `'spanish'`, `'swedish'`, `'tamil'`, `'turkish'`, or `'none'` if no stemming is to be used.

Example Usage

```
-- create a table and fill it with text data
CREATE TABLE documents(document_identifier VARCHAR, text_content VARCHAR,
 ↪  author VARCHAR, doc_version INTEGER);
INSERT INTO documents
VALUES ('doc1', 'The mallard is a dabbling duck that breeds throughout the
 ↪  temperate.', 'Hannes Mühleisen', 3),
       ('doc2', 'The cat is a domestic species of small carnivorous
 ↪  mammal.', 'Laurens Kuiper', 2);
-- build the index (make both the 'text_content' and 'author' columns
 ↪  searchable)
PRAGMA create_fts_index('documents', 'document_identifier', 'text_content',
 ↪  'author');
-- search the 'author' field index for documents that are written by Hannes
 ↪  - this retrieves 'doc1'
SELECT text_content, score
FROM (SELECT *, fts_main_documents.match_bm25(document_identifier,
 ↪  'Muhleisen', fields := 'author') AS score
      FROM documents) sq
WHERE score IS NOT NULL
  AND doc_version > 2
ORDER BY score DESC;
-- search for documents about 'small cats' - this retrieves 'doc2'
SELECT text_content, score
FROM (SELECT *, fts_main_documents.match_bm25(document_identifier, 'small
 ↪  cats') AS score
```

```
      FROM documents) sq
WHERE score IS NOT NULL
ORDER BY score DESC;
```

> **Note.** The FTS index will not update automatically when input table changes. A workaround of this limitation can be recreating the index to refresh.

httpfs Extension

The `httpfs` extension is an autoloadable extension implementing a file system that allows reading remote/writing remote files. For plain HTTP(S), only file reading is supported. For object storage using the S3 API, the `httpfs` extension supports reading/writing/globbing files.

The `httpfs` extension will be, by default, autoloaded on first use of any functionality exposed by this extension. If you prefer to explicitly install and load this extension, you can always run `INSTALL httpfs` on first use and issue `LOAD httpfs` at the start of every session.

Running Queries over HTTP(S)

With the `httpfs` extension, it is possible to directly query files over the HTTP(S) protocol. This works for all files supported by DuckDB or its various extensions, and provides read-only access.

```
SELECT * FROM 'https://domain.tld/file.extension';
```

For CSV files, files will be downloaded entirely in most cases, due to the row-based nature of the format. For Parquet files, DuckDB can use a combination of the Parquet metadata and HTTP range requests to only download the parts of the file that are actually required by the query. For example, the following query will only read the Parquet metadata and the data for the `column_a` column:

```
SELECT column_a FROM 'https://domain.tld/file.parquet';
```

In some cases even, no actual data needs to be read at all as they only require reading the metadata:

```
SELECT COUNT(*) FROM 'https://domain.tld/file.parquet';
```

Scanning multiple files over HTTP(S) is also supported:

```
SELECT * FROM read_parquet(['https://domain.tld/file1.parquet',
 ↪  'https://domain.tld/file2.parquet']);

-- parquet_scan is an alias of read_parquet, so they are equivalent
SELECT * FROM parquet_scan(['https://domain.tld/file1.parquet',
 ↪  'https://domain.tld/file2.parquet']);
```

Running Queries over S3

The `httpfs` extension supports reading/writing/globbing files on object storage servers using the S3 API. S3 offers a standard API to read and write to remote files (while regular http servers, predating S3, do not offer a common write API). DuckDB conforms to the S3 API, that is now common among industry storage providers.

Requirements The `httpfs` filesystem is tested with AWS S3, Minio, Google Cloud, and lakeFS. Other services that implement the S3 API should also work, but not all features may be supported. Below is a list of which parts of the S3 API are required for each `httpfs` feature.

Feature	Required S3 API features
Public file reads	HTTP Range requests
Private file reads	Secret key or session token authentication
File glob	ListObjectV2
File writes	Multipart upload

Configuration To be able to read or write from S3, the correct region should be set:

```
SET s3_region='us-east-1';
```

Optionally, the endpoint can be configured in case a non-AWS object storage server is used:

```
SET s3_endpoint='<domain>.<tld>:<port>';
```

If the endpoint is not SSL-enabled then run:

```
SET s3_use_ssl=false;
```

Switching between path-style and vhost-style URLs is possible using:

```
SET s3_url_style='path';
```

However, note that this may also require updating the endpoint. For example for AWS S3 it is required to change the endpoint to `s3.<region>.amazonaws.com`.

After configuring the correct endpoint and region, public files can be read. To also read private files, authentication credentials can be added:

```
SET s3_access_key_id='<AWS access key id>';
SET s3_secret_access_key='<AWS secret access key>';
```

Alternatively, session tokens are also supported and can be used instead:

```
SET s3_session_token='<AWS session token>';
```

The aws extension allows for loading AWS credentials.

Per-Request Configuration Aside from the global S3 configuration described above, specific configuration values can be used on a per-request basis. This allows for use of multiple sets of credentials, regions, etc. These are used by including them on the S3 URL as query parameters. All the individual configuration values listed above can be set as query parameters. For instance:

```
SELECT *
FROM 's3://bucket/file.parquet?s3_access_key_id=accessKey&s3_secret_access_
 ↪  key=secretKey';
```

Multiple configurations per query are also allowed:

```
SELECT *
FROM 's3://bucket/file.parquet?s3_region=region&s3_session_token=session_
 ↪  token' T1
INNER JOIN 's3://bucket/file.csv?s3_access_key_id=accessKey&s3_secret_
 ↪  access_key=secretKey' T2;
```

Reading Reading files from S3 is now as simple as:

```
SELECT * FROM 's3://bucket/file.extension';
```

Multiple files are also possible, for example:

```
SELECT * FROM read_parquet(['s3://bucket/file1.parquet',
 ↪  's3://bucket/file2.parquet']);
```

Glob File globbing is implemented using the ListObjectV2 API call and allows to use filesystem-like glob patterns to match multiple files, for example:

```
SELECT * FROM read_parquet('s3://bucket/*.parquet');
```

This query matches all files in the root of the bucket with the parquet extension.

Several features for matching are supported, such as `*` to match any number of any character, `?` for any single character or `[0-9]` for a single character in a range of characters:

```
SELECT COUNT(*) FROM read_
 ↪  parquet('s3://bucket/folder*/100?/t[0-9].parquet');
```

A useful feature when using globs is the `filename` option which adds a column with the file that a row originated from:

```
SELECT * FROM read_parquet('s3://bucket/*.parquet', FILENAME = 1);
```

could for example result in:

column_a	column_b	filename
1	examplevalue1	s3://bucket/file1.parquet
2	examplevalue1	s3://bucket/file2.parquet

Hive Partitioning DuckDB also offers support for the Hive partitioning scheme. In the Hive partitioning scheme, data is partitioned in separate files. The columns by which the data is partitioned, are not actually in the files, but are encoded in the file path. So for example let us consider three parquet files Hive paritioned by year:

```
s3://bucket/year=2012/file.parquet
s3://bucket/year=2013/file.parquet
s3://bucket/year=2014/file.parquet
```

If scanning these files with the `HIVE_PARTITIONING` option enabled:

```
SELECT * FROM read_parquet('s3://bucket/*/file.parquet', HIVE_PARTITIONING =
↪ 1);
```

could result in:

column_a	column_b	year
1	examplevalue1	2012
2	examplevalue2	2013
3	examplevalue3	2014

Note that the year column does not actually exist in the parquet files, it is parsed from the filenames. Within DuckDB however, these columns behave just like regular columns. For example, filters can be applied on Hive partition columns:

```
SELECT * FROM read_parquet('s3://bucket/*/file.parquet', HIVE_PARTITIONING =
↪ 1) where year=2013;
```

Writing Writing to S3 uses the multipart upload API. This allows DuckDB to robustly upload files at high speed. Writing to S3 works for both CSV and Parquet:

```
COPY table_name TO 's3://bucket/file.extension';
```

Partitioned copy to S3 also works:

```
COPY table TO 's3://my-bucket/partitioned' (FORMAT PARQUET, PARTITION_BY
↪  (part_col_a, part_col_b));
```

An automatic check is performed for existing files/directories, which is currently quite conservative (and on S3 will add a bit of latency). To disable this check and force writing, an `ALLOW_OVERWRITE` flag is added:

```
COPY table TO 's3://my-bucket/partitioned' (FORMAT PARQUET, PARTITION_BY
↪  (part_col_a, part_col_b), ALLOW_OVERWRITE true);
```

The naming scheme of the written files looks like this:

```
s3://my-bucket/partitioned/part_col_a=<val>/part_col_b=<val>/data_<thread_
↪  number>.parquet
```

Configuration Some additional configuration options exist for the S3 upload, though the default values should suffice for most use cases.

setting	description
`s3_uploader_max_parts_per_file`	used for part size calculation, see AWS docs
`s3_uploader_max_filesize`	used for part size calculation, see AWS docs
`s3_uploader_thread_limit`	maximum number of uploader threads

Additionally, most of the configuration options can be set via environment variables:

DuckDB setting	Environment variable	Note
`s3_region`	`AWS_REGION`	Takes priority over `AWS_DEFAULT_REGION`
`s3_region`	`AWS_DEFAULT_REGION`	
`s3_access_key_id`	`AWS_ACCESS_KEY_ID`	

DuckDB setting	Environment variable	Note
`s3_secret_access_key`	`AWS_SECRET_ACCESS_KEY`	
`s3_session_token`	`AWS_SESSION_TOKEN`	
`s3_endpoint`	`DUCKDB_S3_ENDPOINT`	
`s3_use_ssl`	`DUCKDB_S3_USE_SSL`	

Iceberg Extension

The `iceberg` extension is a loadable extension that implements support for the Apache Iceberg format.

Installing and Loading

To install and load the `iceberg` extension, run:

```
INSTALL iceberg;
LOAD iceberg;
```

Usage

To test the examples, download the `iceberg_data.zip` file and unzip it.

Querying Individual Tables

```
SELECT count(*) FROM iceberg_scan('data/iceberg/lineitem_iceberg', ALLOW_
↪  MOVED_PATHS=true);
```

```
51793
```

> **Note.** The `ALLOW_MOVED_PATHS` option ensures that some path resolution is performed, which allows scanning Iceberg tables that are moved.

Access Iceberg Metadata

```
SELECT * FROM iceberg_metadata('data/iceberg/lineitem_iceberg', ALLOW_MOVED_
 ↪  PATHS=true);
```

```
┌──────────────────────────────────────────────────────┬─────────────────┬──────────────────┬──────────┬──────────┬──────────────────────────────────────────────────────┬─────────────┬─────────┐
│                    manifest_path                     │ manifest_sequence_ │ manifest_content │  status  │ content  │                      file_path                       │ file_format │ record_ │
 ↪  number │ manifest_content │ status  │ content  │
 ↪  file_path                                          │ file_format │ record_
 ↪  count │
│                       varchar                        │          int64
 ↪  │     varchar      │ varchar │ varchar  │
 ↪  varchar                                            │   varchar   │    int64    │
├──────────────────────────────────────────────────────┼─────────────────┼──────────────────┼──────────┼──────────┼──────────────────────────────────────────────────────┼─────────────┼─────────┤
│ lineitem_iceberg/metadata/10eaca8a-1e1c-421e-ad6d-b2…  │
 ↪  2 │ DATA             │ ADDED    │ EXISTING │ lineitem_iceberg/data/00041-
 ↪  414-f3c73457-bbd6-4b92-9c15-17b241171b16-00001.parquet │ PARQUET     │
 ↪  51793 │
│ lineitem_iceberg/metadata/10eaca8a-1e1c-421e-ad6d-b2…  │
 ↪  2 │ DATA             │ DELETED  │ EXISTING │ lineitem_iceberg/data/00000-
 ↪  411-0792dcfe-4e25-4ca3-8ada-175286069a47-00001.parquet │ PARQUET     │
 ↪  60175 │
└──────────────────────────────────────────────────────┴─────────────────┴──────────────────┴──────────┴──────────┴──────────────────────────────────────────────────────┴─────────────┴─────────┘
```

Visualizing Snapshots

```
SELECT * FROM iceberg_snapshots('data/iceberg/lineitem_iceberg');
```

```
┌─────────────────┬─────────────────────┬─────────────────────────┬──────────────────────────────────────────────────────┐
│ sequence_number │     snapshot_id     │      timestamp_ms       │
 ↪  manifest_list                                                 │
│     uint64      │       uint64        │        timestamp        │
 ↪  varchar                                                       │
├─────────────────┼─────────────────────┼─────────────────────────┼──────────────────────────────────────────────────────┤
│               1 │ 3776207205136740581 │ 2023-02-15 15:07:54.504 │ lineitem_
 ↪  iceberg/metadata/snap-3776207205136740581-1-cf3d0be5-cf70-453d-ad8f-
 ↪  48fdc412e608.avro
 ↪  │
│               2 │ 7635660646343998149 │ 2023-02-15 15:08:14.73  │ lineitem_
 ↪  iceberg/metadata/snap-7635660646343998149-1-10eaca8a-1e1c-421e-ad6d-
 ↪  b232e5ee23d3.avro
 ↪  │
└─────────────────┴─────────────────────┴─────────────────────────┴──────────────────────────────────────────────────────┘
```

GitHub Repository

GitHub

ICU Extension

The `icu` extension contains an easy-to-use version of the collation/timezone part of the ICU library.

Installing and Loading

To install and load the `icu` extension, run:

```
INSTALL icu;
LOAD icu;
```

Features

The `icu` extension introduces the following features:

- region-dependent collations
- time zones, used for timestamp data types and timestamp functions

inet Extension

The `inet` extension defines the `INET` data type for storing IPv4 network addresses. It supports the CIDR notation for subnet masks (e.g., `198.51.100.0/22`).

Installing and Loading

To install and load the `inet` extension, run:

```
INSTALL inet;
LOAD inet;
```

Examples

```
SELECT '127.0.0.1'::INET AS addr;
```

addr
inet
127.0.0.1

```
CREATE TABLE tbl(id INTEGER, ip INET);
INSERT INTO tbl VALUES (1, '192.168.0.0/16'), (2, '127.0.0.1'), (2,
 ↪  '8.8.8.8');
```

id int32	ip inet
1	192.168.0.0/16
2	127.0.0.1
2	8.8.8.8

jemalloc Extension

The `jemalloc` extension replaces the system's memory allocator with jemalloc. Unlike other DuckDB extensions, the `jemalloc` extension is statically linked and cannot be installed or loaded during runtime.

Availability

The Linux and macOS versions of DuckDB ship with the `jemalloc` extension by default. On Windows, this extension is not available.

JSON Extension

The `json` extension is a loadable extension that implements SQL functions that are useful for reading values from existing JSON, and creating new JSON data.

Example uses

```
-- read a JSON file from disk, auto-infer options
SELECT * FROM 'todos.json';
-- read_json with custom options
SELECT *
FROM read_json('todos.json',
               format='array',
               columns={userId: 'UBIGINT',
```

```
                        id: 'UBIGINT',
                        title: 'VARCHAR',
                        completed: 'BOOLEAN'});

-- write the result of a query to a JSON file
COPY (SELECT * FROM todos) TO 'todos.json';
```

See more examples on the JSON data page.

JSON Type

The JSON extension makes use of the JSON logical type. The JSON logical type is interpreted as JSON, i.e., parsed, in JSON functions rather than interpreted as VARCHAR, i.e., a regular string. All JSON creation functions return values of this type.

We also allow any of our types to be casted to JSON, and JSON to be casted back to any of our types, for example:

```
-- Cast JSON to our STRUCT type
SELECT '{"duck":42}'::JSON::STRUCT(duck INTEGER);
-- {'duck': 42}

-- And back:
SELECT {duck: 42}::JSON;
-- {"duck":42}
```

This works for our nested types as shown in the example, but also for non-nested types:

```
SELECT '2023-05-12'::DATE::JSON;
-- "2023-05-12"
```

The only exception to this behavior is the cast from VARCHAR to JSON, which does not alter the data, but instead parses and validates the contents of the VARCHAR as JSON.

JSON Table Functions

The following two table functions are used to read JSON:

Function	Description
`read_json_objects(`*`filename`*`)`	Read a JSON object from `filename`, where `filename` can also be a list of files or a glob pattern
`read_ndjson_objects(`*`filename`*`)`	Alias for `read_json_objects` with parameter `format` set to `'newline_delimited'`
`read_json_objects_auto(`*`filename`*`)`	Alias for `read_json_objects` with parameter `format` set to `'auto'`

These functions have the following parameters:

Name	Description	Type	Default
`maximum_object_size`	The maximum size of a JSON object (in bytes)	`UINTEGER`	`16777216`
`format`	Can be one of `['auto', 'unstructured', 'newline_delimited', 'array']`	`VARCHAR`	`'array'`
`ignore_errors`	Whether to ignore parse errors (only possible when `format` is `'newline_delimited'`)	`BOOL`	`false`
`compression`	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `t.json.gz` will use gzip, `t.json` will use none). Options are `'none'`, `'gzip'`, `'zstd'`, and `'auto'`.	`VARCHAR`	`'auto'`
`filename`	Whether or not an extra `filename` column should be included in the result.	`BOOL`	`false`
`hive_partitioning`	Whether or not to interpret the path as a hive partitioned path.	`BOOL`	`false`

The `format` parameter specifies how to read the JSON from a file. With `'unstructured'`, the top-level JSON is read, e.g.:

```
{
  "duck": 42
}
{
  "goose": [1, 2, 3]
}
```

Will result in two objects being read.

With `'newline_delimited'`, NDJSON is read, where each JSON is separated by a newline (`\n`), e.g.:

```
{"duck": 42}
{"goose": [1, 2, 3]}
```

Will also result in two objects being read.

With `'array'`, each array element is read, e.g.:

```
[
  {
    "duck": 42
  },
  {
    "goose": [1, 2, 3]
  }
]
```

Again, will result in two objects being read.

Example usage:

```
SELECT * FROM read_json_objects('my_file1.json');
-- {"duck":42,"goose":[1,2,3]}
SELECT * FROM read_json_objects(['my_file1.json', 'my_file2.json']);
-- {"duck":42,"goose":[1,2,3]}
-- {"duck":43,"goose":[4,5,6],"swan":3.3}
SELECT * FROM read_ndjson_objects('*.json.gz');
-- {"duck":42,"goose":[1,2,3]}
-- {"duck":43,"goose":[4,5,6],"swan":3.3}
```

DuckDB also supports reading JSON as a table, using the following functions:

Function	Description
`read_json(`*`filename`*`)`	Read JSON from `filename`, where `filename` can also be a list of files, or a glob pattern
`read_ndjson(`*`filename`*`)`	Alias for `read_json` with parameter `format` set to `'newline_delimited'`
`read_json_auto(`*`filename`*`)`	Alias for `read_json` with all auto-detection enabled
`read_ndjson_auto(`*`filename`*`)`	Alias for `read_json_auto` with parameter `format` set to `'newline_delimited'`

Besides the `maximum_object_size`, `format`, `ignore_errors` and `compression`, these functions have additional parameters:

Name	Description	Type	Default
`columns`	A struct that specifies the key names and value types contained within the JSON file (e.g., `{key1: 'INTEGER', key2: 'VARCHAR'}`). If `auto_detect` is enabled these will be inferred	`STRUCT`	`(empty)`
`records`	Can be one of `['auto', 'true', 'false']`	`VARCHAR`	`'records'`
`auto_detect`	Whether to auto-detect detect the names of the keys and data types of the values automatically	`BOOL`	`false`
`sample_size`	Option to define number of sample objects for automatic JSON type detection. Set to -1 to scan the entire input file	`UBIGINT`	`20480`
`maximum_depth`	Maximum nesting depth to which the automatic schema detection detects types. Set to -1 to fully detect nested JSON types	`BIGINT`	`-1`
`dateformat`	Specifies the date format to use when parsing dates. See Date Format	`VARCHAR`	`'iso'`

Name	Description	Type	Default
`timestampformat`	Specifies the date format to use when parsing timestamps. See Date Format	VARCHAR	`'iso'`
`union_by_name`	Whether the schema's of multiple JSON files should be unified.	BOOL	`false`

Example usage:

```
SELECT * FROM read_json('my_file1.json', columns={duck: 'INTEGER'});
```

duck
42

DuckDB can convert JSON arrays directly to its internal `LIST` type, and missing keys become `NULL`.

```
SELECT *
FROM read_json(['my_file1.json', 'my_file2.json'],
               columns={duck: 'INTEGER', goose: 'INTEGER[]', swan: 'DOUBLE'});
```

duck	goose	swan
42	[1, 2, 3]	NULL
43	[4, 5, 6]	3.3

DuckDB can automatically detect the types like so:

```
SELECT goose, duck FROM read_json_auto('*.json.gz');
SELECT goose, duck FROM '*.json.gz'; -- equivalent
```

goose	duck
[1, 2, 3]	42
[4, 5, 6]	43

DuckDB can read (and auto-detect) a variety of formats, specified with the `format` parameter. Querying a JSON file that contains an `'array'`, e.g.:

```
[
  {
    "duck": 42,
    "goose": 4.2
  },
  {
    "duck": 43,
    "goose": 4.3
  }
]
```

Can be queried exactly the same as a JSON file that contains `'unstructured'` JSON, e.g.:

```
{
  "duck": 42,
  "goose": 4.2
}
{
  "duck": 43,
  "goose": 4.3
}
```

Both can be read as the table:

duck	goose
42	4.2
43	4.3

If your JSON file does not contain 'records', i.e., any other type of JSON than objects, DuckDB can still read it. This is specified with the `records` parameter. The `records` parameter specifies whether the JSON contains records that should be unpacked into individual columns, i.e., reading the following file with `records`:

```
{"duck": 42, "goose": [1, 2, 3]}
{"duck": 43, "goose": [4, 5, 6]}
```

Results in two columns:

duck	goose
42	[1,2,3]

duck	goose
42	[4,5,6]

You can read the same file with `records` set to `'false'`, to get a single column, which is a STRUCT containing the data:

json
{'duck': 42, 'goose': [1,2,3]}
{'duck': 43, 'goose': [4,5,6]}

For additional examples reading more complex data, please see the Shredding Deeply Nested JSON, One Vector at a Time blog post.

JSON Import/Export

When the JSON extension is installed, `FORMAT JSON` is supported for `COPY FROM`, `COPY TO`, `EXPORT DATABASE` and `IMPORT DATABASE`. See Copy and Import/Export.

By default, COPY expects newline-delimited JSON. If you prefer copying data to/from a JSON array, you can specify `ARRAY true`, i.e.,

```
COPY (SELECT * FROM range(5)) TO 'my.json' (ARRAY true);
```

Will create the following file:

```
[
  {"range":0},
  {"range":1},
  {"range":2},
  {"range":3},
  {"range":4}
]
```

This can be read like so:

```
CREATE TABLE test (range BIGINT);
COPY test FROM 'my.json' (ARRAY true);
```

The format can be detected automatically the format like so:

```
COPY test FROM 'my.json' (AUTO_DETECT true);
```

JSON Scalar Functions

The following scalar JSON functions can be used to gain information about the stored JSON values. With the exception of `json_valid(json)`, all JSON functions produce an error when invalid JSON is supplied.

We support two kinds of notations to describe locations within JSON: JSON Pointer and JSONPath.

Function	Description
`json(json)`	Parse and minify `json`
`json_valid(json)`	Return whether `json` is valid JSON
`json_array_length(json[,path])`	Return the number of elements in the JSON array `json`, or `0` if it is not a JSON array. If `path` is specified, return the number of elements in the JSON array at the given `path`. If `path` is a `LIST`, the result will be `LIST` of array lengths
`json_type(json[,path])`	Return the type of the supplied `json`, which is one of `OBJECT`, `ARRAY`, `BIGINT`, `UBIGINT`, `VARCHAR`, `BOOLEAN`, `NULL`. If `path` is specified, return the type of the element at the given `path`. If `path` is a `LIST`, the result will be `LIST` of types
`json_keys(json[,path])`	Returns the keys of `json` as a `LIST` of `VARCHAR`, if `json` is a JSON object. If `path` is specified, return the keys of the JSON object at the given `path`. If `path` is a `LIST`, the result will be `LIST` of `LIST` of `VARCHAR`
`json_structure(json)`	Return the structure of `json`. Defaults to `JSON` the structure is inconsistent (e.g., incompatible types in an array)
`json_contains(json_haystack, json_needle)`	Returns `true` if `json_needle` is contained in `json_haystack`. Both parameters are of JSON type, but `json_needle` can also be a numeric value or a string, however the string must be wrapped in double quotes

The JSONPointer syntax separates each field with a `/`. For example, to extract the first element of the array with key `"duck"`, you can do:

```
SELECT json_extract('{"duck": [1, 2, 3]}', '/duck/0');
-- 1
```

The JSONPath syntax separates fields with a `.`, and accesses array elements with `[i]`, and always starts with `$`. Using the same example, we can do:

```
SELECT json_extract('{"duck": [1, 2, 3]}', '$.duck[0]');
-- 1
```

JSONPath is more expressive, and can also access from the back of lists:

```
SELECT json_extract('{"duck": [1, 2, 3]}', '$.duck[#-1]');
-- 3
```

JSONPath also allows escaping syntax tokens, using double quotes:

```
SELECT json_extract('{"duck.goose": [1, 2, 3]}', '$."duck.goose"[1]');
-- 2
```

Other examples:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
  (' { "family": "anatidae", "species": [ "duck", "goose", "swan", null ]
↪  }');
SELECT json(j) FROM example;
-- {"family":"anatidae","species":["duck","goose","swan",null]}
SELECT json_valid(j) FROM example;
-- true
SELECT json_valid('{');
-- false
SELECT json_array_length('["duck", "goose", "swan", null]');
-- 4
SELECT json_array_length(j, 'species') FROM example;
-- 4
SELECT json_array_length(j, '/species') FROM example;
-- 4
SELECT json_array_length(j, '$.species') FROM example;
-- 4
SELECT json_array_length(j, ['$.species']) FROM example;
-- [4]
SELECT json_type(j) FROM example;
-- OBJECT
SELECT json_keys FROM example;
-- [family, species]
```

```
SELECT json_structure(j) FROM example;
-- {"family":"VARCHAR","species":["VARCHAR"]}
SELECT json_structure('["duck",{"family":"anatidae"}]');
-- ["JSON"]
SELECT json_contains('{"key":"value"}', '"value"');
-- true
SELECT json_contains('{"key":1}', 1);
-- true
SELECT json_contains('{"top_key":{"key":"value"}}', '{"key":"value"}');
-- true
```

JSON Extraction Functions

There are two extraction functions, which have their respective operators. The operators can only be used if the string is stored as the JSON logical type. These functions supports the same two location notations as the previous functions.

Function	Alias	Operator
`json_extract(`*`json`*`,`*`path`*`)`	`json_extract_path`	`->`
`json_extract_string(`*`json`*`,`*`path`*`)`	`json_extract_path_text`	`->>`

Examples:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
  (' { "family": "anatidae", "species": [ "duck", "goose", "swan", null ]
↪  }');
SELECT json_extract(j, '$.family') FROM example;
-- "anatidae"
SELECT j->'$.family' FROM example;
-- "anatidae"
SELECT j->'$.species[0]' FROM example;
-- "duck"
SELECT j->'$.species'->0 FROM example;
-- "duck"
SELECT j->'species'->>[0,1] FROM example;
-- ["duck", "goose"]
SELECT json_extract_string(j, '$.family') FROM example;
```

```
-- anatidae
SELECT j->>'$.family' FROM example;
-- anatidae
SELECT j->>'$.species[0]' FROM example;
-- duck
SELECT j->'species'->>0 FROM example;
-- duck
SELECT j->'species'->>[0,1] FROM example;
-- [duck, goose]
```

If multiple values need to be extracted from the same JSON, it is more efficient to extract a list of paths:

```
-- The following will cause the JSON to be parsed twice,
-- resulting in a slower query that uses more memory
SELECT json_extract(j, 'family') AS family,
       json_extract(j, 'species') AS species
FROM example;
-- The following is faster and more memory efficient
WITH extracted AS (
  SELECT json_extract(j, ['family', 'species']) extracted_list
  FROM example
)
SELECT extracted_list[1] AS family,
       extracted_list[2] AS species
FROM extracted;
```

JSON Creation Functions

The following functions are used to create JSON.

Function	Description
`to_json(`*any*`)`	Create JSON from a value of *any* type. Our `LIST` is converted to a JSON array, and our `STRUCT` and `MAP` are converted to a JSON object
`json_quote(`*any*`)`	Alias for `to_json`
`array_to_json(`*list*`)`	Alias for `to_json` that only accepts `LIST`
`row_to_json(`*list*`)`	Alias for `to_json` that only accepts `STRUCT`
`json_array([`*any*`, ...])`	Create a JSON array from *any* number of values

Function	Description
`json_object([key,value, ...])`	Create a JSON object from any number of *key*, *value* pairs
`json_merge_patch(json,json)`	Merge two json documents together

Examples:

```
SELECT to_json('duck');
-- "duck"
SELECT to_json([1, 2, 3]);
-- [1,2,3]
SELECT to_json({duck : 42});
-- {"duck":42}
SELECT to_json(map(['duck'],[42]));
-- {"duck":42}
SELECT json_array(42, 'duck', NULL);
-- [42,"duck",null]
SELECT json_object('duck', 42);
-- {"duck":42}
SELECT json_merge_patch('{"duck": 42}', '{"goose": 123}');
-- {"goose":123,"duck":42}
```

JSON Aggregate Functions

There are three JSON aggregate functions.

Function	Description
`json_group_array(any)`	Return a JSON array with all values of *any* in the aggregation
`json_group_object(key, value)`	Return a JSON object with all *key*, *value* pairs in the aggregation
`json_group_structure(json)`	Return the combined `json_structure` of all *json* in the aggregation

Examples:

```
CREATE TABLE example (k VARCHAR, v INTEGER);
INSERT INTO example VALUES ('duck', 42), ('goose', 7);
SELECT json_group_array(v) FROM example;
-- [42, 7]
SELECT json_group_object(k, v) FROM example;
-- {"duck":42,"goose":7}
DROP TABLE example;
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
  ('{"family": "anatidae", "species": ["duck", "goose"], "coolness":
 ↪  42.42}'),
  ('{"family": "canidae", "species": ["labrador", "bulldog"], "hair":
 ↪  true}');
SELECT json_group_structure(j) FROM example;
-- {"fam-
 ↪  ily":"VARCHAR","species":["VARCHAR"],"coolness":"DOUBLE","hair":"BOOLEAN"}
```

Transforming JSON

In many cases, it is inefficient to extract values from JSON one-by-one. Instead, we can "extract" all values at once, transforming JSON to the nested types `LIST` and `STRUCT`.

Function	Description
`json_transform(`*`json`*`,` *`structure`*`)`	Transform *`json`* according to the specified *`structure`*
`from_json(`*`json`*`,` *`structure`*`)`	Alias for `json_transform`
`json_transform_strict(`*`json`*`,` *`structure`*`)`	Same as `json_transform`, but throws an error when type casting fails
`from_json_strict(`*`json`*`,` *`structure`*`)`	Alias for `json_transform_strict`

The *`structure`* argument is JSON of the same form as returned by `json_structure`. The *`structure`* argument can be modified to transform the JSON into the desired structure and types. It is possible to extract fewer key/value pairs than are present in the JSON, and it is also possible to extract more: missing keys become `NULL`.

Examples:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
```

```
  ('{"family": "anatidae", "species": ["duck", "goose"], "coolness":
 ↪  42.42}'),
  ('{"family": "canidae", "species": ["labrador", "bulldog"], "hair":
 ↪  true}');
SELECT json_transform(j, '{"family": "VARCHAR", "coolness": "DOUBLE"}') FROM
 ↪  example;
-- {'family': anatidae, 'coolness': 42.420000}
-- {'family': canidae, 'coolness': NULL}
SELECT json_transform(j, '{"family": "TINYINT", "coolness": "DECIMAL(4,
 ↪  2)"}') FROM example;
-- {'family': NULL, 'coolness': 42.42}
-- {'family': NULL, 'coolness': NULL}
SELECT json_transform_strict(j, '{"family": "TINYINT", "coolness":
 ↪  "DOUBLE"}') FROM example;
-- Invalid Input Error: Failed to cast value: "anatidae"
```

De/Serializing SQL to JSON and Vice Versa

The JSON extension also provides functions to serialize and deserialize `SELECT` statements between SQL and JSON, as well as executing JSON serialized statements.

Function	Type	Description
`json_serialize_sql(`*`varchar`*`, skip_empty :=`*`boolean`*`, skip_null :=`*`boolean`*`, format :=`*`boolean`*`)`	Scalar	Serialize a set of `;` separated select statments to an equivalent list of *`json`* serialized statements
`json_deserialize_sql(`*`json`*`)`	Scalar	Deserialize one or many *`json`* serialized statements back to an equivalent sql string
`json_execute_serialized_sql(`*`varchar`*`)`	Table	Execute *`json`* serialized statements and return the resulting rows. Only one statement at a time is supported for now.
`PRAGMA json_execute_serialized_sql(`*`varchar`*`)`	Pragma	Pragma version of the `json_execute_serialized_sql` function.

The `json_serialize_sql(varchar)` function takes three optional parameters, `skip_empty`, `skip_null`, and `format` that can be used to control the output of the serialized statements.

If you run the `json_execute_serialize_sql(varchar)` table function inside of a transaction the serialized statements will not be able to see any transaction local changes. This is because the statements are executed in a separate query context. You can use the `PRAGMA json_execute_serialize_sql(varchar)` pragma version to execute the statements in the same query context as the pragma, although with the limitation that the serialized json must be provided as a constant string. I.E. you cannot do `PRAGMA json_execute_serialize_sql(json_serialize_sql(...))`.

Note that these functions do not preserve syntactic sugar such as `FROM * SELECT ...`, so a statement round-tripped through `json_deserialize_sql(json_serialize_sql(...))` may not be identical to the original statement, but should always be semantically equivalent and produce the same output.

Examples:

```
-- Simple example
SELECT json_serialize_sql('SELECT 2');
-- '{"error":false,"statements":[{"node":{"type":"SELECT_
 ↪  NODE","modifiers":[],"cte_map":{"map":[]},"select_
 ↪  list":[{"class":"CONSTANT","type":"VALUE_
 ↪  CONSTANT","alias":"","value":{"type":{"id":"INTEGER","type_
 ↪  info":null},"is_null":false,"value":2}}],"from_
 ↪  table":{"type":"EMPTY","alias":"","sample":null},"where_
 ↪  clause":null,"group_expressions":[],"group_sets":[],"aggregate_
 ↪  handling":"STANDARD_
 ↪  HANDLING","having":null,"sample":null,"qualify":null}}]}'

-- Example with multiple statements and skip options
SELECT json_serialize_sql('SELECT 1 + 2; SELECT a + b FROM tbl1', skip_empty
 ↪  := true, skip_null := true);
```

```
-- '{"error":false,"statements":[{"node":{"type":"SELECT_NODE","select_
 ↪  list":[{"class":"FUNCTION","type":"FUNCTION","function_
 ↪  name":"+","children":[{"class":"CONSTANT","type":"VALUE_
 ↪  CONSTANT","value":{"type":{"id":"INTEGER"},"is_
 ↪  null":false,"value":1}},{"class":"CONSTANT","type":"VALUE_
 ↪  CONSTANT","value":{"type":{"id":"INTEGER"},"is_
 ↪  null":false,"value":2}}],"order_bys":{"type":"ORDER_
 ↪  MODIFIER"},"distinct":false,"is_operator":true,"export_
 ↪  state":false}],"from_table":{"type":"EMPTY"},"aggregate_
 ↪  handling":"STANDARD_HANDLING"}},{"node":{"type":"SELECT_NODE","select_
 ↪  list":[{"class":"FUNCTION","type":"FUNCTION","function_
 ↪  name":"+","children":[{"class":"COLUMN_REF","type":"COLUMN_
 ↪  REF","column_names":["a"]},{"class":"COLUMN_REF","type":"COLUMN_
 ↪  REF","column_names":["b"]}],"order_bys":{"type":"ORDER_
 ↪  MODIFIER"},"distinct":false,"is_operator":true,"export_
 ↪  state":false}],"from_table":{"type":"BASE_TABLE","table_
 ↪  name":"tbl1"},"aggregate_handling":"STANDARD_HANDLING"}}]}'

-- Example with a syntax error
SELECT json_serialize_sql('TOTALLY NOT VALID SQL');
-- '{"error":true,"error_type":"parser","error_message":"syntax error at or
 ↪  near \"TOTALLY\"\nLINE 1: TOTALLY NOT VALID SQL\n        ^"}'

-- Example with deserialize
SELECT json_deserialize_sql(json_serialize_sql('SELECT 1 + 2'));
-- 'SELECT (1 + 2)'

-- Example with deserialize and syntax sugar
SELECT json_deserialize_sql(json_serialize_sql('FROM x SELECT 1 + 2'));
-- 'SELECT (1 + 2) FROM x'

-- Example with execute
SELECT * FROM json_execute_serialized_sql(json_serialize_sql('SELECT 1 +
 ↪  2'));
-- 3

-- Example with error
SELECT * FROM json_execute_serialized_sql(json_serialize_sql('TOTALLY NOT
 ↪  VALID SQL'));
-- Error: Parser Error: Error parsing json: parser: syntax error at or near
 ↪  "TOTALLY"
```

MySQL Scanner Extension

The `mysql_scanner` extension allows DuckDB to directly read and write data from/to a running MySQL instance. The data can be queried directly from the underlying MySQL database. Data can be loaded from MySQL tables into DuckDB tables, or vice versa.

> **Note.** The MySQL Scanner extension is currently in preview and not yet available as a binary package.

Reading Data from MySQL

To make a MySQL database accessible to DuckDB use the `ATTACH` command:

```
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS
 ↪  mysqlscanner (TYPE mysql_scanner)
USE mysqlscanner;
```

The connection string determines the parameters for how to connect to MySQL as a set of `key=value` pairs. Any options not provided are replaced by their default values, as per the table below.

Setting	Default
host	localhost
user	current user
password	
database	NULL
port	0
socket	NULL

The tables in the file can be read as if they were normal DuckDB tables, but the underlying data is read directly from MySQL at query time.

```
SHOW TABLES;
```

name varchar
signed_integers

```
SELECT * FROM signed_integers;
```

t	s	m	i	b
int8	int16	int32	int32	int64
-128	-32768	-8388608	-2147483648	-9223372036854775808
127	32767	8388607	2147483647	9223372036854775807
NULL	NULL	NULL	NULL	NULL

It might be desirable to create a copy of the MySQL databases in DuckDB to prevent the system from re-reading the tables from MySQL continuously, particularly for large tables.

Data can be copied over from MySQL to DuckDB using standard SQL, for example:

```
CREATE TABLE duckdb_table AS FROM mysqlscanner.mysql_table;
```

Writing Data to MySQL

In addition to reading data from MySQL, create tables, ingest data into MySQL and make other modifications to a MySQL database using standard SQL queries.

This allows you to use DuckDB to, for example, export data that is stored in a MySQL database to Parquet, or read data from a Parquet file into MySQL.

Below is a brief example of how to create a new table in MySQL and load data into it.

```
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db
 ↪  (TYPE mysql_scanner);
CREATE TABLE mysql_db.tbl(id INTEGER, name VARCHAR);
INSERT INTO mysql_db.tbl VALUES (42, 'DuckDB');
```

Many operations on MySQL tables are supported. All these operations directly modify the MySQL database, and the result of subsequent operations can then be read using MySQL. Note that if modifications are not desired, ATTACH can be run with the READ_ONLY property which prevents making modifications to the underlying database. For example:

```
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db
 ↪  (TYPE mysql_scanner, READ_ONLY);
```

Below is a list of supported operations.

CREATE TABLE

```
CREATE TABLE mysql_db.tbl(id INTEGER, name VARCHAR);
```

INSERT INTO

```
INSERT INTO mysql_db.tbl VALUES (42, 'DuckDB');
```

SELECT

```
SELECT * FROM mysql_db.tbl;
```

id	name
int64	varchar
42	DuckDB

COPY

```
COPY mysql_db.tbl TO 'data.parquet';
COPY mysql_db.tbl FROM 'data.parquet';
```

UPDATE

```
UPDATE mysql_db.tbl SET name='Woohoo' WHERE id=42;
```

DELETE

```
DELETE FROM mysql_db.tbl WHERE id=42;
```

ALTER TABLE

```
ALTER TABLE mysql_db.tbl ADD COLUMN k INTEGER;
```

DROP TABLE

```
DROP TABLE mysql_db.tbl;
```

CREATE VIEW

```
CREATE VIEW mysql_db.v1 AS SELECT 42;
```

CREATE SCHEMA/DROP SCHEMA

```
CREATE SCHEMA mysql_db.s1;
CREATE TABLE mysql_db.s1.integers(i int);
INSERT INTO mysql_db.s1.integers VALUES (42);
SELECT * FROM mysql_db.s1.integers;
```

```
┌─────────┐
│    i    │
│  int32  │
├─────────┤
│      42 │
└─────────┘
```

```
DROP SCHEMA mysql_db.s1;
```

Transactions

```
CREATE TABLE mysql_db.tmp(i INTEGER);
BEGIN;
INSERT INTO mysql_db.tmp VALUES (42);
SELECT * FROM mysql_db.tmp;
```

```
┌─────────┐
│    i    │
│  int64  │
├─────────┤
│      42 │
└─────────┘
```

```sql
ROLLBACK;
SELECT * FROM mysql_db.tmp;
```

```
┌─────────┐
│    i    │
│  int64  │
├─────────┤
│ 0 rows  │
└─────────┘
```

> **Note.** Note that DDL statements are not transactional in MySQL.

Building & Loading the Extension

The extension currently cannot be installed from a binary package. To build it, type:

```
make
```

To run, run the bundled `duckdb` shell:

```
./build/release/duckdb -unsigned
```

Then, load the MySQL extension like so:

```
LOAD 'build/release/extension/mysql_scanner/mysql_scanner.duckdb_
 ↪  extension';
```

PostgreSQL Scanner Extension

The `postgres` extension allows DuckDB to directly read data from a running PostgreSQL instance. The data can be queried directly from the underlying PostgreSQL tables, or read into DuckDB tables. See the official announcement for implementation details and background.

Usage

To make a PostgreSQL database accessible to DuckDB, use the `postgres_attach` command:

```
-- load all data from "public" schema of the postgres instance running on
 ↪  localhost into the schema "main"
CALL postgres_attach('');
-- attach the database with the given schema, loading tables from the source
 ↪  schema "public" into the target schema "abc"
CALL postgres_attach('dbname=postgres user=postgres host=127.0.0.1', source_
 ↪  schema='public', sink_schema='abc');
```

`postgres_attach` takes a single required string parameter, which is the `libpq` connection string. For example you can pass `'dbname=postgresscanner'` to select a different database name. In the simplest case, the parameter is just `''`. There are three additional named parameters:

- `source_schema` the name of a non-standard schema name in PostgreSQL to get tables from. Default: `public`.
- `sink_schema` the schema name in DuckDB to create views. Default: `main`.
- `overwrite` whether we should overwrite existing views in the target schema. Default: `false`.
- `filter_pushdown` whether filter predicates that DuckDB derives from the query should be forwarded to PostgreSQL. Default: `false`.

The tables in the database are registered as views in DuckDB, you can list them as follows:

```
PRAGMA show_tables;
```

Then you can query those views normally using SQL.

Querying Individual Tables

If you prefer to not attach all tables, but just query a single table, that is possible using the `postgres_scan` function, e.g.:

```
SELECT * FROM postgres_scan('', 'public', 'mytable');
```

The `postgres_scan` function takes three string parameters, the `libpq` connection string (see above), a PostgreSQL schema name and a table name. The schema often used in PostgreSQL is `public`.

To use `filter_pushdown` use the `postgres_scan_pushdown` function.

Loading the Extension

PostgreSQL extension will be, by default, autoloaded on first use. If you prefer to do so explicitly, it can always be done using the following commands:

```
INSTALL postgres;
LOAD postgres;
```

GitHub Repository

GitHub

Spatial Extension

The `spatial` extension provides support for geospatial data processing in DuckDB. For an overview of the extension, see our blog post.

Installing and Loading

To install and load the `spatial` extension, run:

```
INSTALL spatial;
LOAD spatial;
```

GEOMETRY type

The core of the spatial extension is the `GEOMETRY` type. If you're unfamiliar with geospatial data and GIS tooling, this type probably works very different from what you'd expect.

In short, while the `GEOMETRY` type is a binary representation of "geometry" data made up out of sets of vertices (pairs of `X` and `Y` `double` precision floats), it actually stores one of several geometry subtypes. These are `POINT`, `LINESTRING`, `POLYGON`, as well as their "collection" equivalents, `MULTIPOINT`, `MULTILINESTRING` and `MULTIPOLYGON`. Lastly there is `GEOMETRYCOLLECTION`, which can contain any of the other subtypes, as well as other `GEOMETRYCOLLECTION`s recursively.

This may seem strange at first, since DuckDB already have types like `LIST`, `STRUCT` and `UNION` which could be used in a similar way, but the design and behaviour of the `GEOMETRY` type is actually based on the Simple Features geometry model, which is a standard used by many other databases and GIS software.

That said, the spatial extension also includes a couple of experimental non-standard explicit geometry types, such as `POINT_2D`, `LINESTRING_2D`, `POLYGON_2D` and `BOX_2D` that are based on DuckDBs native nested types, such as structs and lists. In theory it should be possible to optimize a lot of operations for these types much better than for the `GEOMETRY` type (which is just a binary blob), but only a couple functions are implemented so far.

All of these are implicitly castable to `GEOMETRY` but with a conversion cost, so the `GEOMETRY` type is still the recommended type to use for now if you are planning to work with a lot of different spatial functions.

`GEOMETRY` is not currently capable of storing additional geometry types, Z/M coordinates, or SRID information. These features may be added in the future.

Spatial Scalar Functions

The spatial extension implements a large number of scalar functions and overloads. Most of these are implemented using the GEOS library, but we'd like to implement more of them natively in this extension to better utilize DuckDB's vectorized execution and memory management. The following symbols are used to indicate which implementation is used:

🧭 - GEOS - functions that are implemented using the GEOS library

🦆 - DuckDB - functions that are implemented natively in this extension that are capable of operating directly on the DuckDB types

🔄 - `CAST(GEOMETRY)` - functions that are supported by implicitly casting to `GEOMETRY` and then using the `GEOMETRY` implementation

The currently implemented spatial functions can roughly be categorized into the following groups:

Geometry Conversion Convert between geometries and other formats.

Scalar functions	GEOMETRY	POINT_2D	LINESTRING_2D	POLYGON_2D	BOX_2D
`VARCHAR ST_AsText(GEOMETRY)`	🧭	🪶	🪶	🪶	🔄 (as POLYGON)
`WKB_BLOB ST_AsWKB(GEOMETRY)`	🪶	🪶	🪶	🪶	🪶
`VARCHAR ST_AsHEXWKB(GEOMETRY)`	🪶	🪶	🪶	🪶	🪶
`VARCHAR ST_AsGeoJSON(GEOMETRY)`	🔄	🔄	🔄	🔄	🔄 (as POLYGON)
`GEOMETRY ST_GeomFromText(VARCHAR)`	🧭	🔄	🔄	🔄	🔄 (as POLYGON)
`GEOMETRY ST_GeomFromWKB(BLOB)`	🪶	🪶	🪶	🪶	🔄 (as POLYGON)
`GEOMETRY ST_GeomFromHEXWKB(VARCHAR)`	🪶				
`GEOMETRY ST_GeomFromGeoJSON(VARCHAR)`	🪶				

Geometry Construction Construct new geometries from other geometries or other data.

Scalar functions	GEOMETRY	POINT_2D	LINESTRING_2D	POLYGON_2D	BOX_2D
`GEOMETRY ST_Point(DOUBLE, DOUBLE)`	🪶	🪶			
`GEOMETRY ST_ConvexHull(GEOMETRY)`	🧭	🔄	🔄	🔄	🔄 (as POLYGON)

Scalar functions	GEOMETRY	POINT_2D	LINESTRING_2D	POLYGON_2D	BOX_2D
GEOMETRY ST_Boundary(GEOMETRY)	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
GEOMETRY ST_Buffer(GEOMETRY)	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
GEOMETRY ST_Centroid(GEOMETRY)	🚫	🐌	🐌	🐌	🐌
GEOMETRY ST_Collect(GEOMETRY[])	🐌	🐌	🐌	🐌	🐌
GEOMETRY ST_Normalize(GEOMETRY)	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
GEOMETRY ST_SimplifyPreserveTopology(GEOMETRY, DOUBLE)	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
GEOMETRY ST_Simplify(GEOMETRY, DOUBLE)	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
GEOMETRY ST_Union(GEOMETRY, GEOMETRY)	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
GEOMETRY ST_Intersection(GEOMETRY, GEOMETRY)	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
GEOMETRY ST_MakeLine(GEOMETRY[])	🐌		🐌		
GEOMETRY ST_Envelope(GEOMETRY)	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
GEOMETRY ST_FlipCoordinates(GEOMETRY)	🐌	🐌	🐌	🐌	🐌
GEOMETRY ST_Transform(GEOMETRY, VARCHAR, VARCHAR)	🐌	🐌	🐌	🐌	🐌

Scalar functions	GEOMETRY	POINT_2D	LINESTRING_2D	POLYGON_2D	BOX_2D
`BOX_2D ST_Extent(GEOMETRY)`					
`GEOMETRY ST_PointN(GEOMETRY, INTEGER)`					
`GEOMETRY ST_StartPoint(GEOMETRY)`					
`GEOMETRY ST_EndPoint(GEOMETRY)`					
`GEOMETRY ST_ExteriorRing(GEOMETRY)`					
`GEOMETRY ST_Reverse(GEOMETRY)`					
`GEOMETRY ST_RemoveRepeatedPoints(GEOMETRY)`					(as POLYGON)
`GEOMETRY ST_RemoveRepeatedPoints(GEOMETRY, DOUBLE)`					(as POLYGON)
`GEOMETRY ST_ReducePrecision(GEOMETRY, DOUBLE)`					(as POLYGON)
`GEOMETRY ST_PointOnSurface(GEOMETRY)`					(as POLYGON)
`GEOMETRY ST_CollectionExtract(GEOMETRY)`					
`GEOMETRY ST_CollectionExtract(GEOMETRY, INTEGER)`					

Spatial Properties Calculate and access spatial properties of geometries.

Scalar functions	GEOMETRY	POINT_2D	LINESTRING_2D	POLYGON_2D	BOX_2D
DOUBLE ST_Area(GEOMETRY)	✔	✔	✔	✔	✔
BOOLEAN ST_IsClosed(GEOMETRY)	✘	🔄	🔄	🔄	🔄 (as POLYGON)
BOOLEAN ST_IsEmpty(GEOMETRY)	✔	✔	✔	✔	🔄 (as POLYGON)
BOOLEAN ST_IsRing(GEOMETRY)	✘	🔄	🔄	🔄	🔄 (as POLYGON)
BOOLEAN ST_IsSimple(GEOMETRY)	✘	🔄	🔄	🔄	🔄 (as POLYGON)
BOOLEAN ST_IsValid(GEOMETRY)	✘	🔄	🔄	🔄	🔄 (as POLYGON)
DOUBLE ST_X(GEOMETRY)	✘	✔			
DOUBLE ST_Y(GEOMETRY)	✘	✔			
DOUBLE ST_XMax(GEOMETRY)	✔	✔	✔	✔	✔
DOUBLE ST_YMax(GEOMETRY)	✔	✔	✔	✔	✔
DOUBLE ST_XMin(GEOMETRY)	✔	✔	✔	✔	✔
DOUBLE ST_YMin(GEOMETRY)	✔	✔	✔	✔	✔
GeometryType ST_GeometryType(GEOMETRY)	✔	✔	✔	✔	🔄 (as POLYGON)
DOUBLE ST_Length(GEOMETRY)	✔	✔	✔	✔	🔄 (as POLYGON)
INTEGER ST_NGeometries(GEOMETRY)	✔				

Scalar functions	GEOMETRY	POINT_2D	LINESTRING_2D	POLYGON_2D	BOX_2D
`INTEGER ST_NPoints(GEOMETRY)`	✔	✔	✔	✔	✔
`INTEGER ST_NInteriorRings(GEOMETRY)`	✔			✔	

Spatial Relationships Compute relationships and spatial predicates between geometries.

Scalar functions	GEOMETRY	POINT_2D	LINESTRING_2D	POLYGON_2D	BOX_2D
`BOOLEAN ST_Within(GEOMETRY, GEOMETRY)`	✘	✔ or 🔄	🔄	🔄	🔄 (as POLYGON)
`BOOLEAN ST_Touches(GEOMETRY, GEOMETRY)`	✘	🔄	🔄	🔄	🔄 (as POLYGON)
`BOOLEAN ST_Overlaps(GEOMETRY, GEOMETRY)`	✘	🔄	🔄	🔄	🔄 (as POLYGON)
`BOOLEAN ST_Contains(GEOMETRY, GEOMETRY)`	✘	🔄	🔄	✔ or 🔄	🔄 (as POLYGON)
`BOOLEAN ST_CoveredBy(GEOMETRY, GEOMETRY)`	✘	🔄	🔄	🔄	🔄 (as POLYGON)
`BOOLEAN ST_Covers(GEOMETRY, GEOMETRY)`	✘	🔄	🔄	🔄	🔄 (as POLYGON)
`BOOLEAN ST_Crosses(GEOMETRY, GEOMETRY)`	✘	🔄	🔄	🔄	🔄 (as POLYGON)

Scalar functions	GEOMETRY	POINT_2D	LINESTRING_2D	POLYGON_2D	BOX_2D
`BOOLEAN ST_Difference(GEOMETRY, GEOMETRY)`	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
`BOOLEAN ST_Disjoint(GEOMETRY, GEOMETRY)`	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
`BOOLEAN ST_Intersects(GEOMETRY, GEOMETRY)`	🚫	🔄	🔄	🔄	🐢
`BOOLEAN ST_Equals(GEOMETRY, GEOMETRY)`	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
`DOUBLE ST_Distance(GEOMETRY, GEOMETRY)`	🚫	🐢 or 🔄	🐢 or 🔄	🔄	🔄 (as POLYGON)
`BOOLEAN ST_DWithin(GEOMETRY, GEOMETRY, DOUBLE)`	🚫	🔄	🔄	🔄	🔄 (as POLYGON)
`BOOLEAN ST_Intersects_Extent(GEOMETRY, GEOMETRY)`	🐢	🐢	🐢	🐢	🐢

Spatial Aggregate Functions

Aggregate functions	Implemented with
`GEOMETRY ST_Envelope_Agg(GEOMETRY)`	🐢
`GEOMETRY ST_Union_Agg(GEOMETRY)`	🚫
`GEOMETRY ST_Intersection_Agg(GEOMETRY)`	🚫

Spatial Table Functions

ST_Read() - Read spatial data from files The spatial extension provides a ST_Read table function based on the GDAL translator library to read spatial data from a variety of geospatial vector file formats as if they were DuckDB tables. For example to create a new table from a GeoJSON file, you can use the following query:

```
CREATE TABLE <table> AS SELECT * FROM ST_
↪ Read('some/file/path/filename.json');
```

ST_Read can take a number of optional arguments, the full signature is:

```
ST_Read(VARCHAR, sequential_layer_scan : BOOLEAN, spatial_filter : WKB_BLOB,
↪ open_options : VARCHAR[], layer : VARCHAR, allowed_drivers : VARCHAR[],
↪ sibling_files : VARCHAR[], spatial_filter_box : BOX_2D, keep_wkb :
↪ BOOLEAN)
```

- `sequential_layer_scan` (default: `false`): If set to `true`, the table function will scan through all layers sequentially and return the first layer that matches the given `layer` name. This is required for some drivers to work properly, e.g., the OSM driver.
- `spatial_filter` (default: `NULL`): If set to a WKB blob, the table function will only return rows that intersect with the given WKB geometry. Some drivers may support efficient spatial filtering natively, in which case it will be pushed down. Otherwise the filtering is done by GDAL which may be much slower.
- `open_options` (default: `[]`): A list of key-value pairs that are passed to the GDAL driver to control the opening of the file. E.g., the GeoJSON driver supports a `FLATTEN_NESTED_ATTRIBUTES=YES` option to flatten nested attributes.
- `layer` (default: `NULL`): The name of the layer to read from the file. If `NULL`, the first layer is returned. Can also be a layer index (starting at 0).
- `allowed_drivers` (default: `[]`): A list of GDAL driver names that are allowed to be used to open the file. If empty, all drivers are allowed.
- `sibling_files` (default: `[]`): A list of sibling files that are required to open the file. E.g., the `ESRI Shapefile` driver requires a `.shx` file to be present. Although most of the time these can be discovered automatically.
- `spatial_filter_box` (default: `NULL`): If set to a `BOX_2D`, the table function will only return rows that intersect with the given bounding box. Similar to `spatial_filter`.
- `keep_wkb` (default: `false`): If set, the table function will return geometries in a `wkb_geometry` column with the type `WKB_BLOB` (which can be cast to `BLOB`) instead of `GEOMETRY`. This is useful if you want to use DuckDB with more exotic geometry subtypes that DuckDB spatial doesnt support representing in the `GEOMETRY` type yet.

Note that GDAL is single-threaded, so this table function will not be able to make full use of parallelism.

We're planning to implement support for the most common vector formats natively in this extension with additional table functions in the future.

We currently support over 50 different formats. You can generate the following table of supported GDAL drivers yourself by executing `SELECT * FROM ST_Drivers()`.

short_name	long_name	can_create	can_copy	can_open	help_url
ESRI Shapefile	ESRI Shapefile	true	false	true	https://gdal.org/drivers/vector/shapefile.html
MapInfo File	MapInfo File	true	false	true	https://gdal.org/drivers/vector/mitab.html
UK .NTF	UK .NTF	false	false	true	https://gdal.org/drivers/vector/ntf.html
LVBAG	Kadaster LV BAG Extract 2.0	false	false	true	https://gdal.org/drivers/vector/lvbag.html
S57	IHO S-57 (ENC)	true	false	true	https://gdal.org/drivers/vector/s57.html
DGN	Microstation DGN	true	false	true	https://gdal.org/drivers/vector/dgn.html
OGR_VRT	VRT - Virtual Datasource	false	false	true	https://gdal.org/drivers/vector/vrt.html
Memory	Memory	true	false	true	
CSV	Comma Separated Value (.csv)	true	false	true	https://gdal.org/drivers/vector/csv.html
GML	Geography Markup Language (GML)	true	false	true	https://gdal.org/drivers/vector/gml.html
GPX	GPX	true	false	true	https://gdal.org/drivers/vector/gpx.html
KML	Keyhole Markup Language (KML)	true	false	true	https://gdal.org/drivers/vector/kml.html
GeoJSON	GeoJSON	true	false	true	https://gdal.org/drivers/vector/geojson.html

short_name	long_name	can_create	can_copy	can_open	help_url
GeoJSONSeq	GeoJSON Sequence	true	false	true	https://gdal.org/drivers/vector/geojsonseq.html
ESRIJSON	ESRIJSON	false	false	true	https://gdal.org/drivers/vector/esrijson.html
TopoJSON	TopoJSON	false	false	true	https://gdal.org/drivers/vector/topojson.html
OGR_GMT	GMT ASCII Vectors (.gmt)	true	false	true	https://gdal.org/drivers/vector/gmt.html
GPKG	GeoPackage	true	true	true	https://gdal.org/drivers/vector/gpkg.html
SQLite	SQLite / Spatialite	true	false	true	https://gdal.org/drivers/vector/sqlite.html
WAsP	WAsP .map format	true	false	true	https://gdal.org/drivers/vector/wasp.html
OpenFileGDB	ESRI FileGDB	true	false	true	https://gdal.org/drivers/vector/openfilegdb.html
DXF	AutoCAD DXF	true	false	true	https://gdal.org/drivers/vector/dxf.html
CAD	AutoCAD Driver	false	false	true	https://gdal.org/drivers/vector/cad.html
FlatGeobuf	FlatGeobuf	true	false	true	https://gdal.org/drivers/vector/flatgeobuf.html
Geoconcept	Geoconcept	true	false	true	
GeoRSS	GeoRSS	true	false	true	https://gdal.org/drivers/vector/georss.html
VFK	Czech Cadastral Exchange Data Format	false	false	true	https://gdal.org/drivers/vector/vfk.html

short_name	long_name	can_create	can_copy	can_open	help_url
PGDUMP	PostgreSQL SQL dump	true	false	false	https://gdal.org/drivers/vector/pgdump.html
OSM	OpenStreetMap XML and PBF	false	false	true	https://gdal.org/drivers/vector/osm.html
GPSBabel	GPSBabel	true	false	true	https://gdal.org/drivers/vector/gpsbabel.html
WFS	OGC WFS (Web Feature Service)	false	false	true	https://gdal.org/drivers/vector/wfs.html
OAPIF	OGC API - Features	false	false	true	https://gdal.org/drivers/vector/oapif.html
EDIGEO	French EDIGEO exchange format	false	false	true	https://gdal.org/drivers/vector/edigeo.html
SVG	Scalable Vector Graphics	false	false	true	https://gdal.org/drivers/vector/svg.html
ODS	Open Document/ LibreOffice / OpenOffice Spreadsheet	true	false	true	https://gdal.org/drivers/vector/ods.html
XLSX	MS Office Open XML spreadsheet	true	false	true	https://gdal.org/drivers/vector/xlsx.html
Elasticsearch	Elastic Search	true	false	true	https://gdal.org/drivers/vector/elasticsearch.html
Carto	Carto	true	false	true	https://gdal.org/drivers/vector/carto.html
AmigoCloud	AmigoCloud	true	false	true	https://gdal.org/drivers/vector/amigocloud.html
SXF	Storage and eXchange Format	false	false	true	https://gdal.org/drivers/vector/sxf.html

short_name	long_name	can_create	can_copy	can_open	help_url
Selafin	Selafin	true	false	true	https://gdal.org/drivers/vector/selafin.html
JML	OpenJUMP JML	true	false	true	https://gdal.org/drivers/vector/jml.html
PLSCENES	Planet Labs Scenes API	false	false	true	https://gdal.org/drivers/vector/plscenes.html
CSW	OGC CSW (Catalog Service for the Web)	false	false	true	https://gdal.org/drivers/vector/csw.html
VDV	VDV-451/VDV-452/INTREST Data Format	true	false	true	https://gdal.org/drivers/vector/vdv.html
MVT	Mapbox Vector Tiles	true	false	true	https://gdal.org/drivers/vector/mvt.html
NGW	NextGIS Web	true	true	true	https://gdal.org/drivers/vector/ngw.html
MapML	MapML	true	false	true	https://gdal.org/drivers/vector/mapml.html
TIGER	U.S. Census TIGER/Line	false	false	true	https://gdal.org/drivers/vector/tiger.html
AVCBin	Arc/Info Binary Coverage	false	false	true	https://gdal.org/drivers/vector/avcbin.html
AVCE00	Arc/Info E00 (ASCII) Coverage	false	false	true	https://gdal.org/drivers/vector/avce00.html

Note that far from all of these drivers have been tested properly, and some may require additional options to be passed to work as expected. If you run into any issues please first consult the GDAL docs.

ST_ReadOsm() - Read compressed OSM data The spatial extension also provides an experimental `ST_ReadOsm()` table function to read compressed OSM data directly from a `.osm.pbf` file.

This will use multithreading and zero-copy protobuf parsing which makes it a lot faster than using

the `st_read()` OSM driver, but it only outputs the raw OSM data (Nodes, Ways, Relations), without constructing any geometries. For node entities you can trivially construct `POINT` geometries, but it is also possible to construct `LINESTRING` AND `POLYGON` by manually joining refs and nodes together in SQL.

Example usage:

```
SELECT *
FROM st_readosm('tmp/data/germany.osm.pbf')
WHERE tags['highway'] != []
LIMIT 5;
```

```
┌──────────────────────┬──────────┬──────────────────────┬──────────┬──────────────────────
↪ ┬──────────────┬─────────────┬──────────────────────┐
│         kind         │    id    │         tags         │   refs   │         lat
↪ │     lon      │ ref_roles │       ref_types        │
│ enum('node', 'way'… │  int64   │ map(varchar, varch… │ int64[]  │
↪  double        │   double    │ varchar[] │ enum('node', 'way', … │
├──────────────────────┼──────────┼──────────────────────┼──────────┼──────────────────────
┼──────────────┼─────────────┼──────────────────────┤
│ node                 │ 122351   │ {bicycle=yes, butt… │          │
↪  53.5492951 │   9.977553  │           │                      │
│ node                 │ 122397   │ {crossing=no, high… │          │
↪  53.520990100000006 │ 10.0156924  │           │                      │
│ node                 │ 122493   │ {TMC:cid_58:tabcd_… │          │
↪  53.129614600000004 │  8.1970173  │           │                      │
│ node                 │ 123566   │ {highway=traffic_s… │          │
↪  54.617268200000005 │  8.9718171  │           │                      │
│ node                 │ 125801   │ {TMC:cid_58:tabcd_… │          │
↪  53.070685000000005 │  8.7819939  │           │                      │
└──────────────────────┴──────────┴──────────────────────┴──────────┴──────────────────────
↪ ┴──────────────┴─────────────┴──────────────────────┘
```

Spatial replacement scans

The spatial extension also provides "replacement scans" for common geospatial file formats, allowing you to query files of these formats as if they were tables.

```
SELECT * FROM './path/to/some/shapefile/dataset.shp';
```

In practice this is just syntax-sugar for calling `ST_Read`, so there is no difference in performance. If you want to pass additional options, you should use the `ST_Read` table function directly.

The following formats are currently recognized by their file extension:

- ESRI ShapeFile, `.shp`
- GeoPackage, `.gpkg`

- FlatGeoBuf, `.fgb`

Similarly there is a `.osm.pbf` replacement scan for `ST_ReadOsm`.

Spatial Copy Functions

Much like the `ST_Read` table function the spatial extension provides a GDAL based `COPY` function to export duckdb tables to different geospatial vector formats. For example to export a table to a GeoJSON file, with generated bounding boxes, you can use the following query:

```
COPY <table> TO 'some/file/path/filename.geojson'
WITH (FORMAT GDAL, DRIVER 'GeoJSON', LAYER_CREATION_OPTIONS 'WRITE_
 ↪  BBOX=YES');
```

Available options:

- `FORMAT`: is the only required option and must be set to `GDAL` to use the GDAL based copy function.
- `DRIVER`: is the GDAL driver to use for the export. See the table above for a list of available drivers.
- `LAYER_CREATION_OPTIONS`: list of options to pass to the GDAL driver. See the GDAL docs for the driver you are using for a list of available options.
- `SRS`: Set a spatial reference system as metadata to use for the export. This can be a WKT string, an EPSG code or a proj-string, basically anything you would normally be able to pass to GDAL/OGR. This will not perform any reprojection of the input geometry though, it just sets the metadata if the target driver supports it.

GitHub Repository

GitHub

SQLite Scanner Extension

The `sqlite` extension allows DuckDB to directly read data from a SQLite database file. The data can be queried directly from the underlying SQLite tables, or read into DuckDB tables.

Usage

To make a SQLite file accessible to DuckDB, use the ATTACH statement, which supports read & write, or the older `sqlite_attach` function

For example with the bundled `sakila.db` file:

```
ATTACH 'sakila.db' (TYPE sqlite);
-- or
CALL sqlite_attach('sakila.db');
```

The tables in the file are registered as views in DuckDB, you can list them as follows:

```
PRAGMA show_tables;
```

name
actor
address
category
city
country
customer
customer_list
film
film_actor
film_category
film_list
film_text
inventory
language
payment
rental
sales_by_film_category
sales_by_store
staff
staff_list
store

Then you can query those views normally using SQL, e.g., using the example queries from sakila-examples.sql

```
SELECT
  cat.name category_name,
```

```
  sum(ifnull(pay.amount, 0)) revenue
FROM category cat
LEFT JOIN film_category flm_cat
       ON cat.category_id = flm_cat.category_id
LEFT JOIN film fil
       ON flm_cat.film_id = fil.film_id
LEFT JOIN inventory inv
       ON fil.film_id = inv.film_id
LEFT JOIN rental ren
       ON inv.inventory_id = ren.inventory_id
LEFT JOIN payment pay
       ON ren.rental_id = pay.rental_id
GROUP BY cat.name
ORDER BY revenue DESC
LIMIT 5;
```

Querying Individual Tables

Instead of attaching, you can also query individual tables using the `sqlite_scan` function.

```
SELECT * FROM sqlite_scan('sakila.db', 'film');
```

Data Types

SQLite is a weakly typed database system. As such, when storing data in a SQLite table, types are not enforced. The following is valid SQL in SQLite:

```
CREATE TABLE numbers(i INTEGER);
INSERT INTO numbers VALUES ('hello');
```

DuckDB is a strongly typed database system, as such, it requires all columns to have defined types and the system rigorously checks data for correctness.

When querying SQLite, DuckDB must deduce a specific column type mapping. DuckDB follows SQLite's type affinity rules with a few extensions.

1. If the declared type contains the string ”INT” then it is translated into the type `BIGINT`
2. If the declared type of the column contains any of the strings ”CHAR”, ”CLOB”, or ”TEXT” then it is translated into VARCHAR.
3. If the declared type for a column contains the string ”BLOB” or if no type is specified then it is translated into `BLOB`.

4. If the declared type for a column contains any of the strings ”REAL”, ”FLOA”, ”DOUB”, ”DEC” or ”NUM” then it is translated into DOUBLE.
5. If the declared type is ”DATE”, then it is translated into DATE.
6. If the declared type contains the string ”TIME”, then it is translated into TIMESTAMP.
7. If none of the above apply, then it is translated into VARCHAR.

As DuckDB enforces the corresponding columns to contain only correctly typed values, we cannot load the string ”hello” into a column of type BIGINT. As such, an error is thrown when reading from the ”numbers” table above:

```
Error: Mismatch Type Error: Invalid type in column "i": column was declared
↪  as integer, found "hello" of type "text" instead.
```

This error can be avoided by setting the sqlite_all_varchar option:

```
SET GLOBAL sqlite_all_varchar=true;
```

When set, this option overrides the type conversion rules described above, and instead always converts the SQLite columns into a VARCHAR column. Note that this setting must be set *before* sqlite_attach is called.

Running More Than Once

If you want to run the sqlite_scan procedure more than once in the same DuckDB session, you'll need to pass in the overwrite flag, as shown below:

```
CALL sqlite_attach('sakila.db', overwrite=true);
```

Loading the Extension

The SQLite Scanner extension is by default installed and loaded on first use. If you prefer to do so explicitly, run the following commands:

```
INSTALL sqlite;
LOAD sqlite;
```

GitHub Repository

GitHub

Substrait Extension

The main goal of the `substrait` extension is to support both production and consumption of Substrait query plans in DuckDB.

This extension is mainly exposed via 3 different APIs - the SQL API, the Python API, and the R API. Here we depict how to consume and produce Substrait query plans in each API.

> **Note.** The Substrait integration is currently experimental. Support is currently only available on request. If you have not asked for permission to ask for support, contact us prior to opening an issue. If you open an issue without doing so, we will close it without further review.

Installing and Loading

The Substrait extension is an autoloadable extensions, meaning that it will be loaded at runtime whenever one of the substrait functions is called. To explicitly install and load the released version of the Substrait extension, you can also use the following SQL commands.

```
INSTALL substrait;
LOAD substrait;
```

SQL

In the SQL API, users can generate Substrait plans (into a BLOB or a JSON) and consume Substrait plans.

BLOB Generation To generate a Substrait BLOB the `get_substrait(sql)` function must be called with a valid SQL select query.

```
CREATE TABLE crossfit (exercise text, difficulty_level int);
INSERT INTO crossfit VALUES ('Push Ups', 3), ('Pull Ups', 5) , ('Push Jerk',
 ↪  7), ('Bar Muscle Up', 10);

.mode line
CALL get_substrait('SELECT count(exercise) AS exercise FROM crossfit WHERE
 ↪  difficulty_level <= 5');
```

```
Plan BLOB =
 ↪  \x12\x09\x1A\x07\x10\x01\x1A\x03lte\x12\x11\x1A\x0F\x10\x02\x1A\x0Bis_
 ↪  not_
 ↪  null\x12\x09\x1A\x07\x10\x03\x1A\x03and\x12\x0B\x1A\x09\x10\x04\x1A\x05count\x1A\xC8
 ↪  level\x12\x11\x0A\x07\xB2\x01\x04\x08\x0D\x18\x01\x0A\x04*\x02\x10\x01\x18\x02\x1AJ\
 ↪  \x1A\x1E\x08\x01\x1A\x04*\x02\x10\x01\x22\x0C\x1A\x0A\x12\x08\x0A\x04\x12\x02\x08\x0
```

JSON Generation To generate a JSON representing the Substrait plan the `get_substrait_json(sql)` function must be called with a valid SQL select query.

```
CALL get_substrait_json('SELECT count(exercise) AS exercise FROM crossfit
 ↪  WHERE difficulty_level <= 5');
```

```
Json = {"exten-
 ↪  sions":[{"extensionFunction":{"functionAnchor":1,"name":"lte"}},{"extensionFunction
 ↪  not_
 ↪  null"}},{"extensionFunction":{"functionAnchor":3,"name":"and"}},{"extensionFunction
 ↪  level"],"struct":{"types":[{"varchar":{"length":13,"nullability":"NULLABILITY_
 ↪  NULLABLE"}},{"i32":{"nullability":"NULLABILITY_
 ↪  NULLABLE"}}],"nullability":"NULLABILITY_
 ↪  REQUIRED"}},"filter":{"scalarFunction":{"functionReference":3,"outputType":{"bool":
 ↪  NULLABLE"}},"arguments":[{"value":{"scalarFunction":{"functionReference":1,"outputT
 ↪  NULLABLE"}},"arguments":[{"value":{"selection":{"directReference":{"structField":{"
 ↪  NULLABLE"}},"arguments":[{"value":{"selection":{"directReference":{"structField":{"
 ↪  NULLABLE"}}}}]}},"expressions":[{"selection":{"directReference":{"structField":{}},
```

BLOB Consumption To consume a Substrait BLOB the `from_substrait(blob)` function must be called with a valid Substrait BLOB plan.

```
CALL from_
 ↪  substrait('\x12\x09\x1A\x07\x10\x01\x1A\x03lte\x12\x11\x1A\x0F\x10\x02\x1A\x0Bis_
 ↪  not_
 ↪  null\x12\x09\x1A\x07\x10\x03\x1A\x03and\x12\x0B\x1A\x09\x10\x04\x1A\x05count\x1A\xC8
 ↪  level\x12\x11\x0A\x07\xB2\x01\x04\x08\x0D\x18\x01\x0A\x04*\x02\x10\x01\x18\x02\x1AJ\
 ↪  \x1A\x1E\x08\x01\x1A\x04*\x02\x10\x01\x22\x0C\x1A\x0A\x12\x08\x0A\x04\x12\x02\x08\x0
```

```
exercise = 2
```

Python

Substrait extension is autoloadable, but if you prefer to do so explicitly, you can use the relevant Python syntax within a connection:

```
import duckdb

con = duckdb.connect()
con.install_extension("substrait")
con.load_extension("substrait")
```

BLOB Generation To generate a Substrait BLOB the `get_substrait(sql)` function must be called, from a connection, with a valid SQL select query.

```
con.execute(query='CREATE TABLE crossfit (exercise text, difficulty_level
 ↪  int)')
con.execute(query="INSERT INTO crossfit VALUES ('Push Ups', 3), ('Pull Ups',
 ↪  5) , ('Push Jerk', 7), ('Bar Muscle Up', 10)")

proto_bytes = con.get_substrait(query="SELECT count(exercise) AS exercise
 ↪  FROM crossfit WHERE difficulty_level <= 5").fetchone()[0]
```

Json Generation To generate a JSON representing the Substrait plan the `get_substrait_json(sql)` function, from a connection, must be called with a valid SQL select query.

```
json = con.get_substrait_json("SELECT count(exercise) AS exercise FROM
 ↪  crossfit WHERE difficulty_level <= 5").fetchone()[0]
```

BLOB Consumption To consume a Substrait BLOB the `from_substrait(blob)` function must be called, from the connection, with a valid Substrait BLOB plan.

```
query_result = con.from_substrait(proto=proto_bytes)
```

R

By default the extension will be autoloaded on first use. To explicitly install and load this extension in R, use the following commands:

```
library("duckdb")
con <- dbConnect(duckdb::duckdb())
dbExecute(con, "INSTALL substrait")
dbExecute(con, "LOAD substrait")
```

BLOB Generation To generate a Substrait BLOB the `duckdb_get_substrait(con, sql)` function must be called, with a connection and a valid SQL select query.

```r
dbExecute(con, "CREATE TABLE crossfit (exercise text, difficulty_level
 ↪  int)")
dbExecute(con, "INSERT INTO crossfit VALUES ('Push Ups', 3), ('Pull Ups', 5)
 ↪  , ('Push Jerk', 7), ('Bar Muscle Up', 10)")

proto_bytes <- duckdb::duckdb_get_substrait(con, "SELECT * FROM crossfit
 ↪  LIMIT 5")
```

JSON Generation To generate a JSON representing the Substrait plan `duckdb_get_substrait_json(con, sql)` function, with a connection and a valid SQL select query.

```r
json <- duckdb::duckdb_get_substrait_json(con, "SELECT count(exercise) AS
 ↪  exercise FROM crossfit WHERE difficulty_level <= 5")
```

BLOB Consumption To consume a Substrait BLOB the `duckdb_prepare_substrait(con, blob)` function must be called, with a connection and a valid Substrait BLOB plan.

```r
result <- duckdb::duckdb_prepare_substrait(con, proto_bytes)
df <- dbFetch(result)
```

GitHub Repository

GitHub

TPC-DS Extension

The `tpcds` extension implements the data generator and queries for the TPC-DS benchmark.

Installing and Loading

The `tpcds` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL tpcds;
LOAD tpcds;
```

Usage

To generate data for scale factor 1, use:

```
CALL dsdgen(sf=1);
```

To run a query, e.g., query 8, use:

```
PRAGMA tpcds(8);
```

s_store_name varchar	sum(ss_net_profit) decimal(38,2)
able	-10354620.18
ation	-10576395.52
bar	-10625236.01
ese	-10076698.16
ought	-10994052.78

TPC-H Extension

The `tpch` extension implements the data generator and queries for the TPC-H benchmark.

Installing and Loading

The `tpch` extension is shipped by default in some DuckDB builds, otheriwse it will be transparently autoloaded on first use. If you would like to install and load it manually, run:

```
INSTALL tpch;
LOAD tpch;
```

Usage

To generate data for scale factor 1, use:

```
CALL dbgen(sf=1);
```

To run a query, e.g., query 4, use:

```
PRAGMA tpch(4);
```

o_orderpriority varchar	order_count int64
1-URGENT	21188
2-HIGH	20952
3-MEDIUM	20820
4-NOT SPECIFIED	21112
5-LOW	20974

Guides

Data Import & Export

CSV Import

To read data from a CSV file, use the `read_csv_auto` function in the `FROM` clause of a query.

```
SELECT * FROM read_csv_auto('input.csv');
```

To create a new table using the result from a query, use `CREATE TABLE AS` from a `SELECT` statement.

```
CREATE TABLE new_tbl AS SELECT * FROM read_csv_auto('input.csv');
```

We can use DuckDB's optional FROM-first syntax to omit `SELECT *`:

```
CREATE TABLE new_tbl AS FROM read_csv_auto('input.csv');
```

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement.

```
INSERT INTO tbl SELECT * FROM read_csv_auto('input.csv');
```

Alternatively, the COPY statement can also be used to load data from a CSV file into an existing table.

```
COPY tbl FROM 'input.csv';
```

For additional options, see the CSV Import reference and the COPY statement documentation.

CSV Export

To export the data from a table to a CSV file, use the COPY statement.

```
COPY tbl TO 'output.csv' (HEADER, DELIMITER ',');
```

The result of queries can also be directly exported to a CSV file.

```
COPY (SELECT * FROM tbl) TO 'output.csv' (HEADER, DELIMITER ',');
```

For additional options, see the COPY statement documentation.

Parquet Import

To read data from a Parquet file, use the `read_parquet` function in the `FROM` clause of a query.

```
SELECT * FROM read_parquet('input.parquet');
```

To create a new table using the result from a query, use `CREATE TABLE AS` from a `SELECT` statement.

```
CREATE TABLE new_tbl AS SELECT * FROM read_parquet('input.parquet');
```

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement.

```
INSERT INTO tbl SELECT * FROM read_parquet('input.parquet');
```

Alternatively, the `COPY` statement can also be used to load data from a Parquet file into an existing table.

```
COPY tbl FROM 'input.parquet' (FORMAT PARQUET);
```

For additional options, see the Parquet Loading reference.

Parquet Export

To export the data from a table to a Parquet file, use the `COPY` statement.

```
COPY tbl TO 'output.parquet' (FORMAT PARQUET);
```

The result of queries can also be directly exported to a Parquet file.

```
COPY (SELECT * FROM tbl) TO 'output.parquet' (FORMAT PARQUET);
```

The flags for setting compression, row group size, etc. are listed in the Reading and Writing Parquet files page.

Parquet Import

To run a query directly on a Parquet file, use the `read_parquet` function in the `FROM` clause of a query.

```
SELECT * FROM read_parquet('input.parquet');
```

The Parquet file will be processed in parallel. Filters will be automatically pushed down into the Parquet scan, and only the relevant columns will be read automatically.

For more information see the blog post "Querying Parquet with Precision using DuckDB".

HTTP Parquet Import

To load a Parquet file over HTTP(S), the `httpfs` extension is required. This can be installed use the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL httpfs;
```

To load the `httpfs` extension for usage, use the `LOAD` SQL command:

```
LOAD httpfs;
```

After the `httpfs` extension is set up, Parquet files can be read over `http(s)` using the following command:

```
SELECT * FROM read_parquet('https://<domain>/path/to/file.parquet');
```

S3, GCS, or R2 Parquet Import

To load a Parquet file from S3, the `httpfs` extension is required. This can be installed use the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL httpfs;
```

To load the `httpfs` extension for usage, use the `LOAD` SQL command:

```
LOAD httpfs;
```

After loading the `httpfs` extension, set up the credentials and S3 region to read data. Firstly, the region where the data resides needs to be configured:

```
SET s3_region='us-east-1';
```

With the only the region set, public S3 data can be queried. To query private S3 data, you need to either use an access key and secret:

```
SET s3_access_key_id='<AWS access key id>';
SET s3_secret_access_key='<AWS secret access key>';
```

or a session token:

```
SET s3_session_token='<AWS session token>';
```

After the `httpfs` extension is set up and the S3 configuration is set correctly, Parquet files can be read from S3 using the following command:

```
SELECT * FROM read_parquet('s3://<bucket>/<file>');
```

For Google Cloud Storage (GCS), the Interoperability API enables you to have access to it like an S3 connection. You need to create HMAC keys and declare them:

```
SET s3_endpoint='storage.googleapis.com';
SET s3_access_key_id='key_id';
SET s3_secret_access_key='access_key';
```

Please note you will need to use the `s3://` URL to read your data.

```
SELECT * FROM read_parquet('s3://<gcs_bucket>/<file>');
```

For Cloudflare R2, the S3 Compatibility API allows you to use DuckDB's S3 support to read and write from R2 buckets. You will need to generate an S3 auth token and update the `s3_endpoint` used:

```
SET s3_region="auto"
SET s3_endpoint='<your-account-id>.r2.cloudflarestorage.com';
SET s3_access_key_id='key_id';
SET s3_secret_access_key='access_key';
```

Note that you will need to use the `s3://` URL to read your data from R2:

```
SELECT * FROM read_parquet('s3://<r2_bucket_name>/<file>');
```

S3 Parquet Export

To write a Parquet file to S3, the `httpfs` extension is required. This can be installed use the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL httpfs;
```

To load the `httpfs` extension for usage, use the `LOAD` SQL command:

```
LOAD httpfs;
```

After loading the `httpfs` extension, set up the credentials and S3 region to write data. You may either use an access key and secret, or a token.

```
SET s3_region='us-east-1';
SET s3_access_key_id='<AWS access key id>';
SET s3_secret_access_key='<AWS secret access key>';
```

The alternative is to use a token:

```
SET s3_region='us-east-1';
SET s3_session_token='<AWS session token>';
```

After the `httpfs` extension is set up and the S3 credentials are correctly configured, Parquet files can be written to S3 using the following command:

```
COPY <table_name> TO 's3://bucket/file.parquet';
```

Similarly, Google Cloud Storage (GCS) is supported through the Interoperability API. You need to create HMAC keys and declare them:

```
SET s3_endpoint='storage.googleapis.com';
SET s3_access_key_id='key_id';
SET s3_secret_access_key='access_key';
```

Please note you will need to use the `s3://` URL to write your files.

```
COPY <table_name> TO 's3://gcs_bucket/file.parquet';
```

JSON Import

To read data from a JSON file, use the `read_json_auto` function in the `FROM` clause of a query.

```
SELECT * FROM read_json_auto('input.json');
```

To create a new table using the result from a query, use `CREATE TABLE AS` from a `SELECT` statement.

```
CREATE TABLE new_tbl AS SELECT * FROM read_json_auto('input.json');
```

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement.

```
INSERT INTO tbl SELECT * FROM read_json_auto('input.json');
```

Alternatively, the `COPY` statement can also be used to load data from a JSON file into an existing table.

```
COPY tbl FROM 'input.json';
```

For additional options, see the JSON Loading reference and the COPY statement documentation.

JSON Export

To export the data from a table to a JSON file, use the `COPY` statement.

```
COPY tbl TO 'output.json';
```

The result of queries can also be directly exported to a JSON file.

```
COPY (SELECT * FROM tbl) TO 'output.json';
```

For additional options, see the COPY statement documentation.

Excel Import

To read data from an Excel file, install and load the spatial extension, then use the `st_read` function in the `FROM` clause of a query. Use the `layer` parameter to specify the Excel worksheet name.

```
INSTALL spatial; -- Only needed once per DuckDB connection
LOAD spatial; -- Only needed once per DuckDB connection

SELECT * FROM st_read('test_excel.xlsx', layer='Sheet1');
```

To create a new table using the result from a query, use `CREATE TABLE AS` from a `SELECT` statement.

```
INSTALL spatial; -- Only needed once per DuckDB connection
LOAD spatial; -- Only needed once per DuckDB connection

CREATE TABLE new_tbl AS
SELECT * FROM st_read('test_excel.xlsx', layer='Sheet1');
```

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement.

```
INSTALL spatial; -- Only needed once per DuckDB connection
LOAD spatial; -- Only needed once per DuckDB connection

INSERT INTO tbl
SELECT * FROM st_read('test_excel.xlsx', layer='Sheet1');
```

Several configuration options are also available for the underlying GDAL library that is doing the xlsx parsing. Set those options in an environment variable prior to executing the DuckDB SQL statement. The options include:

- `OGR_XLSX_HEADERS = FORCE / DISABLE / AUTO`
  - Either `FORCE` the first row to be interpreted as headers, `DISABLE` to treat the first row as a row of data, or `AUTO` to detect automatically.
- `OGR_XLSX_FIELD_TYPES = STRING / AUTO`
  - Either `AUTO` detect the data types in the file, or force all data types to be `STRING`.

For additional details, see the spatial extension page, the GDAL XLSX driver page, and the GDAL configuration options page.

Excel Export

To export the data from a table to an Excel file, install and load the spatial extension, then use the COPY statement. The file will contain one worksheet with the same name as the file, but without the .xlsx extension.

```
INSTALL spatial; -- Only needed once per DuckDB connection
LOAD spatial; -- Only needed once per DuckDB connection

COPY tbl TO 'output.xlsx' WITH (FORMAT GDAL, DRIVER 'xlsx');
```

The result of queries can also be directly exported to an Excel file.

```
INSTALL spatial; -- Only needed once per DuckDB connection
LOAD spatial; -- Only needed once per DuckDB connection

COPY (SELECT * FROM tbl) TO 'output.xlsx' WITH (FORMAT GDAL, DRIVER 'xlsx');
```

Note: Dates and timestamps are not supported by the xlsx writer driver. Cast columns of those types to VARCHAR prior to creating the xlsx file.

Note: The output file must not already exist.

For additional details, see the spatial extension page and the GDAL XLSX driver page.

SQLite Import

To run a query directly on a SQLite file, the `sqlite` extension is required. This can be installed use the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL sqlite;
```

To load the `sqlite` extension for usage, use the `LOAD` SQL command:

```
LOAD sqlite;
```

After the SQLite extension is installed, tables can be queried from SQLite using the `sqlite_scan` function:

```
-- scan the table "tbl_name" from the SQLite file "test.db"
SELECT * FROM sqlite_scan('test.db', 'tbl_name');
```

Alternatively, the entire file can be attached using the `sqlite_attach` command. This creates views over all of the tables in the file that allow you to query the tables using regular SQL syntax.

```
-- attach the SQLite file "test.db"
CALL sqlite_attach('test.db');
-- the table "tbl_name" can now be queried as if it is a regular table
SELECT * FROM tbl_name;
```

For more information see the SQLite scanner documentation.

PostgreSQL Import

To run a query directly on a running PostgreSQL database, the `postgres` extension is required. This can be installed use the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL postgres;
```

To load the `postgres` extension for usage, use the `LOAD` SQL command:

```
LOAD postgres;
```

After the `postgres` extension is installed, tables can be queried from PostgreSQL using the `postgres_scan` function:

```
-- scan the table "mytable" from the schema "public" using the empty
 ↪  (default) connection string
SELECT * FROM postgres_scan('', 'public', 'mytable');
```

The first parameter to the `postgres_scan` function is the postgres connection string.

Alternatively, the entire file can be attached using the `postgres_attach` command. This creates views over all of the tables in the PostgreSQL database that allow you to query the tables using regular SQL syntax.

```
-- attach the Postgres database using the given connection string
CALL postgres_attach('');
-- the table "tbl_name" can now be queried as if it is a regular table
SELECT * FROM tbl_name;
```

For more information see the PostgreSQL scanner documentation.

Meta Queries

List Tables

The `SHOW TABLES` command can be used to obtain a list of all tables within the selected schema.

```
CREATE TABLE tbl(i INTEGER);
SHOW TABLES;
```

name
tbl

`DESCRIBE`, `SHOW` or `SHOW ALL TABLES` can be used to obtain a list of all tables within **all** attached databases and schemas.

```
CREATE TABLE tbl(i INTEGER);
CREATE SCHEMA s1;
CREATE TABLE s1.tbl(v VARCHAR);
SHOW ALL TABLES;
```

database	schema	table_name	column_names	column_types	temporary
memory	main	tbl	[i]	[INTEGER]	false
memory	s1	tbl	[v]	[VARCHAR]	false

To view the schema of an individual table, use the `DESCRIBE` command.

```
CREATE TABLE tbl(i INTEGER PRIMARY KEY, j VARCHAR);
DESCRIBE tbl;
```

column_name	column_type	null	key	default	extra
i	INTEGER	NO	PRI	NULL	NULL
j	VARCHAR	YES	NULL	NULL	NULL

The SQL-standard `information_schema` views are also defined.

DuckDB also defines `sqlite_master`, and many PostgreSQL system catalog tables for compatibility with SQLite and PostgreSQL respectively.

Describe

In order to view the schema of the result of a query, prepend `DESCRIBE` to a query.

```
DESCRIBE SELECT * FROM tbl;
```

In order to view the schema of a table, use `DESCRIBE` followed by the table name.

```
DESCRIBE tbl;
```

Below is an example of `DESCRIBE` on the `lineitem` table of TPC-H.

column_name	column_type	null	key	default	extra
l_orderkey	INTEGER	NO	NULL	NULL	NULL
l_partkey	INTEGER	NO	NULL	NULL	NULL
l_suppkey	INTEGER	NO	NULL	NULL	NULL
l_linenumber	INTEGER	NO	NULL	NULL	NULL
l_quantity	INTEGER	NO	NULL	NULL	NULL
l_extendedprice	DECIMAL(15,2)	NO	NULL	NULL	NULL
l_discount	DECIMAL(15,2)	NO	NULL	NULL	NULL
l_tax	DECIMAL(15,2)	NO	NULL	NULL	NULL
l_returnflag	VARCHAR	NO	NULL	NULL	NULL
l_linestatus	VARCHAR	NO	NULL	NULL	NULL
l_shipdate	DATE	NO	NULL	NULL	NULL
l_commitdate	DATE	NO	NULL	NULL	NULL
l_receiptdate	DATE	NO	NULL	NULL	NULL
l_shipinstruct	VARCHAR	NO	NULL	NULL	NULL
l_shipmode	VARCHAR	NO	NULL	NULL	NULL
l_comment	VARCHAR	NO	NULL	NULL	NULL

Summarize

The `SUMMARIZE` command can be used to easily compute a number of aggregates over a table or a query. The `SUMMARIZE` command launches a query that computes a number of aggregates over all columns, including `min`, `max`, `avg`, `std` and `approx_unique`.

In order to summarize the contents of a table, use `SUMMARIZE` followed by the table name.

```
SUMMARIZE tbl;
```

In order to summarize a query, prepend `SUMMARIZE` to a query.

```
SUMMARIZE SELECT * FROM tbl;
```

Below is an example of `SUMMARIZE` on the `lineitem` table of TPC-H `SF1`.

```
┌──────────────────┬──────────────┬──────────┬──────────────────────┬──────────────┬──────────┬──────────┬──────────┬──────────┬──────────┬──────────────────┐
│   column_name    │ column_type  │   min    │         max          │ approx_unique│   avg    │   std    │   q25    │   q50    │   q75    │   count  │ null_percentage  │
├──────────────────┼──────────────┼──────────┼──────────────────────┼──────────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────────────┤
│ l_orderkey       │ INTEGER      │ 1        │ 6000000              │ 1486805      │ 3000279.604204982 │ 1732187.8734803426 │ 1497471  │ 3022276  │ 4523225  │ 6001215  │ 0.0%             │
│ l_partkey        │ INTEGER      │ 1        │ 200000               │ 196125       │ 100017.98932999402 │ 57735.69082650517 │ 50056    │ 99973    │ 150007   │ 6001215  │ 0.0%             │
│ l_suppkey        │ INTEGER      │ 1        │ 10000                │ 10010        │ 5000.602606138924 │ 2886.9619987306205 │ 2499     │ 5001     │ 7498     │ 6001215  │ 0.0%             │
│ l_linenumber     │ INTEGER      │ 1        │ 7                    │ 7            │ 3.0005757167506912 │ 1.7324314036519335 │ 1        │ 3        │ 4        │ 6001215  │ 0.0%             │
│ l_quantity       │ INTEGER      │ 1        │ 50                   │ 50           │ 25.507967136654827 │ 14.426262537016953 │ 12       │ 25       │ 37       │ 6001215  │ 0.0%             │
│ l_extendedprice  │ DECIMAL(15,2)│ 901.00   │ 104949.50            │ 939196       │ 38255.138484656854 │ 23300.438710962204 │ 18747    │ 36719    │ 55141    │ 6001215  │ 0.0%             │
│ l_discount       │ DECIMAL(15,2)│ 0.00     │ 0.10                 │ 11           │ 0.04999943011540163 │ 0.031619855108125976 │ 0        │ 0        │ 0        │ 6001215  │ 0.0%             │
│ l_tax            │ DECIMAL(15,2)│ 0.00     │ 0.08                 │ 9            │ 0.04001350893110812 │ 0.02581655179884275 │ 0        │ 0        │ 0        │ 6001215  │ 0.0%             │
```

```
│ l_returnflag    │ VARCHAR       │ A           │ R
↪  │ 3             │ NULL                │ NULL                  │ NULL    │
↪  NULL    │ NULL    │ 6001215 │ 0.0%            │
│ l_linestatus    │ VARCHAR       │ F           │ O
↪  │ 2             │ NULL                │ NULL                  │ NULL    │
↪  NULL    │ NULL    │ 6001215 │ 0.0%            │
│ l_shipdate      │ DATE          │ 1992-01-02  │ 1998-12-01
↪  │ 2554          │ NULL                │ NULL                  │ NULL    │
↪  NULL    │ NULL    │ 6001215 │ 0.0%            │
│ l_commitdate    │ DATE          │ 1992-01-31  │ 1998-10-31
↪  │ 2491          │ NULL                │ NULL                  │ NULL    │
↪  NULL    │ NULL    │ 6001215 │ 0.0%            │
│ l_receiptdate   │ DATE          │ 1992-01-04  │ 1998-12-31
↪  │ 2585          │ NULL                │ NULL                  │ NULL    │
↪  NULL    │ NULL    │ 6001215 │ 0.0%            │
│ l_shipinstruct  │ VARCHAR       │ COLLECT COD │ TAKE BACK RETURN
↪  │ 4             │ NULL                │ NULL                  │ NULL    │
↪  NULL    │ NULL    │ 6001215 │ 0.0%            │
│ l_shipmode      │ VARCHAR       │ AIR         │ TRUCK
↪  │ 7             │ NULL                │ NULL                  │ NULL    │
↪  NULL    │ NULL    │ 6001215 │ 0.0%            │
│ l_comment       │ VARCHAR       │  Tiresias   │ zzle? slyly final platelets
↪  sleep quickly.  │ 4587836       │ NULL                │ NULL
↪  │ NULL    │ NULL    │ NULL    │ 6001215 │ 0.0%            │
└─────────────────┴───────────────┴─────────────┴──────────────────────────────────────────────────────────────────────────────
```

Explain

In order to view the query plan of a query, prepend `EXPLAIN` to a query.

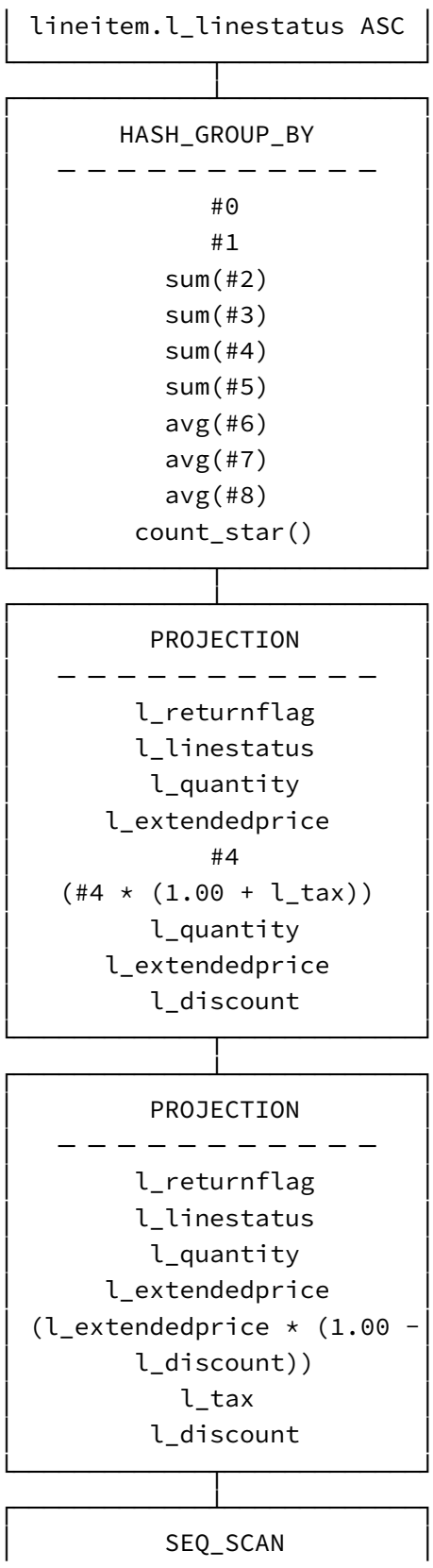
```
EXPLAIN SELECT * FROM tbl;
```

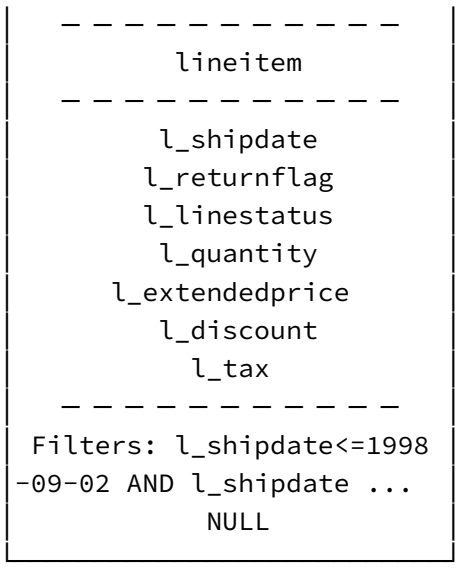
By default only the final physical plan is shown. In order to see the unoptimized and optimized logical plans, change the `explain_output` setting:

```
SET explain_output='all';
```

Below is an example of running `EXPLAIN` on `Q1` of the TPC-H benchmark.

```
┌──────────────────────────┐
│         ORDER_BY         │
│   — — — — — — — — — —    │
│ lineitem.l_returnflag ASC │
```


lineitem.l_linestatus ASC
HASH_GROUP_BY
#0
#1
sum(#2)
sum(#3)
sum(#4)
sum(#5)
avg(#6)
avg(#7)
avg(#8)
count_star()
PROJECTION
l_returnflag
l_linestatus
l_quantity
l_extendedprice
#4
(#4 * (1.00 + l_tax))
l_quantity
l_extendedprice
l_discount
PROJECTION
l_returnflag
l_linestatus
l_quantity
l_extendedprice
(l_extendedprice * (1.00 -
l_discount))
l_tax
l_discount
SEQ_SCAN



Profile Queries

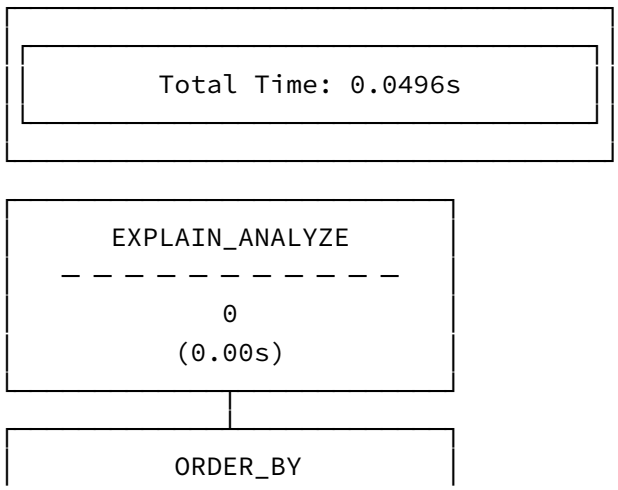
In order to profile a query, prepend `EXPLAIN ANALYZE` to a query.

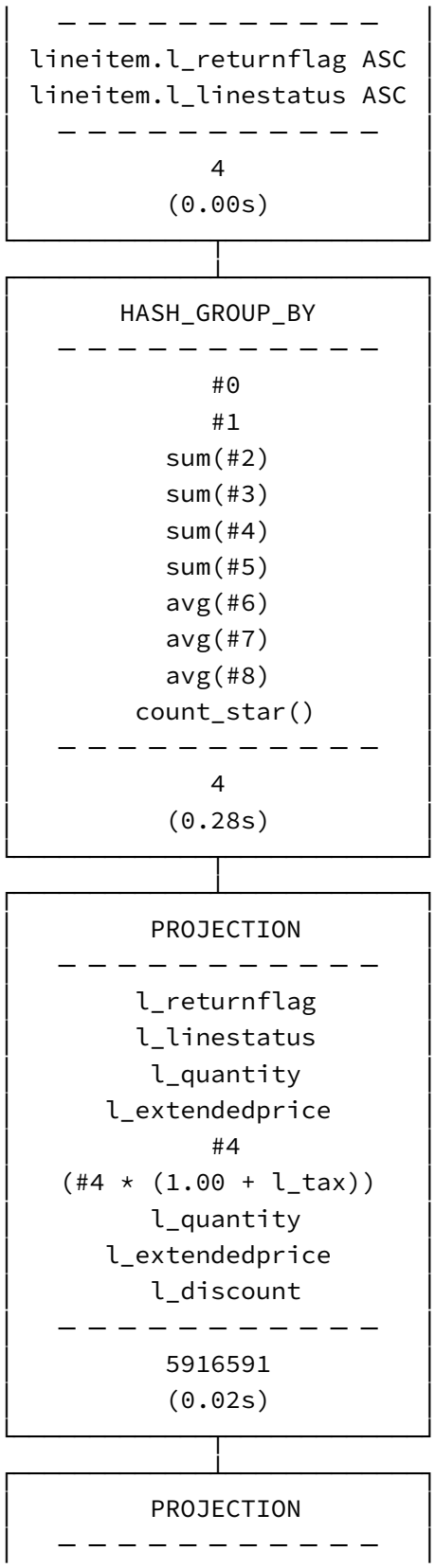
```
EXPLAIN ANALYZE SELECT * FROM tbl;
```

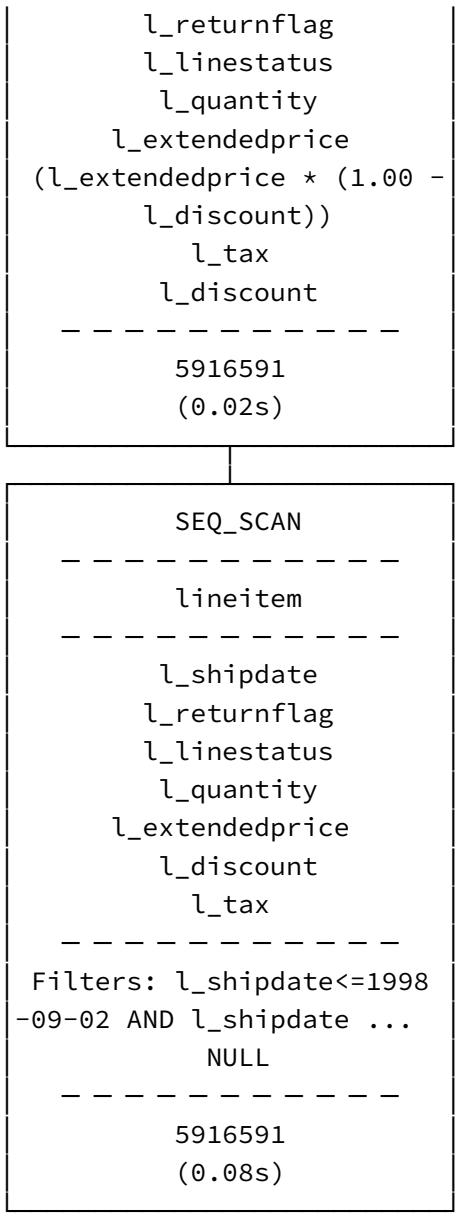
The query plan will be pretty-printed to the screen using timings for every operator.

Note that the **cumulative** wall-clock time that is spent on every operator is shown. When multiple threads are processing the query in parallel, the total processing time of the query may be lower than the sum of all the times spent on the individual operators.

Below is an example of running `EXPLAIN ANALYZE` on `Q1` of the TPC-H benchmark.




lineitem.l_returnflag ASC
lineitem.l_linestatus ASC
4
(0.00s)
HASH_GROUP_BY
#0
#1
sum(#2)
sum(#3)
sum(#4)
sum(#5)
avg(#6)
avg(#7)
avg(#8)
count_star()
4
(0.28s)
PROJECTION
l_returnflag
l_linestatus
l_quantity
l_extendedprice
#4
(#4 * (1.00 + l_tax))
l_quantity
l_extendedprice
l_discount
5916591
(0.02s)
PROJECTION


l_returnflag
l_linestatus
l_quantity
l_extendedprice
(l_extendedprice * (1.00 -
l_discount))
l_tax
l_discount
5916591
(0.02s)
SEQ_SCAN
lineitem
l_shipdate
l_returnflag
l_linestatus
l_quantity
l_extendedprice
l_discount
l_tax
Filters: l_shipdate<=1998
-09-02 AND l_shipdate ...
NULL
5916591
(0.08s)

ODBC

ODBC 101: A Duck Themed Guide to ODBC

- What is ODBC?
- General Concepts
- Setting up an Application
- Sample Application

What is ODBC?

ODBC which stands for Open Database Connectivity, is a standard that allows different programs to talk to different databases including, of course, **DuckDB** 🦆. This makes it easier to build programs that work with many different databases, which saves time as developers don't have to write custom code to connect to each database. Instead, they can use the standardized ODBC interface, which reduces development time and costs, and programs are easier to maintain. However, ODBC can be slower than other methods of connecting to a database, such as using a native driver, as it adds an extra layer of abstraction between the application and the database. Furthermore, because DuckDB is column-based and ODBC is row-based, there can be some inefficiencies when using ODBC with DuckDB.

> **Note.** There are links throughout this page to the official Microsoft ODBC documentation, which is a great resource for learning more about ODBC.

General Concepts

- Handles
- Connecting
- Error Handling and Diagnostics
- Buffers and Binding

Handles A handle is a pointer to a specific ODBC object which is used to interact with the database. There are several different types of handles, each with a different purpose, these are the environment handle, the connection handle, the statement handle, and the descriptor handle. Handles are allocated using the `SQLAllocHandle` which takes as input the type of handle to allocate, and a pointer to the handle, the driver then creates a new handle of the specified type which it returns to the application.

Handle Types

Handle	Type	Description	Use Case	Additional Information
Environment	SQL_HANDLE_ENV	Manages the environment settings for ODBC operations, and provides a global context in which to access data.	Initializing ODBC, managing driver behavior, resource allocation	Must be allocated once per application upon starting, and freed at the end.
Connection	SQL_HANDLE_DBC	Represents a connection to a data source. Used to establish, manage, and terminate connections. Defines both the driver and the data source to use within the driver.	Establishing a connection to a database, managing the connection state	Multiple connection handles can be created as needed, allowing simultaneous connections to multiple data sources. *Note:* Allocating a connection handle does not establish a connection, but must be allocated first, and then used once the connection has been established.

Handle	Type	Description	Use Case	Additional Information
Statement	SQL_HANDLE_STMT	Handles the execution of SQL statements, as well as the returned result sets.	Executing SQL queries, fetching result sets, managing statement options.	To facilitate the execution of concurrent queries, multiple handles can be allocated per connection.
Descriptor	SQL_HANDLE_DESC	Describes the attributes of a data structure or parameter, and allows the application to specify the structure of data to be bound/retrieved.	Describing table structures, result sets, binding columns to application buffers	Used in situations where data structures need to be explicitly defined, for example during parameter binding or result set fetching. They are automatically allocated when a statement is allocated, but can also be allocated explicitly.

Connecting The first step is to connect to the data source so that the application can perform database operations. First the application must allocate an environment handle, and then a connection handle. The connection handle is then used to connect to the data source. There are two functions which can be used to connect to a data source, `SQLDriverConnect` and `SQLConnect`. The former is used to connect to a data source using a connection string, while the latter is used to connect to a data source using a DSN.

Connection String A connection string is a string which contains the information needed to connect to a data source. It is formatted as a semicolon separated list of key-value pairs, however DuckDB currently only utilizes the DSN and ignores the rest of the parameters.

DSN A DSN (*Data Source Name*) is a string that identifies a database. It can be a file path, URL, or a database name. For example: `C:\Users\me\duckdb.db` and `DuckDB` are both valid DSNs. More information on DSNs can be found here.

Error Handling and Diagnostics All functions in ODBC return a code which represents the success or failure of the function. This allows for easy error handling, as the application can simply check the return code of each function call to determine if it was successful. When unsuccessful, the application can then use the `SQLGetDiagRec` function to retrieve the error information. The following table defines the return codes:

Return Code	Description
`SQL_SUCCESS`	The function completed successfully.
`SQL_SUCCESS_WITH_INFO`	The function completed successfully, but additional information is available, including a warning
`SQL_ERROR`	The function failed.
`SQL_INVALID_HANDLE`	The handle provided was invalid, indicating a programming error, i.e., when a handle is not allocated before it is used, or is the wrong type
`SQL_NO_DATA`	The function completed successfully, but no more data is available
`SQL_NEED_DATA`	More data is needed, such as when a parameter data is sent at execution time, or additional connection information is required.
`SQL_STILL_EXECUTING`	A function that was asynchronously executed is still executing.

Buffers and Binding A buffer is a block of memory used to store data. Buffers are used to store data retrieved from the database, or to send data to the database. Buffers are allocated by the application, and then bound to a column in a result set, or a parameter in a query, using the `SQLBindCol` and `SQLBindParameter` functions. When the application fetches a row from the result set, or executes a query, the data is stored in the buffer. When the application sends a query to the database, the data in the buffer is sent to the database.

Setting up an Application

The following is a step-by-step guide to setting up an application that uses ODBC to connect to a database, execute a query, and fetch the results in `C++`.

> **Note.** To install the driver as well as anything else you will need follow these instructions.

1. Include the SQL Header Files
2. Define the ODBC Handles and Connect to the Database
3. Adding a Query
4. Fetching Results
5. Go Wild
6. Free the Handles and Disconnecting

1. Include the SQL Header Files The first step is to include the SQL header files:

```
#include <sql.h>
#include <sqlext.h>
```

These files contain the definitions of the ODBC functions, as well as the data types used by ODBC. In order to be able to use these header files you have to have the `unixodbc` package installed:

```
brew install unixodbc
# or
sudo apt-get install unixodbc-dev
# or
sudo yum install unixODBC-devel
```

Remember to include the header file location in your `CFLAGS`.

For `MAKEFILE`:

```
CFLAGS=-I/usr/local/include
# or
CFLAGS=-/opt/homebrew/Cellar/unixodbc/2.3.11/include
```

For `CMAKE`:

```
include_directories(/usr/local/include)
# or
include_directories(/opt/homebrew/Cellar/unixodbc/2.3.11/include)
```

You also have to link the library in your `CMAKE` or `MAKEFILE`: For `CMAKE`:

```
target_link_libraries(ODBC_application /path/to/duckdb_odbc/libduckdb_
↪  odbc.dylib)
```

For `MAKEFILE`:

```
LDLIBS=-L/path/to/duckdb_odbc/libduckdb_odbc.dylib
```

2. Define the ODBC Handles and Connect to the Database Then set up the ODBC handles, allocate them, and connect to the database. First the environment handle is allocated, then the environment is set to ODBC version 3, then the connection handle is allocated, and finally the connection is made to the database. The following code snippet shows how to do this:

```
SQLHANDLE env;
SQLHANDLE dbc;

SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);

SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);

SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);

std::string dsn = "DSN=duckdbmemory";
SQLConnect(dbc, (SQLCHAR*)dsn.c_str(), SQL_NTS, NULL, 0, NULL, 0);

std::cout << "Connected!" << std::endl;
```

3. Adding a Query Now that the application is set up, we can add a query to it. First, we need to allocate a statement handle:

```
SQLHANDLE stmt;
SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
```

Then we can execute a query:

```
SQLExecDirect(stmt, (SQLCHAR*)"SELECT * FROM integers", SQL_NTS);
```

4. Fetching Results Now that we have executed a query, we can fetch the results. First, we need to bind the columns in the result set to buffers:

```
SQLLEN int_val;
SQLLEN null_val;
SQLBindCol(stmt, 1, SQL_C_SLONG, &int_val, 0, &null_val);
```

Then we can fetch the results:

```
SQLFetch(stmt);
```

5. Go Wild Now that we have the results, we can do whatever we want with them. For example, we can print them:

```
std::cout << "Value: " << int_val << std::endl;
```

or do any other processing we want. As well as executing more queries and doing any thing else we want to do with the database such as inserting, updating, or deleting data.

6. Free the Handles and Disconnecting Finally, we need to free the handles and disconnect from the database. First, we need to free the statement handle:

```
SQLFreeHandle(SQL_HANDLE_STMT, stmt);
```

Then we need to disconnect from the database:

```
SQLDisconnect(dbc);
```

And finally, we need to free the connection handle and the environment handle:

```
SQLFreeHandle(SQL_HANDLE_DBC, dbc);
SQLFreeHandle(SQL_HANDLE_ENV, env);
```

Freeing the connection and environment handles can only be done after the connection to the database has been closed. Trying to free them before disconnecting from the database will result in an error.

Sample Application

The following is a sample application that includes a `cpp` file that connects to the database, executes a query, fetches the results, and prints them. It also disconnects from the database and frees the handles, and includes a function to check the return value of ODBC functions. It also includes a `CMakeLists.txt` file that can be used to build the application.

Sample `.cpp` file

```
#include <iostream>
#include <sql.h>
#include <sqlext.h>

void check_ret(SQLRETURN ret, std::string msg) {
    if (ret != SQL_SUCCESS && ret != SQL_SUCCESS_WITH_INFO) {
        std::cout << ret << ": " << msg << " failed" << std::endl;
        exit(1);
    }
    if (ret == SQL_SUCCESS_WITH_INFO) {
```

```
        std::cout << ret << ": " << msg << " succeeded with info" <<
        ↪  std::endl;
    }
}

int main() {
    SQLHANDLE env;
    SQLHANDLE dbc;
    SQLRETURN ret;

    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
    check_ret(ret, "SQLAllocHandle(env)");

    ret = SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
    check_ret(ret, "SQLSetEnvAttr");

    ret = SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);
    check_ret(ret, "SQLAllocHandle(dbc)");

    std::string dsn = "DSN=duckdbmemory";
    ret = SQLConnect(dbc, (SQLCHAR*)dsn.c_str(), SQL_NTS, NULL, 0, NULL, 0);
    check_ret(ret, "SQLConnect");

    std::cout << "Connected!" << std::endl;

    SQLHANDLE stmt;
    ret = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
    check_ret(ret, "SQLAllocHandle(stmt)");

    ret = SQLExecDirect(stmt, (SQLCHAR*)"SELECT * FROM integers", SQL_NTS);
    check_ret(ret, "SQLExecDirect(SELECT * FROM integers)");

    SQLLEN int_val;
    SQLLEN null_val;
    ret = SQLBindCol(stmt, 1, SQL_C_SLONG, &int_val, 0, &null_val);
    check_ret(ret, "SQLBindCol");

    ret = SQLFetch(stmt);
    check_ret(ret, "SQLFetch");

    std::cout << "Value: " << int_val << std::endl;

    ret = SQLFreeHandle(SQL_HANDLE_STMT, stmt);
```

```
    check_ret(ret, "SQLFreeHandle(stmt)");

    ret = SQLDisconnect(dbc);
    check_ret(ret, "SQLDisconnect");

    ret = SQLFreeHandle(SQL_HANDLE_DBC, dbc);
    check_ret(ret, "SQLFreeHandle(dbc)");

    ret = SQLFreeHandle(SQL_HANDLE_ENV, env);
    check_ret(ret, "SQLFreeHandle(env)");
}
```

Sample `CMakelists.txt` file

```
cmake_minimum_required(VERSION 3.25)
project(ODBC_Tester_App)

set(CMAKE_CXX_STANDARD 17)
include_directories(/opt/homebrew/Cellar/unixodbc/2.3.11/include)

add_executable(ODBC_Tester_App main.cpp)
target_link_libraries(ODBC_Tester_App /duckdb_odbc/libduckdb_odbc.dylib)
```

Python

Install the Python Client

The latest release of the Python client can be installed using `pip`.

```
pip install duckdb
```

The pre-release Python client can be installed using `--pre`.

```
pip install duckdb --upgrade --pre
```

The latest Python client can be installed from source from the `tools/pythonpkg` directory in the DuckDB GitHub repository.

```
BUILD_PYTHON=1 GEN=ninja make
cd tools/pythonpkg
python setup.py install
```

Execute SQL

SQL queries can be executed using the `duckdb.sql` command.

```
import duckdb
duckdb.sql("SELECT 42").show()
```

By default this will create a relation object. The result can be converted to various formats using the result conversion functions. For example, the `fetchall` method can be used to convert the result to Python objects.

```
results = duckdb.sql("SELECT 42").fetchall()
print(results)
# [(42,)]
```

Several other result objects exist. For example, you can use `df` to convert the result to a Pandas DataFrame.

```
results = duckdb.sql("SELECT 42").df()
print(results)
#    42
# 0  42
```

By default, a global in-memory connection will be used. Any data stored in files will be lost after shutting down the program. A connection to a persistent database can be created using the `connect` function.

After connecting, SQL queries can be executed using the `sql` command.

```
con = duckdb.connect('file.db')
con.sql('CREATE TABLE integers(i INTEGER)')
con.sql('INSERT INTO integers VALUES (42)')
con.sql('SELECT * FROM integers').show()
```

Jupyter Notebooks

DuckDB's Python client can be used directly in Jupyter notebooks with no additional configuration if desired. However, additional libraries can be used to simplify SQL query development. This guide will describe how to utilize those additional libraries. See other guides in the Python section for how to use DuckDB and Python together.

In this example, we use the JupySQL package.

This example workflow is also available as a Google Colab notebook.

Library Installation

Four additional libraries improve the DuckDB experience in Jupyter notebooks.

1. jupysql
    - Convert a Jupyter code cell into a SQL cell
2. Pandas
    - Clean table visualizations and compatibility with other analysis
3. matplotlib
    - Plotting with Python
4. duckdb-engine (DuckDB SQLAlchemy driver)

- Used by SQLAlchemy to connect to DuckDB (optional)

```
# Run these pip install commands from the command line if Jupyter Notebook
 ↪  is not yet installed.
# Otherwise, see Google Collab link above for an in-notebook example
pip install duckdb

# Install Jupyter Notebook (Note: you can also install JupyterLab: pip
 ↪  install jupyterlab)
pip install notebook

# Install supporting libraries
pip install jupysql
pip install pandas
pip install matplotlib
pip install duckdb-engine
```

Library Import and Configuration

Open a Jupyter Notebook and import the relevant libraries.

Connecting to DuckDB Natively To connect to DuckDB, run:

```
import duckdb
import pandas as pd

%load_ext sql
conn = duckdb.connect()
%sql conn --alias duckdb
```

Connecting to DuckDB via SQLAlchemy Using `duckdb_engine` Alternatively, you can connect to DuckDB via SQLAlchemy using `duckdb_engine`. See the performance and feature differences.

```
import duckdb
import pandas as pd
# No need to import duckdb_engine
#  jupysql will auto-detect the driver needed based on the connection string!

# Import jupysql Jupyter extension to create SQL cells
%load_ext sql
```

Set configurations on jupysql to directly output data to Pandas and to simplify the output that is printed to the notebook.

```
%config SqlMagic.autopandas = True
%config SqlMagic.feedback = False
%config SqlMagic.displaycon = False
```

Connect jupysql to DuckDB using a SQLAlchemy-style connection string. Either connect to a new in-memory DuckDB, the default connection or a file backed db.

```
%sql duckdb:///:default:
# %sql duckdb:///:memory:
# %sql duckdb:///path/to/file.db
```

> **Note.** The `%sql` command and `duckdb.sql` share the same default connection if you provide `duckdb:///:default:` as the SQLAlchemy connection string.

Querying DuckDB

Single line SQL queries can be run using `%sql` at the start of a line. Query results will be displayed as a Pandas DF.

```
%sql SELECT 'Off and flying!' AS a_duckdb_column
```

An entire Jupyter cell can be used as a SQL cell by placing `%%sql` at the start of the cell. Query results will be displayed as a Pandas DF.

```
%%sql
SELECT
    schema_name,
    function_name
FROM duckdb_functions()
ORDER BY ALL DESC
LIMIT 5
```

To store the query results in a Python variable, use `<<` as an assignment operator. This can be used with both the `%sql` and `%%sql` Jupyter magics.

```
%sql res << SELECT 'Off and flying!' AS a_duckdb_column
```

If the `%config SqlMagic.autopandas = True` option is set, the variable is a Pandas dataframe, otherwise, it is a `ResultSet` that can be converted to Pandas with the `DataFrame()` function.

Querying Pandas Dataframes

DuckDB is able to find and query any dataframe stored as a variable in the Jupyter notebook.

```
input_df = pd.DataFrame.from_dict({"i":[1, 2, 3],
                                   "j":["one", "two", "three"]})
```

The dataframe being queried can be specified just like any other table in the FROM clause.

```
%sql output_df << SELECT sum(i) AS total_i FROM input_df
```

Visualizing DuckDB Data

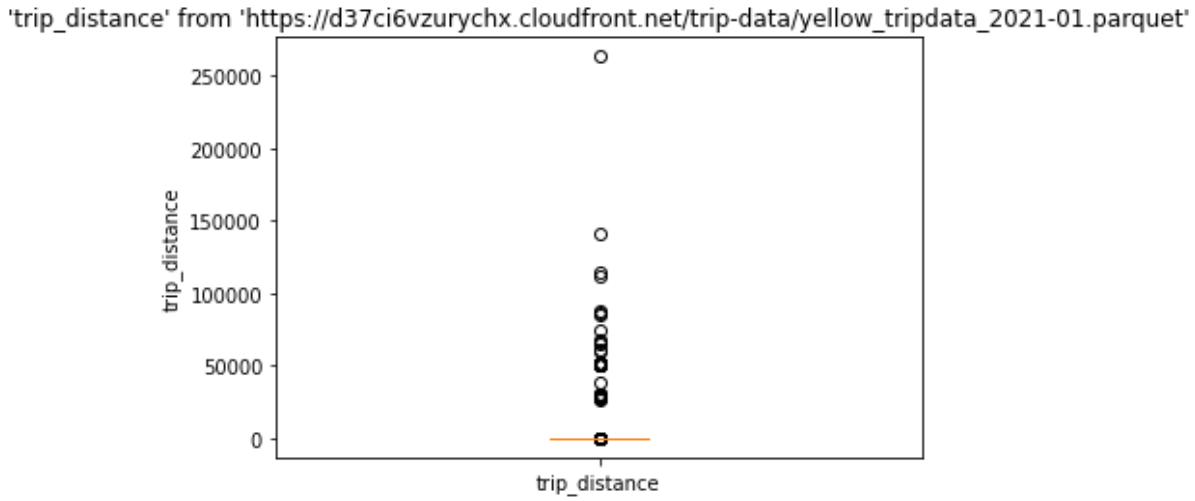
The most common way to plot datasets in Python is to load them using Pandas and then use matplotlib or seaborn for plotting. This approach requires loading all data into memory which is highly inefficient. The plotting module in JupySQL runs computations in the SQL engine. This delegates memory management to the engine and ensures that intermediate computations do not keep eating up memory, efficiently plotting massive datasets.

Install and Load DuckDB httpfs extension DuckDB's httpfs extension allows parquet and CSV files to be queried remotely over http. These examples query a parquet file that contains historical taxi data from NYC. Using the parquet format allows DuckDB to only pull the rows and columns into memory that are needed rather than downloading the entire file. DuckDB can be used to process local parquet files as well, which may be desirable if querying the entire parquet file, or running multiple queries that require large subsets of the file.

```
%%sql
INSTALL httpfs;
LOAD httpfs;
```

Boxplot & Histogram To create a boxplot, call `%sqlplot boxplot`, passing the name of the table and the column to plot. In this case, the name of the table is the URL of the remotely stored parquet file.

```
%sqlplot boxplot --table
 ↪  https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_
 ↪  2021-01.parquet --column trip_distance
```

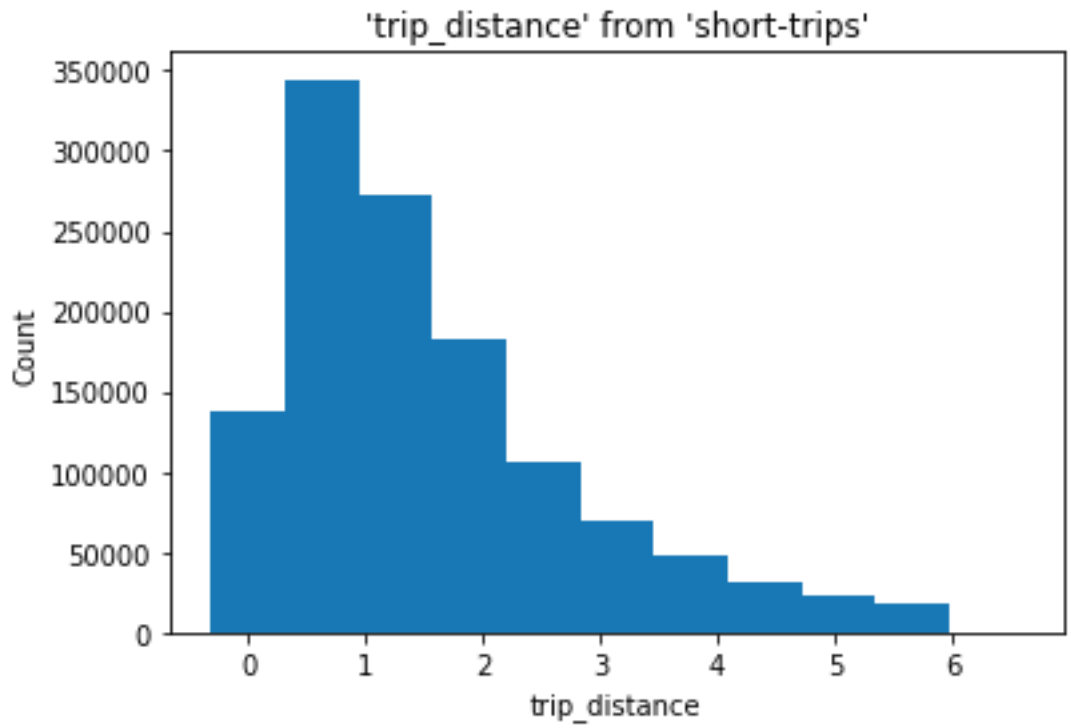


Now, create a query that filters by the 90th percentile. Note the use of the `--save`, and `--no-execute` functions. This tells JupySQL to store the query, but skips execution. It will be referenced in the next plotting call.

```
%%sql --save short-trips --no-execute
SELECT *
FROM "https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_
 ↪  2021-01.parquet"
WHERE trip_distance < 6.3
```

To create a histogram, call `%sqlplot histogram` and pass the name of the table, the column to plot, and the number of bins. This uses `--with short-trips` so JupySQL uses the query defined previously and therefore only plots a subset of the data.

```
%sqlplot histogram --table short-trips --column trip_distance --bins 10
 ↪  --with short-trips
```



Summary

You now have the ability to alternate between SQL and Pandas in a simple and highly performant way! You can plot massive datasets directly through the engine (avoiding both the download of the entire file and loading all of it into Pandas in memory). Dataframes can be read as tables in SQL, and SQL results can be output into Dataframes. Happy analyzing!

SQL on Pandas

Pandas DataFrames stored in local variables can be queried as if they are regular tables within DuckDB.

```
import duckdb
import pandas

# Create a Pandas dataframe
my_df = pandas.DataFrame.from_dict({'a': [42]})

# query the Pandas DataFrame "my_df"
```

```
# Note: duckdb.sql connects to the default in-memory database connection
results = duckdb.sql("SELECT * FROM my_df").df()
```

Import from Pandas

`CREATE TABLE AS` and `INSERT INTO` can be used to create a table from any query. We can then create tables or insert into existing tables by referring to referring to the Pandas DataFrame in the query.

```
import duckdb
import pandas

# Create a Pandas dataframe
my_df = pandas.DataFrame.from_dict({'a': [42]})

# create the table "my_table" from the DataFrame "my_df"
# Note: duckdb.sql connects to the default in-memory database connection
duckdb.sql("CREATE TABLE my_table AS SELECT * FROM my_df")

# insert into the table "my_table" from the DataFrame "my_df"
duckdb.sql("INSERT INTO my_table SELECT * FROM my_df")
```

Export to Pandas

The result of a query can be converted to a Pandas DataFrame using the `df()` function.

```
import duckdb

# read the result of an arbitrary SQL query to a Pandas DataFrame
results = duckdb.sql("SELECT 42").df()
```

SQL on Apache Arrow

DuckDB can query multiple different types of Apache Arrow objects.

Apache Arrow Tables

Arrow Tables stored in local variables can be queried as if they are regular tables within DuckDB.

```
import duckdb
import pyarrow as pa

# connect to an in-memory database
con = duckdb.connect()

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# query the Apache Arrow Table "my_arrow_table" and return as an Arrow Table
results = con.execute("SELECT * FROM my_arrow_table WHERE i = 2").arrow()
```

Apache Arrow Datasets

Arrow Datasets stored as variables can also be queried as if they were regular tables. Datasets are useful to point towards directories of Parquet files to analyze large datasets. DuckDB will push column selections and row filters down into the dataset scan operation so that only the necessary data is pulled into memory.

```
import duckdb
import pyarrow as pa
import tempfile
import pathlib
import pyarrow.parquet as pq
import pyarrow.dataset as ds

# connect to an in-memory database
con = duckdb.connect()

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# create example parquet files and save in a folder
base_path = pathlib.Path(tempfile.gettempdir())
(base_path / "parquet_folder").mkdir(exist_ok=True)
pq.write_to_dataset(my_arrow_table, str(base_path / "parquet_folder"))

# link to parquet files using an Arrow Dataset
my_arrow_dataset = ds.dataset(str(base_path / 'parquet_folder/'))

# query the Apache Arrow Dataset "my_arrow_dataset" and return as an Arrow
 ↪  Table
```

```
results = con.execute("SELECT * FROM my_arrow_dataset WHERE i = 2").arrow()
```

Apache Arrow Scanners

Arrow Scanners stored as variables can also be queried as if they were regular tables. Scanners read over a dataset and select specific columns or apply row-wise filtering. This is similar to how DuckDB pushes column selections and filters down into an Arrow Dataset, but using Arrow compute operations instead. Arrow can use asynchronous IO to quickly access files.

```
import duckdb
import pyarrow as pa
import tempfile
import pathlib
import pyarrow.parquet as pq
import pyarrow.dataset as ds
import pyarrow.compute as pc

# connect to an in-memory database
con = duckdb.connect()

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# create example parquet files and save in a folder
base_path = pathlib.Path(tempfile.gettempdir())
(base_path / "parquet_folder").mkdir(exist_ok=True)
pq.write_to_dataset(my_arrow_table, str(base_path / "parquet_folder"))

# link to parquet files using an Arrow Dataset
my_arrow_dataset = ds.dataset(str(base_path / 'parquet_folder/'))

# define the filter to be applied while scanning
# equivalent to "WHERE i = 2"
scanner_filter = (pc.field("i") == pc.scalar(2))

arrow_scanner = ds.Scanner.from_dataset(my_arrow_dataset, filter=scanner_
 ↪  filter)

# query the Apache Arrow scanner "arrow_scanner" and return as an Arrow Table
results = con.execute("SELECT * FROM arrow_scanner").arrow()
```

Apache Arrow RecordBatchReaders

Arrow RecordBatchReaders are a reader for Arrow's streaming binary format and can also be queried directly as if they were tables. This streaming format is useful when sending Arrow data for tasks like interprocess communication or communicating between language runtimes.

```
import duckdb
import pyarrow as pa

# connect to an in-memory database
con = duckdb.connect()

my_recordbatch = pa.RecordBatch.from_pydict({'i': [1, 2, 3, 4],
                                             'j': ["one", "two", "three",
                                              ↪  "four"]})

my_recordbatchreader = pa.ipc.RecordBatchReader.from_batches(my_
 ↪  recordbatch.schema, [my_recordbatch])

# query the Apache Arrow RecordBatchReader "my_recordbatchreader" and return
 ↪  as an Arrow Table
results = con.execute("SELECT * FROM my_recordbatchreader WHERE i =
 ↪  2").arrow()
```

Import from Apache Arrow

`CREATE TABLE AS` and `INSERT INTO` can be used to create a table from any query. We can then create tables or insert into existing tables by referring to referring to the Apache Arrow object in the query. This example imports from an Arrow Table, but DuckDB can query different Apache Arrow formats as seen in the SQL on Arrow guide.

```
import duckdb
import pyarrow as pa

# connect to an in-memory database
my_arrow = pa.Table.from_pydict({'a': [42]})

# create the table "my_table" from the DataFrame "my_arrow"
duckdb.sql("CREATE TABLE my_table AS SELECT * FROM my_arrow")

# insert into the table "my_table" from the DataFrame "my_arrow"
duckdb.sql("INSERT INTO my_table SELECT * FROM my_arrow")
```

Export to Apache Arrow

All results of a query can be exported to an Apache Arrow Table using the `arrow` function. Alternatively, results can be returned as a RecordBatchReader using the `fetch_record_batch` function and results can be read one batch at a time. In addition, relations built using DuckDB's Relational API can also be exported.

Export to an Arrow Table

```
import duckdb
import pyarrow as pa

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# query the Apache Arrow Table "my_arrow_table" and return as an Arrow Table
results = duckdb.sql("SELECT * FROM my_arrow_table").arrow()
```

Export as a RecordBatchReader

```
import duckdb
import pyarrow as pa

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# query the Apache Arrow Table "my_arrow_table" and return as an Arrow
 ↪  RecordBatchReader
chunk_size = 1_000_000
results = duckdb.sql("SELECT * FROM my_arrow_table").fetch_record_
 ↪  batch(chunk_size)

# Loop through the results. A StopIteration exception is thrown when the
 ↪  RecordBatchReader is empty
while True:
    try:
        # Process a single chunk here (just printing as an example)
        print(results.read_next_batch().to_pandas())
    except StopIteration:
        print('Already fetched all batches')
        break
```

Export from Relational API

Arrow objects can also be exported from the Relational API. A relation can be converted to an Arrow table using the `arrow` or `to_arrow_table` functions, or a record batch using `record_batch`. A result can be exported to an Arrow table with `arrow` or the alias `fetch_arrow_table`, or to a RecordBatchReader using `fetch_arrow_reader`.

```
import duckdb

# connect to an in-memory database
con = duckdb.connect()

con.execute('CREATE TABLE integers (i integer)')
con.execute('INSERT INTO integers VALUES (0), (1), (2), (3), (4), (5), (6),
 ↪  (7), (8), (9), (NULL)')

# Create a relation from the table and export the entire relation as Arrow
rel = con.table("integers")
relation_as_arrow = rel.arrow() # or .to_arrow_table()

# Or, calculate a result using that relation and export that result to Arrow
res = rel.aggregate("sum(i)").execute()
result_as_arrow = res.arrow() # or fetch_arrow_table()
```

Relational API and Pandas

DuckDB offers a relational API that can be used to chain together query operations. These are lazily evaluated so that DuckDB can optimize their execution. These operators can act on Pandas DataFrames, DuckDB tables or views (which can point to any underlying storage format that DuckDB can read, such as CSV or parquet files, etc.). Here we show a simple example of reading from a Pandas DataFrame and returning a DataFrame.

```
import duckdb
import pandas

# connect to an in-memory database
con = duckdb.connect()

input_df = pandas.DataFrame.from_dict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})
```

```
# create a DuckDB relation from a dataframe
rel = con.from_df(input_df)

# chain together relational operators (this is a lazy operation, so the
 ↪  operations are not yet executed)
# equivalent to: SELECT i, j, i*2 AS two_i FROM input_df ORDER BY i DESC
 ↪  LIMIT 2
transformed_rel = rel.filter('i >= 2').project('i, j, i*2 as two_
 ↪  i').order('i desc').limit(2)

# trigger execution by requesting .df() of the relation
# .df() could have been added to the end of the chain above - it was
 ↪  separated for clarity
output_df = transformed_rel.df()
```

Relational operators can also be used to group rows, aggregate, find distinct combinations of values, join, union, and more. They are also able to directly insert results into a DuckDB table or write to a CSV.

Please see these additional examples and the available relational methods on the DuckDBPyRelation class.

Multiple Python Threads

This page demonstrates how to simultaneously insert into and read from a DuckDB database across multiple Python threads. This could be useful in scenarios where new data is flowing in and an analysis should be periodically re-run. Note that this is all within a single Python process (see the FAQ for details on DuckDB concurrency). Feel free to follow along in this Google Colab notebook.

Setup

First, import duckdb and several modules from the Python standard library. Note: if using Pandas, add `import pandas` at the top of the script as well (as it must be imported prior to the multi-threading). Then connect to a file-backed DuckDB database and create an example table to store inserted data. This table will track the name of the thread that completed the insert and automatically insert the timestamp when that insert occurred using the DEFAULT expression.

```
import duckdb
from threading import Thread, current_thread
import random
```

```
duckdb_con = duckdb.connect('my_peristent_db.duckdb')
# duckdb_con = duckdb.connect() # Pass in no parameters for an in memory
 ↪  database
duckdb_con.execute("""
    CREATE OR REPLACE TABLE my_inserts (
        thread_name varchar,
        insert_time timestamp DEFAULT current_timestamp
    )
""")
```

Reader and Writer Functions

Next, define functions to be executed by the writer and reader threads. Each thread must use the `.cursor()` method to create a thread-local connection to the same DuckDB file based on the original connection. This approach also works with in-memory DuckDB databases.

```
def write_from_thread(duckdb_con):
    # Create a DuckDB connection specifically for this thread
    local_con = duckdb_con.cursor()
    # Insert a row with the name of the thread. insert_time is
     ↪  auto-generated.
    thread_name = str(current_thread().name)
    result = local_con.execute("""
        INSERT INTO my_inserts (thread_name)
        VALUES (?)
    """, (thread_name,)).fetchall()

def read_from_thread(duckdb_con):
    # Create a DuckDB connection specifically for this thread
    local_con = duckdb_con.cursor()
    # Query the current row count
    thread_name = str(current_thread().name)
    results = local_con.execute("""
        SELECT
            ? AS thread_name,
            count(*) AS row_counter,
            current_timestamp
        FROM my_inserts
    """, (thread_name,)).fetchall()
    print(results)
```

Create Threads

We define how many writers and readers to use, and define a list to track all of the Threads that will be created. Then, create first writer and then reader Threads. Next, shuffle them so that they will be kicked off in a random order to simulate simultaneous writers and readers. Note that the Threads have not yet been executed, only defined.

```
write_thread_count = 50
read_thread_count = 5
threads = []

# Create multiple writer and reader threads (in the same process)
# Pass in the same connection as an argument
for i in range(write_thread_count):
    threads.append(Thread(target=write_from_thread,
                          args=(duckdb_con,),
                          name='write_thread_'+str(i)))

for j in range(read_thread_count):
    threads.append(Thread(target=read_from_thread,
                          args=(duckdb_con,),
                          name='read_thread_'+str(j)))

# Shuffle the threads to simulate a mix of readers and writers
random.seed(6) # Set the seed to ensure consistent results when testing
random.shuffle(threads)
```

Run Threads and Show Results

Now, kick off all threads to run in parallel, then wait for all of them to finish before printing out the results. Note that the timestamps of readers and writers are interspersed as expected due to the randomization.

```
# Kick off all threads in parallel
for thread in threads:
    thread.start()

# Ensure all threads complete before printing final results
for thread in threads:
    thread.join()

print(duckdb_con.execute("""
```

```
    SELECT
        *
    FROM my_inserts
    ORDER BY
        insert_time
""").df())
```

DuckDB with Ibis

Ibis is a Python dataframe library that supports 15+ backends, with DuckDB as the default. Ibis with DuckDB provides a Pythonic interface for SQL with great performance.

Installation

You can pip install Ibis with the DuckDB backend:

```
pip install 'ibis-framework[duckdb]'
```

or use conda:

```
conda install ibis-framework
```

or use mamba:

```
mamba install ibis-framework
```

Create a Database File

Ibis can work with several file types, but at its core, it connects to existing databases and interacts with the data there. You can get started with your own DuckDB databases or create a new one with example data.

```
import ibis

con = ibis.connect("duckdb://penguins.ddb")
con.create_table(
    "penguins", ibis.examples.penguins.fetch().to_pyarrow(), overwrite=True
)

# Output:
DatabaseTable: penguins
  species           string
```

```
  island            string
  bill_length_mm    float64
  bill_depth_mm     float64
  flipper_length_mm int64
  body_mass_g       int64
  sex               string
  year              int64
```

You can now see the example dataset copied over to the database:

```
# reconnect to the persisted database (dropping temp tables)
con = ibis.connect("duckdb://penguins.ddb")
con.list_tables()
```

```
# Output:
['penguins']
```

There's one table, called `penguins`. We can ask Ibis to give us an object that we can interact with.

```
penguins = con.table("penguins")
penguins
```

```
# Output:
DatabaseTable: penguins
  species           string
  island            string
  bill_length_mm    float64
  bill_depth_mm     float64
  flipper_length_mm int64
  body_mass_g       int64
  sex               string
  year              int64
```

Ibis is lazily evaluated, so instead of seeing the data, we see the schema of the table. To peek at the data, we can call `head` and then `to_pandas` to get the first few rows of the table as a pandas DataFrame.

```
penguins.head().to_pandas()
```

```
  species     island  bill_length_mm  bill_depth_mm  flipper_length_mm
 ↪  body_mass_g     sex  year
0  Adelie  Torgersen            39.1           18.7               181.0
 ↪  3750.0    male  2007
1  Adelie  Torgersen            39.5           17.4               186.0
 ↪  3800.0  female  2007
```

```
2  Adelie  Torgersen           40.3          18.0               195.0
 ↪  3250.0  female  2007
3  Adelie  Torgersen            NaN           NaN                 NaN
 ↪  NaN    None  2007
4  Adelie  Torgersen           36.7          19.3               193.0
 ↪  3450.0  female  2007
```

`to_pandas` takes the existing lazy table expression and evaluates it. If we leave it off, you'll see the Ibis representation of the table expression that `to_pandas` will evaluate (when you're ready!).

```
penguins.head()
```

```
# Output:
r0 := DatabaseTable: penguins
  species           string
  island            string
  bill_length_mm    float64
  bill_depth_mm     float64
  flipper_length_mm int64
  body_mass_g       int64
  sex               string
  year              int64

Limit[r0, n=5]
```

Ibis returns results as a pandas DataFrame using `to_pandas`, but isn't using pandas to perform any of the computation. The query is executed by DuckDB. Only when `to_pandas` is called does Ibis then pull back the results and convert them into a DataFrame.

Interactive Mode

For the rest of this intro, we'll turn on interactive mode, which partially executes queries to give users a preview of the results. There is a small difference in the way the output is formatted, but otherwise this is the same as calling `to_pandas` on the table expression with a limit of 10 result rows returned.

```
ibis.options.interactive = True
penguins.head()
```

```
┏━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┓
┃ species ┃ island   ┃ bill_length_mm ┃ bill_depth_mm ┃ flipper_length_mm ┃
 ↪  body_mass_g ┃ sex    ┃ year  ┃
┡━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
│ string  │ string   │ float64        │ float64       │ int64             │
 ↪  int64       │ string │ int64 │
```

```
┌─────────┬───────────┬────────────────┬───────────────┬───────────────────┬─────────
│ Adelie  │ Torgersen │           39.1 │          18.7 │               181 │
↪  3750 │ male   │  2007 │
│ Adelie  │ Torgersen │           39.5 │          17.4 │               186 │
↪  3800 │ female │  2007 │
│ Adelie  │ Torgersen │           40.3 │          18.0 │               195 │
↪  3250 │ female │  2007 │
│ Adelie  │ Torgersen │            nan │           nan │              NULL │
↪  NULL │ NULL   │  2007 │
│ Adelie  │ Torgersen │           36.7 │          19.3 │               193 │
↪  3450 │ female │  2007 │
└─────────┴───────────┴────────────────┴───────────────┴───────────────────┴─────────
```

Common Operations

Ibis has a collection of useful table methods to manipulate and query the data in a table.

filter `filter` allows you to select rows based on a condition or set of conditions.

We can filter so we only have penguins of the species Adelie:

```
penguins.filter(penguins.species == "Gentoo")
```

```
┏━━━━━━━━━┳━━━━━━━━┳━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┓
┃ species ┃ island ┃ bill_length_mm ┃ bill_depth_mm ┃ flipper_length_mm ┃
↪  body_mass_g ┃ sex    ┃ year  ┃
┡━━━━━━━━━╇━━━━━━━━╇━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
│ string  │ string │ float64        │ float64       │ int64             │
↪  int64       │ string │ int64 │
├─────────┼────────┼────────────────┼───────────────┼───────────────────┤
│ Gentoo  │ Biscoe │           46.1 │          13.2 │               211 │
↪  4500 │ female │  2007 │
│ Gentoo  │ Biscoe │           50.0 │          16.3 │               230 │
↪  5700 │ male   │  2007 │
│ Gentoo  │ Biscoe │           48.7 │          14.1 │               210 │
↪  4450 │ female │  2007 │
│ Gentoo  │ Biscoe │           50.0 │          15.2 │               218 │
↪  5700 │ male   │  2007 │
│ Gentoo  │ Biscoe │           47.6 │          14.5 │               215 │
↪  5400 │ male   │  2007 │
│ Gentoo  │ Biscoe │           46.5 │          13.5 │               210 │
↪  4550 │ female │  2007 │
```

```
┃ Gentoo  ┃ Biscoe ┃           45.4 ┃          14.6 ┃               211 ┃
↪  4800 ┃ female ┃  2007 ┃
┃ Gentoo  ┃ Biscoe ┃           46.7 ┃          15.3 ┃               219 ┃
↪  5200 ┃ male   ┃  2007 ┃
┃ Gentoo  ┃ Biscoe ┃           43.3 ┃          13.4 ┃               209 ┃
↪  4400 ┃ female ┃  2007 ┃
┃ Gentoo  ┃ Biscoe ┃           46.8 ┃          15.4 ┃               215 ┃
↪  5150 ┃ male   ┃  2007 ┃
┃ …       ┃ …      ┃              … ┃             … ┃                 … ┃
↪  … ┃ …      ┃     … ┃
```

Or filter for Adelie penguins that reside on the island of Torgersen:

```
penguins.filter((penguins.species == "Gentoo") & (penguins.body_mass_g >
↪  6000))
```

```
┃ species ┃ island ┃ bill_length_mm ┃ bill_depth_mm ┃ flipper_length_mm ┃
↪  body_mass_g ┃ sex    ┃ year  ┃
┃ string  ┃ string ┃ float64        ┃ float64       ┃ int64             ┃
↪  int64       ┃ string ┃ int64 ┃
┃ Gentoo  ┃ Biscoe ┃           49.2 ┃          15.2 ┃               221 ┃
↪  6300 ┃ male   ┃  2007 ┃
┃ Gentoo  ┃ Biscoe ┃           59.6 ┃          17.0 ┃               230 ┃
↪  6050 ┃ male   ┃  2007 ┃
```

You can use any boolean comparison in a filter (although if you try to do something like use < on a string, Ibis will yell at you).

select Your data analysis might not require all the columns present in a given table. `select` lets you pick out only those columns that you want to work with.

To select a column you can use the name of the column as a string:

```
penguins.select("species", "island", "year").limit(3)
```

```
┃ species ┃ island ┃ year  ┃
┃ string  ┃ string ┃ int64 ┃
```

```
┌─────────┬───────────┬─────────┐
│ Adelie  │ Torgersen │  2007   │
│ Adelie  │ Torgersen │  2007   │
│ Adelie  │ Torgersen │  2007   │
│ …       │ …         │    …    │
└─────────┴───────────┴─────────┘
```

Or you can use column objects directly (this can be convenient when paired with tab-completion):

```
penguins.select(penguins.species, penguins.island, penguins.year).limit(3)
```

```
┌─────────┬───────────┬─────────┐
│ species │ island    │ year    │
│ string  │ string    │ int64   │
├─────────┼───────────┼─────────┤
│ Adelie  │ Torgersen │  2007   │
│ Adelie  │ Torgersen │  2007   │
│ Adelie  │ Torgersen │  2007   │
│ …       │ …         │    …    │
└─────────┴───────────┴─────────┘
```

Or you can mix-and-match:

```
penguins.select("species", "island", penguins.year).limit(3)
```

```
┌─────────┬───────────┬─────────┐
│ species │ island    │ year    │
│ string  │ string    │ int64   │
├─────────┼───────────┼─────────┤
│ Adelie  │ Torgersen │  2007   │
│ Adelie  │ Torgersen │  2007   │
│ Adelie  │ Torgersen │  2007   │
│ …       │ …         │    …    │
└─────────┴───────────┴─────────┘
```

mutate `mutate` lets you add new columns to your table, derived from the values of existing columns.

```
penguins.mutate(bill_length_cm=penguins.bill_length_mm / 10)
```

```
┌─────────┬───────────┬────────────────┬───────────────┬───────────────────┐
│ species │ island    │ bill_length_mm │ bill_depth_mm │ flipper_length_mm │
↪  body_mass_g │ sex    │ year  │ bill_length_cm │
```

```
┌─────────┬──────────┬────────────────┬────────────────┬────────────────┐
│ string  │ string   │ float64        │ float64        │ int64          │
↪  int64       │ string │ int64 │ float64        │
├─────────┼──────────┼────────────────┼────────────────┼────────────────┤
│ Adelie  │ Torgersen │           39.1 │           18.7 │            181 │
↪  3750 │ male   │  2007 │           3.91 │
│ Adelie  │ Torgersen │           39.5 │           17.4 │            186 │
↪  3800 │ female │  2007 │           3.95 │
│ Adelie  │ Torgersen │           40.3 │           18.0 │            195 │
↪  3250 │ female │  2007 │           4.03 │
│ Adelie  │ Torgersen │            nan │            nan │           NULL │
↪  NULL │ NULL   │  2007 │            nan │
│ Adelie  │ Torgersen │           36.7 │           19.3 │            193 │
↪  3450 │ female │  2007 │           3.67 │
│ Adelie  │ Torgersen │           39.3 │           20.6 │            190 │
↪  3650 │ male   │  2007 │           3.93 │
│ Adelie  │ Torgersen │           38.9 │           17.8 │            181 │
↪  3625 │ female │  2007 │           3.89 │
│ Adelie  │ Torgersen │           39.2 │           19.6 │            195 │
↪  4675 │ male   │  2007 │           3.92 │
│ Adelie  │ Torgersen │           34.1 │           18.1 │            193 │
↪  3475 │ NULL   │  2007 │           3.41 │
│ Adelie  │ Torgersen │           42.0 │           20.2 │            190 │
↪  4250 │ NULL   │  2007 │           4.20 │
│ …       │ …        │              … │              … │              … │
↪  … │ …      │    … │              … │
└─────────┴──────────┴────────────────┴────────────────┴────────────────┘
```

Notice that the table is a little too wide to display all the columns now (depending on your screen-size). `bill_length` is now present in millimeters *and* centimeters. Use a `select` to trim down the number of columns we're looking at.

```
penguins.mutate(bill_length_cm=penguins.bill_length_mm / 10).select(
    "species",
    "island",
    "bill_depth_mm",
    "flipper_length_mm",
    "body_mass_g",
    "sex",
    "year",
    "bill_length_cm",
)
```

```
┌─────────┬──────────┬────────────────┬────────────────┬────────────────┐
```

```
┃ species ┃ island    ┃ bill_depth_mm ┃ flipper_length_mm ┃ body_mass_g ┃
↪  sex     ┃ year  ┃ bill_length_cm ┃
┡━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━╇
│ string  │ string    │ float64       │ int64             │ int64       │
↪  string │ int64 │ float64        │
├─────────┼───────────┼───────────────┼───────────────────┼─────────────┼
│ Adelie  │ Torgersen │          18.7 │               181 │        3750 │ male
↪  │  2007 │           3.91 │
│ Adelie  │ Torgersen │          17.4 │               186 │        3800 │
↪  female │  2007 │           3.95 │
│ Adelie  │ Torgersen │          18.0 │               195 │        3250 │
↪  female │  2007 │           4.03 │
│ Adelie  │ Torgersen │           nan │              NULL │        NULL │ NULL
↪  │  2007 │            nan │
│ Adelie  │ Torgersen │          19.3 │               193 │        3450 │
↪  female │  2007 │           3.67 │
│ Adelie  │ Torgersen │          20.6 │               190 │        3650 │ male
↪  │  2007 │           3.93 │
│ Adelie  │ Torgersen │          17.8 │               181 │        3625 │
↪  female │  2007 │           3.89 │
│ Adelie  │ Torgersen │          19.6 │               195 │        4675 │ male
↪  │  2007 │           3.92 │
│ Adelie  │ Torgersen │          18.1 │               193 │        3475 │ NULL
↪  │  2007 │           3.41 │
│ Adelie  │ Torgersen │          20.2 │               190 │        4250 │ NULL
↪  │  2007 │           4.20 │
│ …       │ …         │             … │                 … │           … │ …
↪  │     … │             … │
└─────────┴───────────┴───────────────┴───────────────────┴─────────────┴
```

selectors Typing out *all* of the column names *except* one is a little annoying. Instead of doing that again, we can use a `selector` to quickly select or deselect groups of columns.

```
import ibis.selectors as s

penguins.mutate(bill_length_cm=penguins.bill_length_mm / 10).select(
    ~s.matches("bill_length_mm")
    # match every column except `bill_length_mm`
)
```

```
┃ species ┃ island    ┃ bill_depth_mm ┃ flipper_length_mm ┃ body_mass_g ┃
↪  sex     ┃ year  ┃ bill_length_cm ┃
```

```
┌─────────┬──────────┬────────────────┬───────────────────┬─────────────┬─────────┐
│ string  │ string   │ float64        │ int64             │ int64       │
↪  string │ int64 │ float64        │
├─────────┼──────────┼────────────────┼───────────────────┼─────────────┼─────────┤
│ Adelie  │ Torgersen │          18.7 │               181 │        3750 │ male
↪   │  2007 │           3.91 │
│ Adelie  │ Torgersen │          17.4 │               186 │        3800 │
↪  female │  2007 │           3.95 │
│ Adelie  │ Torgersen │          18.0 │               195 │        3250 │
↪  female │  2007 │           4.03 │
│ Adelie  │ Torgersen │           nan │              NULL │        NULL │ NULL
↪   │  2007 │            nan │
│ Adelie  │ Torgersen │          19.3 │               193 │        3450 │
↪  female │  2007 │           3.67 │
│ Adelie  │ Torgersen │          20.6 │               190 │        3650 │ male
↪   │  2007 │           3.93 │
│ Adelie  │ Torgersen │          17.8 │               181 │        3625 │
↪  female │  2007 │           3.89 │
│ Adelie  │ Torgersen │          19.6 │               195 │        4675 │ male
↪   │  2007 │           3.92 │
│ Adelie  │ Torgersen │          18.1 │               193 │        3475 │ NULL
↪   │  2007 │           3.41 │
│ Adelie  │ Torgersen │          20.2 │               190 │        4250 │ NULL
↪   │  2007 │           4.20 │
│ …       │ …        │              … │                 … │           … │ …
↪   │     … │            … │
└─────────┴──────────┴────────────────┴───────────────────┴─────────────┴─────────┘
```

You can also use a `selector` alongside a column name.

```
penguins.select("island", s.numeric())
```

```
┌──────────┬────────────────┬───────────────┬───────────────────┬─────────────┐
│ island   │ bill_length_mm │ bill_depth_mm │ flipper_length_mm │ body_
↪  mass_g │ year  │
├──────────┼────────────────┼───────────────┼───────────────────┼─────────────┤
│ string   │ float64        │ float64       │ int64             │ int64
↪   │ int64 │
├──────────┼────────────────┼───────────────┼───────────────────┼─────────────┤
│ Torgersen │           39.1 │          18.7 │               181 │        3750
↪   │  2007 │
│ Torgersen │           39.5 │          17.4 │               186 │        3800
↪   │  2007 │
```

```
│ Torgersen │           40.3 │           18.0 │               195 │        3250
↪  │  2007 │
│ Torgersen │            nan │            nan │              NULL │        NULL
↪  │  2007 │
│ Torgersen │           36.7 │           19.3 │               193 │        3450
↪  │  2007 │
│ Torgersen │           39.3 │           20.6 │               190 │        3650
↪  │  2007 │
│ Torgersen │           38.9 │           17.8 │               181 │        3625
↪  │  2007 │
│ Torgersen │           39.2 │           19.6 │               195 │        4675
↪  │  2007 │
│ Torgersen │           34.1 │           18.1 │               193 │        3475
↪  │  2007 │
│ Torgersen │           42.0 │           20.2 │               190 │        4250
↪  │  2007 │
│ …         │              … │              … │                 … │           …
↪  │     … │
└───────────┴────────────────┴────────────────┴───────────────────┴──────────────
```

You can read more about `selectors` in the docs!

order_by `order_by` arranges the values of one or more columns in ascending or descending order.

By default, `ibis` sorts in ascending order:

```
penguins.order_by(penguins.flipper_length_mm).select(
    "species", "island", "flipper_length_mm"
)
```

species	island	flipper_length_mm
string	string	int64
Adelie	Biscoe	172
Adelie	Biscoe	174
Adelie	Torgersen	176
Adelie	Dream	178
Adelie	Dream	178
Adelie	Dream	178
Chinstrap	Dream	178
Adelie	Dream	179

Adelie	Torgersen	180
Adelie	Biscoe	180
…	…	…

You can sort in descending order using the `desc` method of a column:

```
penguins.order_by(penguins.flipper_length_mm.desc()).select(
    "species", "island", "flipper_length_mm"
)
```

species	island	flipper_length_mm
string	string	int64
Gentoo	Biscoe	231
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	229
Gentoo	Biscoe	229
…	…	…

Or you can use `ibis.desc`

```
penguins.order_by(ibis.desc("flipper_length_mm")).select(
    "species", "island", "flipper_length_mm"
)
```

species	island	flipper_length_mm
string	string	int64
Gentoo	Biscoe	231
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230

Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	229
Gentoo	Biscoe	229
…	…	…

aggregate Ibis has several aggregate functions available to help summarize data.

`mean`, `max`, `min`, `count`, `sum` (the list goes on).

To aggregate an entire column, call the corresponding method on that column.

```
penguins.flipper_length_mm.mean()
```

```
# Output:
200.91520467836258
```

You can compute multiple aggregates at once using the `aggregate` method:

```
penguins.aggregate([penguins.flipper_length_mm.mean(), penguins.bill_depth_
↪   mm.max()])
```

Mean(flipper_length_mm)	Max(bill_depth_mm)
float64	float64
200.915205	21.5

But `aggregate` *really* shines when it's paired with `group_by`.

group_by `group_by` creates groupings of rows that have the same value for one or more columns.

But it doesn't do much on its own -- you can pair it with `aggregate` to get a result.

```
penguins.group_by("species").aggregate()
```

species
string

species
Adelie
Gentoo
Chinstrap

We grouped by the `species` column and handed it an "empty" aggregate command. The result of that is a column of the unique values in the `species` column.

If we add a second column to the `group_by`, we'll get each unique pairing of the values in those columns.

```
penguins.group_by(["species", "island"]).aggregate()
```

species	island
string	string
Adelie	Torgersen
Adelie	Biscoe
Adelie	Dream
Gentoo	Biscoe
Chinstrap	Dream

Now, if we add an aggregation function to that, we start to really open things up.

```
penguins.group_by(["species", "island"]).aggregate(penguins.bill_length_
↪  mm.mean())
```

species	island	Mean(bill_length_mm)
string	string	float64
Adelie	Torgersen	38.950980
Adelie	Biscoe	38.975000
Adelie	Dream	38.501786
Gentoo	Biscoe	47.504878
Chinstrap	Dream	48.833824

By adding that `mean` to the `aggregate`, we now have a concise way to calculate aggregates over each of the distinct groups in the `group_by`. And we can calculate as many aggregates as we need.

```
penguins.group_by(["species", "island"]).aggregate(
    [penguins.bill_length_mm.mean(), penguins.flipper_length_mm.max()]
)
```

species	island	Mean(bill_length_mm)	Max(flipper_length_mm)
string	string	float64	int64
Adelie	Torgersen	38.950980	210
Adelie	Biscoe	38.975000	203
Adelie	Dream	38.501786	208
Gentoo	Biscoe	47.504878	231
Chinstrap	Dream	48.833824	212

If we need more specific groups, we can add to the `group_by`.

```
penguins.group_by(["species", "island", "sex"]).aggregate(
    [penguins.bill_length_mm.mean(), penguins.flipper_length_mm.max()]
)
```

species	island	sex	Mean(bill_length_mm)	Max(flipper_length_mm)
string	string	string	float64	int64
Adelie	Torgersen	male	40.586957	210
Adelie	Torgersen	female	37.554167	196
Adelie	Torgersen	NULL	37.925000	193
Adelie	Biscoe	female	37.359091	199
Adelie	Biscoe	male	40.590909	203
Adelie	Dream	female	36.911111	202
Adelie	Dream	male	40.071429	208
Adelie	Dream	NULL	37.500000	179
Gentoo	Biscoe	female	45.563793	222
Gentoo	Biscoe	male	49.473770	231
…	…	…	…	…

Chaining It All Together

We've already chained some Ibis calls together. We used `mutate` to create a new column and then `select` to only view a subset of the new table. We were just chaining `group_by` with `aggregate`.

There's nothing stopping us from putting all of these concepts together to ask questions of the data.

How about:

- What was the largest female penguin (by body mass) on each island in the year 2008?

```
penguins.filter((penguins.sex == "female") & (penguins.year == 2008)).group_
↪   by(
    ["island"]
).aggregate(penguins.body_mass_g.max())
```

island	Max(body_mass_g)
string	int64
Biscoe	5200
Torgersen	3800
Dream	3900

- What about the largest male penguin (by body mass) on each island for each year of data collection?

```
penguins.filter(penguins.sex == "male").group_by(["island",
↪   "year"]).aggregate(
    penguins.body_mass_g.max().name("max_body_mass")
).order_by(["year", "max_body_mass"])
```

island	year	max_body_mass
string	int64	int64
Dream	2007	4650
Torgersen	2007	4675
Biscoe	2007	6300
Torgersen	2008	4700
Dream	2008	4800

```
┌────────────┬──────┬──────────────────┐
│ Biscoe     │ 2008 │             6000 │
│ Torgersen  │ 2009 │             4300 │
│ Dream      │ 2009 │             4475 │
│ Biscoe     │ 2009 │             6000 │
└────────────┴──────┴──────────────────┘
```

Learn More

That's all for this quick-start guide. If you want to learn more, check out the Ibis documentation.

DuckDB with Polars

Polars is a DataFrames library built in Rust with bindings for Python and Node.js. It uses Apache Arrow's columnar format as its memory model. DuckDB can read Polars DataFrames and convert query results to Polars DataFrames. It does this internally using the efficient Apache Arrow integration. Note that the `pyarrow` library must be installed for the integration to work.

Installation

```
pip install duckdb
pip install -U 'polars[pyarrow]'
```

Polars to DuckDB

DuckDB can natively query Polars DataFrames by referring to the name of Polars DataFrames as they exist in the current scope.

```
import duckdb
import polars as pl

df = pl.DataFrame(
    {
        "A": [1, 2, 3, 4, 5],
        "fruits": ["banana", "banana", "apple", "apple", "banana"],
        "B": [5, 4, 3, 2, 1],
        "cars": ["beetle", "audi", "beetle", "beetle", "beetle"],
    }
)
duckdb.sql('SELECT * FROM df').show()
```

DuckDB to Polars

DuckDB can output results as Polars DataFrames using the `.pl()` result-conversion method.

```
df = duckdb.sql("""
SELECT 1 AS id, 'banana' AS fruit
UNION ALL
SELECT 2, 'apple'
UNION ALL
SELECT 3, 'mango'""").pl()
print(df)
```

To learn more about Polars, feel free to explore their Python API Reference!

DuckDB with Vaex

Vaex is a high performance DataFrame library in Python. Vaex is a hybrid DataFrame, as it supports both Numpy's and Apache Arrow's data structures. Vaex DataFrames can export data as Apache Arrow Table, which can be directly used by DuckDB. Since DuckDB can output results as an Apache Arrow Table which can be easily turned into a Vaex DataFrame, one can easily alternate between DuckDB and Vaex.

The following example shows how one can use both DuckDB and Vaex DataFrame for a simple exploratory work.

Installation

```
pip install duckdb
pip install vaex
```

Vaex DataFrame to DuckDB

A Vaex DataFrame can be exported as an Arrow Table via the `to_arrow_table()` method. This operation does not take extra memory if the data being exported is already in memory or memory-mapped. The exported Arrow Table can be queried directly via DuckDB.

Let's use the well known Titanic dataset that also ships with Vaex, to do some operations like filling missing values and creating new columns. Then we will export the DataFrame to an Arrow Table:

```
import duckdb
import vaex

df = vaex.datasets.titanic()

df['age'] = df.age.fillna(df.age.mean())
df['fare'] = df.age.fillna(df.fare.mean())
df['family_size'] = (df.sibsp + df.parch + 1)
df['fare_per_family_member'] = df.fare / df.family_size
df['name_title'] = df['name'].str.replace('.* ([A-Z][a-z]+)\..*', "\\1",
 ↪  regex=True)

arrow_table  = df.to_arrow_table()
```

Now we can directly query the Arrow Table using DuckDB, the output of which can be another Arrow Table, which can be used for subsequent DuckDB queries, or it can be converted to a Vaex DataFrame:

```
query_result_arrow_table = duckdb.query('''
    SELECT
        pclass,
        MEAN(age) AS age,
        MEAN(family_size) AS family_size,
        MEAN(fare_per_family_member) AS fare_per_family_member,
        COUNT(DISTINCT(name_title)) AS distinct_titles,
        LIST(DISTINCT(name_title))
    FROM arrow_table
    GROUP BY pclass
    ORDER BY pclass
''').arrow()
```

DuckDB to Vaex DataFrame

The output of a DuckDB query can be an Arrow Table, which can be easily converted to a Vaex DataFrame via the `vaex.from_arrow_table()` method. One can also pass data around via Pandas DataFrames, but Arrow is faster.

We can use the query result from above and convert it to a vaex DataFrame:

```
df_from_duckdb = vaex.from_arrow_table(query_result_arrow_table)
```

One can then continue to use Vaex, and also export the data or part of it to an Arrow Table to be used with DuckDB as needed.

To learn more about Vaex, feel free to explore their Documentation.

DuckDB with DataFusion

DataFusion is a DataFrame and SQL library built in Rust with bindings for Python. It uses Apache Arrow's columnar format as its memory model. DataFusion can output results as Apache Arrow, and DuckDB can read those results directly. DuckDB can also rapidly output results to Apache Arrow, which can be easily converted to a DataFusion DataFrame. Due to the interoperability of Apache Arrow, workflows can alternate between DuckDB and DataFusion with ease!

This example workflow is also available as a Google Colab notebook.

Installation

```
pip install --quiet duckdb datafusion pyarrow
```

DataFusion to DuckDB

To convert from DataFusion to DuckDB, first save DataFusion results into Arrow batches using the `collect` function, and then create an Arrow table using PyArrow's `Table.from_batches` function. Then include that Arrow Table in the FROM clause of a DuckDB query.

As a note, Pandas is not required as a first step prior to using DataFusion, but was helpful for generating example data to reuse in the second example below.

Import the libraries, create an example Pandas DataFrame, then convert to DataFusion.

```
import duckdb
import pyarrow as pa
import pandas as pd
import datafusion as df
from datafusion import functions as f

pandas_df = pd.DataFrame(
    {
        "A": [1, 2, 3, 4, 5],
        "fruits": ["banana", "banana", "apple", "apple", "banana"],
        "B": [5, 4, 3, 2, 1],
        "cars": ["beetle", "audi", "beetle", "beetle", "beetle"],
    }
)
```

```
arrow_table = table = pa.Table.from_pandas(pandas_df)
arrow_batches = table.to_batches()

ctx = SessionContext()
datafusion_df = ctx.create_dataframe([arrow_batches])
datafusion_df
```

Calculate a new DataFusion DataFrame and output it to a variable as an Apache Arrow table.

```
arrow_batches = (
    datafusion_df
    .aggregate(
        [df.col("fruits")],
        [f.sum(df.col("A")).alias("sum_A_by_fruits")]
    )
    .sort(df.col("fruits").sort(ascending=True))
    .collect()
)
datafusion_to_arrow = (
    pa.Table.from_batches(arrow_batches)
)
datafusion_to_arrow
```

Then query the Apache Arrow table using DuckDB, and output the results as another Apache Arrow table for use in a subsequent DuckDB or DataFusion operation.

```
output = duckdb.query("""
  SELECT
    fruits,
    first(sum_A_by_fruits) AS sum_A
  FROM datafusion_to_arrow
  GROUP BY ALL
  ORDER BY ALL
""").arrow()
```

DuckDB to DataFusion

DuckDB can output results as Apache Arrow tables, which can be imported into DataFusion with the DataFusion DataFrame constructor. The same approach could be used with Pandas DataFrames, but Arrow is a faster way to pass data between DuckDB and DataFusion.

This example reuses the original Pandas DataFrame created above as a starting point. As a note, Pandas is not required as a first step, but was only used to generate example data.

After the import statements and example DataFrame creation above, query the Pandas DataFrame using DuckDB and output the results as an Arrow table.

```
duckdb_to_arrow = duckdb.query("""
  SELECT
    fruits,
    cars,
    'fruits' AS literal_string_fruits,
    SUM(B) FILTER (cars = 'beetle') OVER () AS B,
    SUM(A) FILTER (B > 2) OVER (PARTITION BY cars) AS sum_A_by_cars,
    SUM(A) OVER (PARTITION BY fruits) AS sum_A_by_fruits
  FROM df
  ORDER BY
    fruits,
    df.B
""").arrow()
```

Load the Apache Arrow table into DataFusion using the DataFusion DataFrame constructor.

```
datafusion_df_2 = ctx.create_dataframe([duckdb_to_arrow.to_batches()])
datafusion_df_2
```

Complete a calculation using DataFusion, then output the results as another Apache Arrow table for use in a subsequent DuckDB or DataFusion operation.

```
output_2 = (
    datafusion_df_2
    .aggregate(
        [df.col("fruits")],
        [f.sum(df.col('sum_A_by_fruits'))]
    )
).collect()
output_2
```

To learn more about DataFusion, feel free to explore their GitHub repository!

Filesystems

DuckDB support for `fsspec` filesystems allows querying data in filesystems that DuckDB's `httpfs` extension does not support. `fsspec` has a large number of inbuilt filesystems, and there are also many external implementations. This capability is only available in DuckDB's Python client because `fsspec` is a Python library, while the `httpfs` extension is available in many DuckDB clients.

Example

The following is an example of using `fsspec` to query a file in Google Cloud Storage (instead of using their s3 inter-compatibility api).

Firstly, you must install `duckdb` and `fsspec`, and a filesystem interface of your choice

```
$ pip install duckdb fsspec gcsfs
```

then you can register whichever filesystem you'd like to query

```
import duckdb
from fsspec import filesystem

# this line will throw an exception if the appropriate filesystem interface
 ↪  is not installed
duckdb.register_filesystem(filesystem('gcs'))

duckdb.sql("SELECT * FROM read_csv_auto('gcs:///bucket/file.csv')")
```

> **Note.** These filesystems are not implemented in C++, hence, their performance may not be comparable to the ones provided by the `httpfs` extension. It is also worth noting that as they are third party libraries, they may contain bugs that are beyond our control.

SQL Features

DuckDB ASOF Join

Problem: we have a time-based price table; traditional joins against this table get NULL results if there is a time which does not exactly match.

Solution: `ASOF JOIN` picks a good value for "in the gap" values.

First, we create a price table and sales table.

```
CREATE TABLE prices AS (
    SELECT '2001-01-01 00:16:00'::TIMESTAMP + INTERVAL (v) MINUTE AS ticker_
     ↪  time,
        v AS unit_price
    FROM range(0, 5) vals(v)
);

CREATE TABLE sales(item TEXT, sale_time TIMESTAMP, quantity INT);
INSERT INTO sales VALUES('a', '2001-01-01 00:18:00', 10);
INSERT INTO sales VALUES('b', '2001-01-01 00:18:30', 20);
INSERT INTO sales VALUES('c', '2001-01-01 00:19:00', 30);
```

We can see that we have a `unit_price` defined for each hour, but not for half hours.

```
SELECT * FROM prices;
```

```
┌─────────────────────┬────────────┐
│     ticker_time     │ unit_price │
│      timestamp      │   int64    │
├─────────────────────┼────────────┤
│ 2001-01-01 00:16:00 │          0 │
│ 2001-01-01 00:17:00 │          1 │
│ 2001-01-01 00:18:00 │          2 │ No unit_price for 18:30!
│ 2001-01-01 00:19:00 │          3 │
│ 2001-01-01 00:20:00 │          4 │
└─────────────────────┴────────────┘
```

```
SELECT * FROM sales;
```

item varchar	sale_time timestamp	quantity int32	
a	2001-01-01 00:18:00	10	
b	2001-01-01 00:18:30	20	A sale time of 18:30!
c	2001-01-01 00:19:00	30	

With a normal `LEFT JOIN`, there is a problem for the 18:30 sale. Since there is not a sale_time of 18:30, a join against that time will be NULL.

```
-- no price value for 18:30, so item b's unit_price and total are NULL!

SELECT s.*, p.unit_price, s.quantity * p.unit_price AS total
FROM sales s
LEFT JOIN prices p
       ON s.sale_time = p.ticker_time;
```

item varchar	sale_time timestamp	quantity int32	unit_price int64	total int64	
a	2001-01-01 00:18:00	10	2	20	
c	2001-01-01 00:19:00	30	3	90	
b	2001-01-01 00:18:30	20	NULL	NULL	NULL result!

The `ASOF JOIN` picks a good price for the 18:30 sale. the `ON s.sale_time >= pp.ticker_time` will cause the nearest lower value (in this case, for 18:00) to be used.

```
-- using ASOF, 18:30 "rounds down" to use the 18:00 unit_price

SELECT s.*, p.unit_price, s.quantity * p.unit_price AS total_cost
FROM sales s ASOF
LEFT JOIN prices p
       ON s.sale_time >= p.ticker_time;
```

item varchar	sale_time timestamp	quantity int32	unit_price int64	total_cost int64
a	2001-01-01 00:18:00	10	2	20

```
│ b       │ 2001-01-01 00:18:30 │       20 │          2 │          40 │ Good
↪  result!
│ c       │ 2001-01-01 00:19:00 │       30 │          3 │          90 │
└─────────┴─────────────────────┴──────────┴────────────┴─────────────┘
```

DuckDB Full Text Search

A full text index allows for a query to quickly search for all occurrences of individual words within longer text strings. Here's an example of building a full text index of Shakespeare's plays.

```
CREATE TABLE corpus AS
  SELECT * FROM read_parquet(
    'https://github.com/marhar/duckdb_
     ↪  tools/raw/main/full-text-shakespeare/shakespeare.parquet');

DESCRIBE corpus;
```

column_name	column_type	null
line_id	VARCHAR	YES
play_name	VARCHAR	YES
line_number	VARCHAR	YES
speaker	VARCHAR	YES
text_entry	VARCHAR	YES

The text of each line is in `text_entry`, and a unique key for each line is in `line_id`.

First, we create the index, specifying the table name, the unique id column, and the column(s) to index. We will just index the single column `text_entry`, which contains the text of the lines in the play.

```
INSTALL fts;
LOAD fts;
PRAGMA create_fts_index('corpus', 'line_id', 'text_entry');
```

The table is now ready to query using the Okapi BM25 ranking function. Rows with no match return a null score.

What does Shakespeare say about butter?

```
SELECT fts_main_corpus.match_bm25(line_id, 'butter') AS score,
    line_id, play_name, speaker, text_entry
  FROM corpus
```

```
  WHERE score IS NOT NULL
  ORDER BY score;
```

```
┌────────────────────┬──────────┬──────────────────────┬──────────────┬──────────────────────────────────────────────────┐
│       score        │ line_id  │      play_name       │   speaker    │                    text_entry                    │
│       double       │ varchar  │       varchar        │   varchar    │                     varchar                      │
├────────────────────┼──────────┼──────────────────────┼──────────────┼──────────────────────────────────────────────────┤
│ 2.683490686835495  │ H4/2.4.115   │ Henry IV             │ PRINCE HENRY │ Didst thou never see Titan kiss a dish of …      │
│ 3.781282331450016  │ H4/1.2.21    │ Henry IV             │ FALSTAFF     │ prologue to an egg and butter.                   │
│ 3.781282331450016  │ H4/2.1.55    │ Henry IV             │ Chamberlain  │ They are up already, and call for eggs and…      │
│ 3.781282331450016  │ H4/4.2.21    │ Henry IV             │ FALSTAFF     │ toasts-and-butter, with hearts in their be…      │
│ 3.781282331450016  │ H4/4.2.62    │ Henry IV             │ PRINCE HENRY │ already made thee butter. But tell me, Jac…      │
│ 3.781282331450016  │ AWW/4.1.40   │ Alls well that end…  │ PAROLLES     │ butter-womans mouth and buy myself another…      │
│ 3.781282331450016  │ AWW/5.2.9    │ Alls well that end…  │ Clown        │ henceforth eat no fish of fortunes butteri…      │
│ 3.781282331450016  │ AYLI/3.2.93  │ As you like it       │ TOUCHSTONE   │ right butter-womens rank to market.              │
│ 3.781282331450016  │ KL/2.4.132   │ King Lear            │ Fool         │ kindness to his horse, buttered his hay.         │
│ 3.781282331450016  │ MWW/2.2.260  │ Merry Wives of Win…  │ FALSTAFF     │ Hang him, mechanical salt-butter rogue! I …      │
│ 3.781282331450016  │ MWW/2.2.284  │ Merry Wives of Win…  │ FORD         │ rather trust a Fleming with my butter, Par…      │
│ 3.781282331450016  │ MWW/3.5.7    │ Merry Wives of Win…  │ FALSTAFF     │ Ill have my brains taen out and buttered, …      │
│ 3.781282331450016  │ MWW/3.5.102  │ Merry Wives of Win…  │ FALSTAFF     │ to heat as butter; a man of continual diss…      │
│ 6.399093176300027  │ H4/2.4.494   │ Henry IV             │ Carrier      │ As fat as butter.                                │
├────────────────────┴──────────┴──────────────────────┴──────────────┴──────────────────────────────────────────────────┤
│ 14 rows                                                                                                  5 columns │
└────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

Unlike standard indexes, full text indexes don't auto-update as the underlying data is changed, so you

need to `PRAGMA drop_fts_index(my_fts_index)` and recreate it when appropriate.

Note on Generating the Corpus Table

Details are here

- The Columns are: line_id, play_name, line_number, speaker, text_entry.
- We need a unique key for each row in order for full text searching to work.
- The line_id ”KL/2.4.132” means King Lear, Act 2, Scene 4, Line 132.

SQL Editors

DBeaver SQL IDE

DBeaver is a powerful and popular desktop sql editor and integrated development environment (IDE). It has both an open source and enterprise version. It is useful for visually inspecting the available tables in DuckDB and for quickly building complex queries. DuckDB's JDBC connector allows DBeaver to query DuckDB files, and by extension, any other files that DuckDB can access (like parquet files).

1. Install DBeaver using the download links and instructions found at their download page.
2. Open DBeaver and create a new connection. Either click on the "New Database Connection" button or go to Database > New Database Connection in the menu bar.

3. Search for DuckDB, select it, and click Next.

4. Enter the path or browse to the DuckDB database file you wish to query. To use an in-memory DuckDB (useful primarily if just interested in querying parquet files, or for testing) enter `:memory:` as the path.

5. Click "Test Connection". This will then prompt you to install the DuckDB JDBC driver. If you are not prompted, see alternative driver installation instructions below.

6. Click "Download" to download DuckDB's JDBC driver from Maven. Once download is complete, click "OK", then click "Finish".

- Note: If you are in a corporate environment or behind a firewall, before clicking download, click the "Download Configuration" link to configure your proxy settings.

1. You should now see a database connection to your DuckDB database in the left hand "Database Navigator" pane. Expand it to see the tables and views in your database. Right click on that

connection and create a new SQL script.

2. Write some SQL and click the ”Execute” button.

3. Now you're ready to fly with DuckDB and DBeaver!

Alternative Driver Installation

1. If not prompted to install the DuckDB driver when testing your connection, return to the ”Connect to a database” dialog and click ”Edit Driver Settings”.

2. (Alternate) You may also access the driver settings menu by returning to the main DBeaver window and clicking Database > Driver Manager in the menu bar. Then select DuckDB, then click Edit.

3. Go to the ”Libraries” tab, then click on the DuckDB driver and click ”Download/Update”. If you do not see the DuckDB driver, first click on ”Reset to Defaults”.

4. Click ”Download” to download DuckDB's JDBC driver from Maven. Once download is complete, click ”OK”, then return to the main DBeaver window and continue with step 7 above.

- Note: If you are in a corporate environment or behind a firewall, before clicking download, click the ”Download Configuration” link to configure your proxy settings.

Data Viewers

Tableau - A Data Visualisation Tool

Tableau is a popular commercial data visualisation tool. In addition to a large number of built in connectors, it also provides generic database connectivity via ODBC and JDBC connectors.

Tableau has two main versions: Desktop and Online (Server).

- For Desktop, connecting to a DuckDB database is similar to working in an embedded environment like Python.
- For Online, since DuckDB is in-process, the data needs to be either on the server itself or in a remote data bucket that is accessible from the server.

Database Creation

When using a DuckDB database file the data sets do not actually need to be imported into DuckDB tables; it suffices to create views of the data. For example, this will create a view of the `h2oai` parquet test file in the current DuckDB code base:

```
CREATE VIEW h2oai AS (
    FROM read_
     ↪ parquet('/Users/username/duckdb/data/parquet-testing/h2oai/h2oai_
     ↪ group_small.parquet')
);
```

Note that you should use full path names to local files so that they can be found from inside Tableau. Also note that you will need to use a version of the driver that is compatible (i.e., from the same release) as the database format used by the DuckDB tool (e.g., Python module, command line) that was used to create the file.

Installing the JDBC Driver

Tableau provides documentation on how to install a JDBC driver for Tableau to use. For now, we recommend using the latest bleeding edge JDBC driver (0.8.2) as a number of fixes have been made

for time compatibility. Note that Tableau (both Desktop and Server versions) need to be restarted any time you add or modify drivers.

Driver Links The link here is for a recent version of the JDBC driver that is compatible with Tableau. If you wish to connect to a database file, you will need to make sure the file was created with a file-compatible version of DuckDB. Also, check that there is only one version of the driver installed as there are multiple filenames in use.

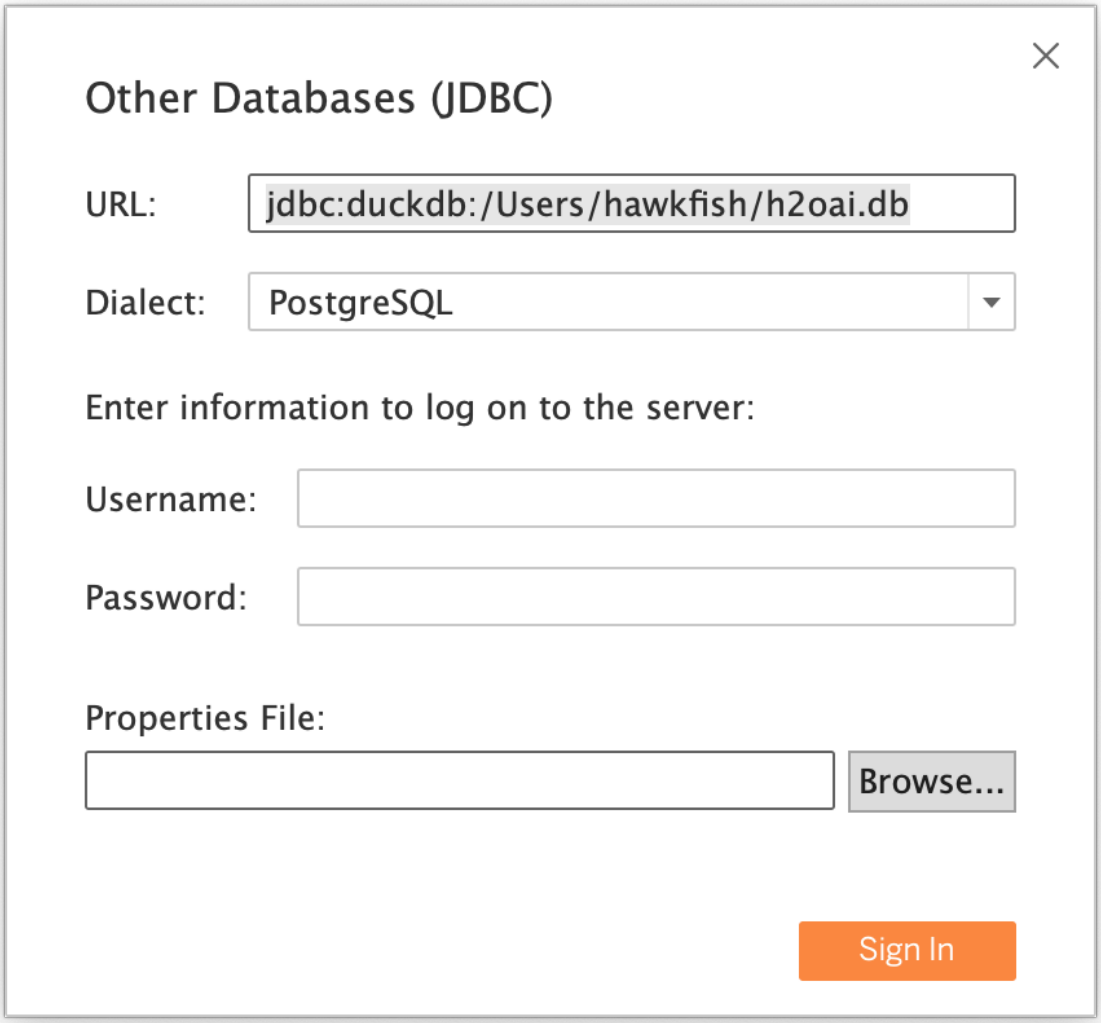
Download the snapshot jar

- MacOS: Copy it to `~/Library/Tableau/Drivers/`
- Windows: Copy it to `C:\Program Files\Tableau\Drivers`
- Linux: Copy it to `/opt/tableau/tableau_driver/jdbc`.

Using the PostgreSQL Dialect

If you just want to do something simple, you can try connecting directly to the JDBC driver and using Tableau-provided PostgreSQL dialect.

1. Create a DuckDB file containing your views and/or data.
2. Launch Tableau
3. Under Connect > To a Server > More… click on "Other Databases (JDBC)" This will bring up the connection dialogue box. For the URL, enter `jdbc:duckdb:/User/username/path/to/database.db`. For the Dialect, choose PostgreSQL. The rest of the fields can be ignored:



However, functionality will be missing such as MEDIAN and PERCENTILE aggregate functions. To make the data source connection more compatible with the PostgreSQL dialect, please use the DuckDB taco connector as described below.

Installing the Tableau DuckDB Connector

While it is possible to use the Tableau-provided PostgreSQL dialect to communicate with the DuckDB JDBC driver, we strongly recommend using the DuckDB "taco" connector. This connector has been fully tested against the Tableau dialect generator and is more compatible than the provided PostgreSQL dialect.

The documentation on how to install and use the connector is in its repository, but essentially you will

need the duckdb_jdbc.taco file. The current version of the Taco is not signed, so you will need to launch Tableau with signature validation disabled. (Despite what the Tableau documentation says, the real security risk is in the JDBC driver code, not the small amount of JavaScript in the Taco.)

Server (Online) On Linux, copy the Taco file to `/opt/tableau/connectors`. On Windows, copy the Taco file to `C:\Program Files\Tableau\Connectors`. Then issue these commands to disable signature validation:

```
$ tsm configuration set -k native_api.disable_verify_connector_plugin_
 ↪  signature -v true
$ tsm pending-changes apply
```

The last command will restart the server with the new settings.

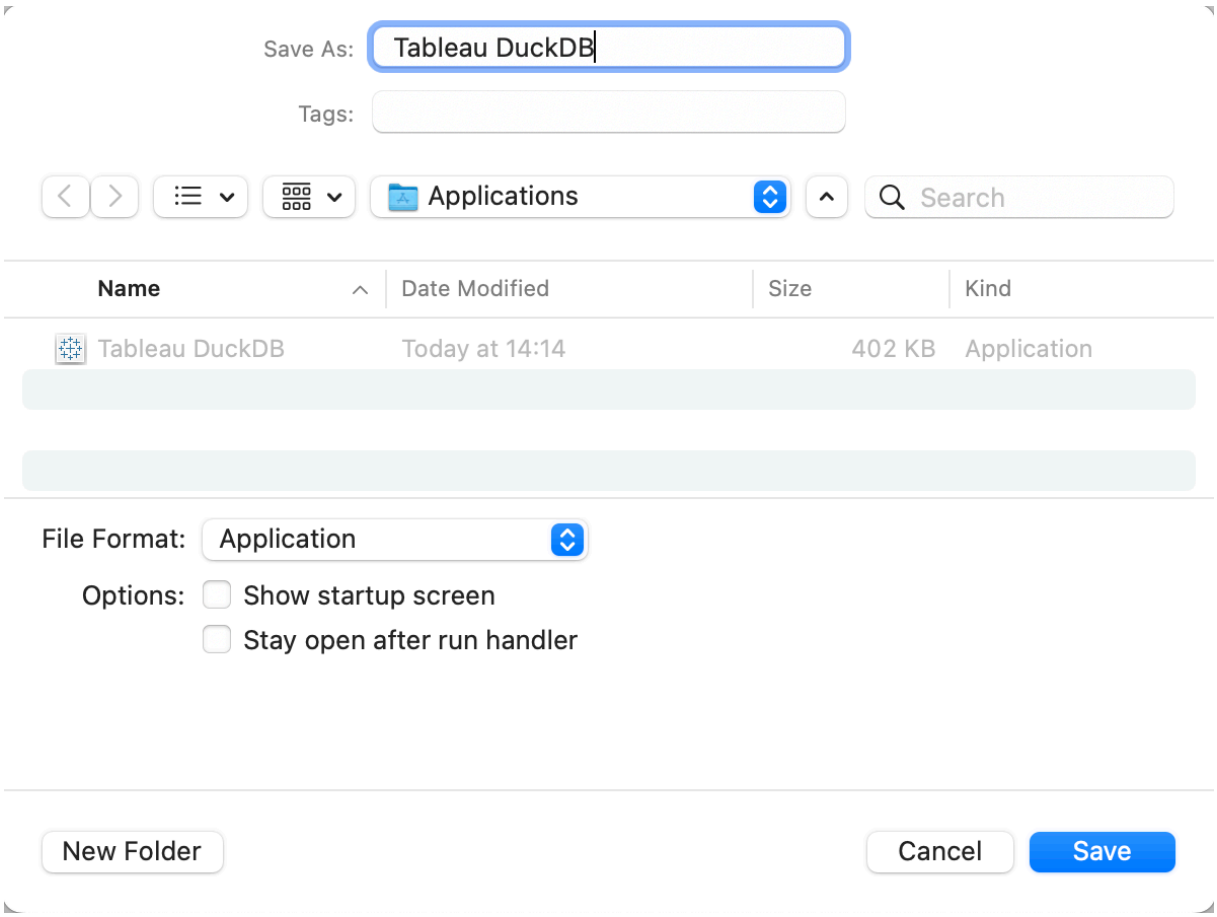
MacOS Desktop Copy the Taco file to the `/Users/[MacOS User]/Documents/My Tableau Repository/Connectors` folder. Then launch Tableau Desktop from the Terminal with the command line argument to disable signature validation:

```
$ /Applications/Tableau\ Desktop\
 ↪  <year>.<quarter>.app/Contents/MacOS/Tableau
 ↪  -DDisableVerifyConnectorPluginSignature=true
```

You can also package this up with AppleScript by using the following script:

```
do shell script "\"/Applications/Tableau Desktop
 ↪  2023.2.app/Contents/MacOS/Tableau\"
 ↪  -DDisableVerifyConnectorPluginSignature=true"
quit
```

Create this file with the Script Editor (located in `/Applications/Utilities`) and save it as a packaged application:

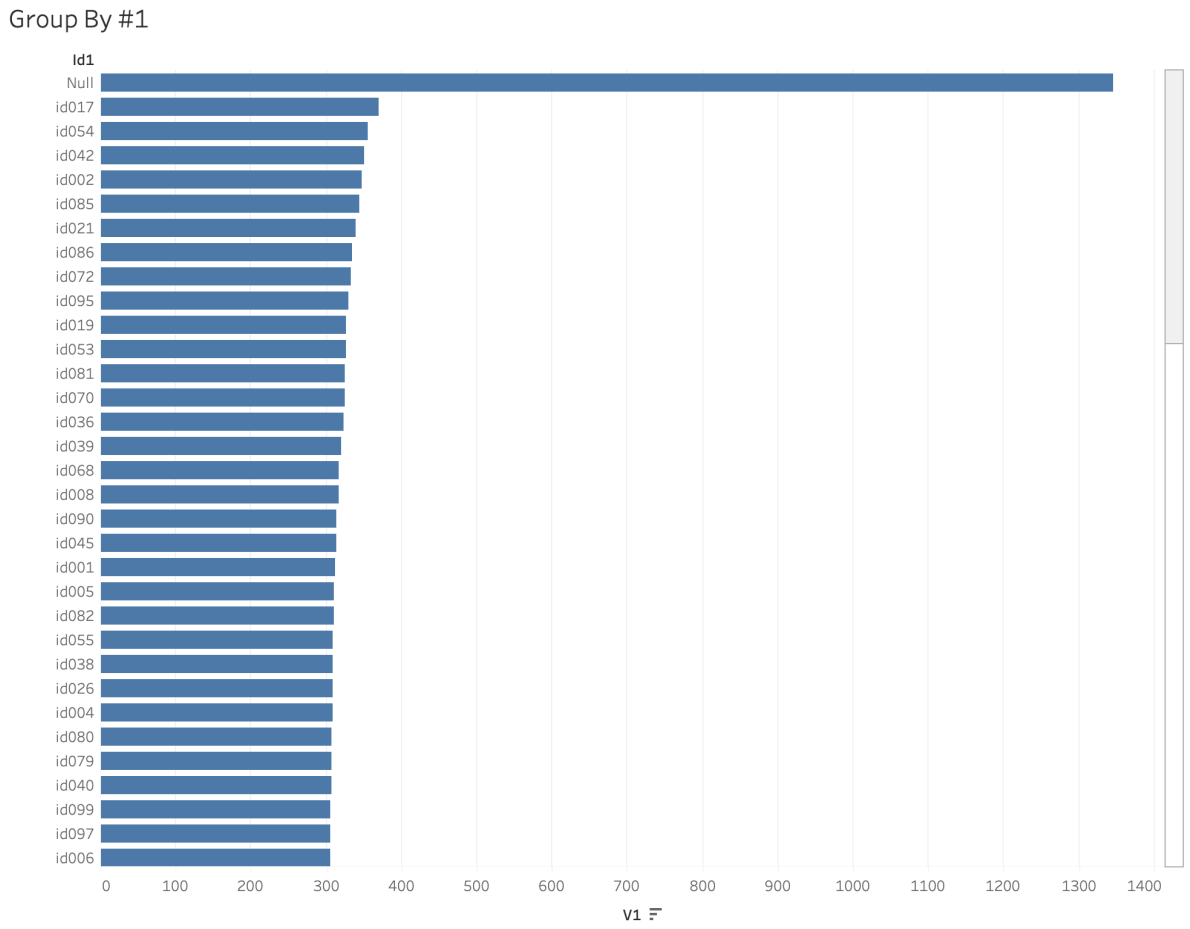


You can then double-click it to launch Tableau. You will need to change the application name in the script when you get upgrades.

Windows Desktop Copy the Taco file to the `C:\Users\[Windows User]\Documents\My Tableau Repository\Connectors` directory. Then launch Tableau Desktop from a shell with the `-DDisableVerifyConnectorPluginSignature=true` argument to disable signature validation.

Output

Once loaded, you can run queries against your data! Here is the result of the first h2oai benchmark query from the parquet test file:



CLI Charting - Using DuckDB with CLI Tools

DuckDB can be used with CLI graphing tools to quickly pipe input to stdout to graph your data in one line.

YouPlot is a Ruby-based CLI tool for drawing visually pleasing plots on the terminal. It can accept input from other programs by piping data from `stdin`. It takes tab-separated (or delimiter of your choice) data and can easily generate various types of plots including bar, line, histogram and scatter.

With DuckDB, you can write to the console (`stdout`) by using the `TO '/dev/stdout'` command. And you can also write comma-separated values by using `WITH (FORMAT 'csv', HEADER)`.

Installing YouPlot

Installation instructions for YouPlot can be found on the main YouPlot repository. If you're on a Mac, you can use:

```
brew install youplot
```

Run `uplot --help` to ensure you've installed it successfully!

Piping DuckDB Queries to stdout

By combining the `COPY...TO` function with a CSV output file, data can be read from any format supported by DuckDB and piped to YouPlot. There are three important steps to doing this.

1. As an example, this is how to read all data from `input.json`:

   ```
   duckdb -s "SELECT * FROM read_json_auto('input.json')"
   ```

2. To prepare the data for YouPlot, write a simple aggregate:

   ```
   duckdb -s "SELECT date, SUM(purchases) AS total_purchases FROM read_
    ↪  json_auto('input.json') GROUP BY 1 ORDER BY 2 DESC LIMIT 10"
   ```

3. Finally, wrap the SELECT in the COPY...TO function with an output location of `/dev/stdout`.

   The syntax looks like this:

   ```
   COPY (<YOUR_SELECT_QUERY>) TO '/dev/stdout' WITH (FORMAT 'csv', HEADER)
   ```

   The full DuckDB command below outputs the query in CSV format with a header:

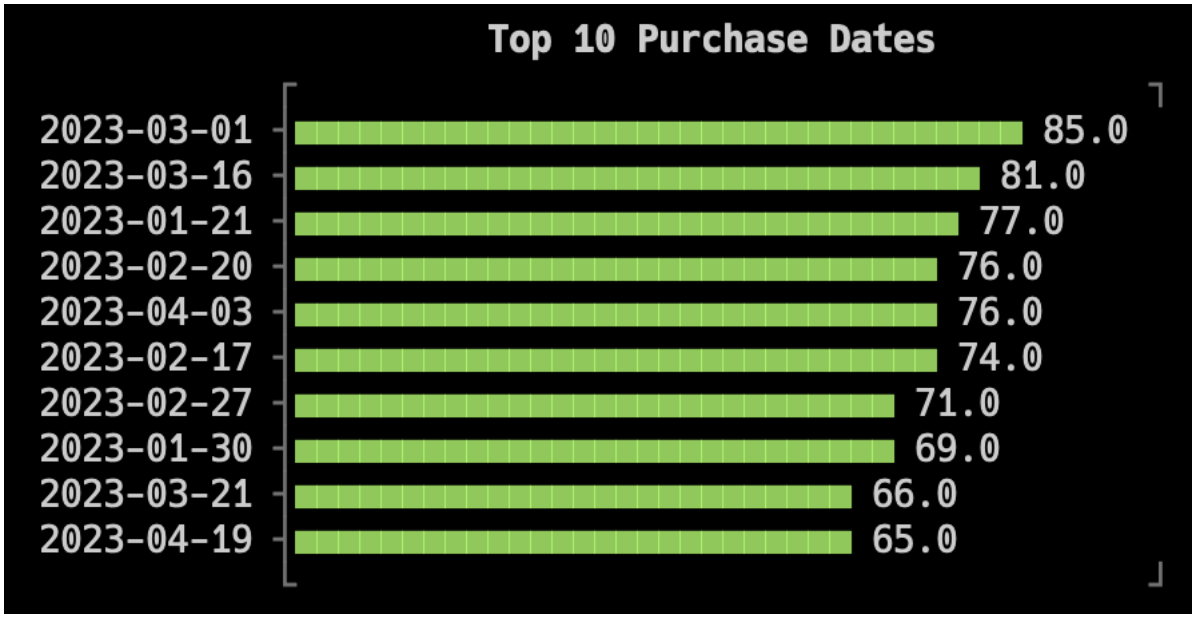
   ```
   duckdb -s "COPY (SELECT date, SUM(purchases) AS total_purchases FROM
    ↪  read_json_auto('input.json') GROUP BY 1 ORDER BY 2 DESC LIMIT 10)
    ↪  TO '/dev/stdout' WITH (FORMAT 'csv', HEADER)"
   ```

Connecting DuckDB to YouPlot

Finally, the data can now be piped to YouPlot! Let's assume we have an `input.json` file with dates and number of purchases made by somebody on that date. Using the query above, we'll pipe the data to the `uplot` command to draw a plot of the Top 10 Purchase Dates

```
duckdb -s "COPY (SELECT date, SUM(purchases) AS total_purchases FROM read_
 ↪  json_auto('input.json') GROUP BY 1 ORDER BY 2 DESC LIMIT 10) TO
 ↪  '/dev/stdout' WITH (FORMAT 'csv', HEADER)" | uplot bar -d, -H -t "Top 10
 ↪  Purchase Dates"
```

This tells `uplot` to draw a bar plot, use a comma-seperated delimiter (`-d ,`), that the data has a header (`-H`), and give the plot a title (`-t`).

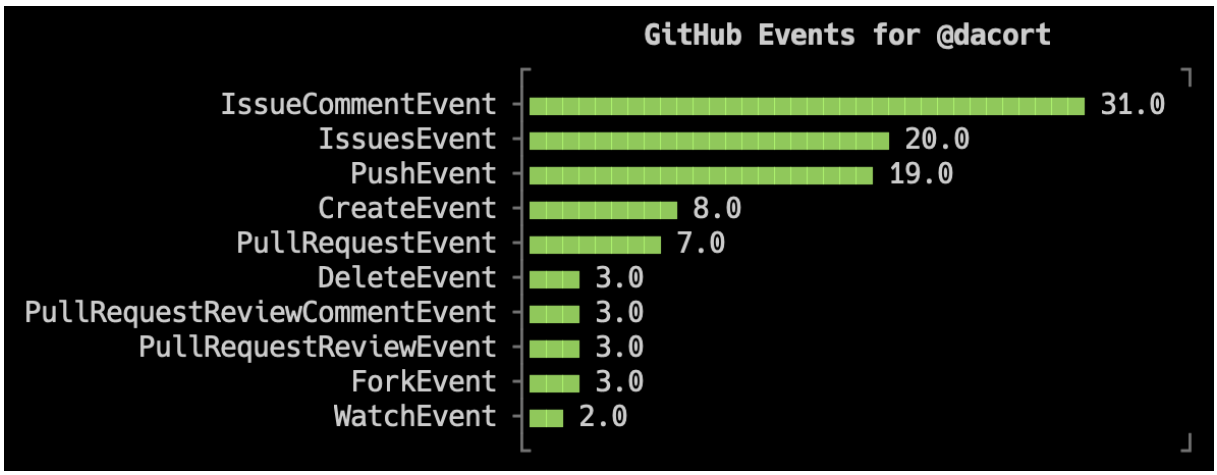


Bonus Round! stdin + stdout

Maybe you're piping some data through `jq`. Maybe you're downloading a JSON file from somewhere. You can also tell DuckDB to read the data from another process by changing the filename to `/dev/stdin`.

Let's combine this with a quick `curl` from GitHub to see what a certain user has been up to lately.

```
curl -sL "https://api.github.com/users/dacort/events?per_page=100" \
    | duckdb -s "COPY (SELECT type, count(*) AS event_count FROM read_json_
    ↪  auto('/dev/stdin') GROUP BY 1 ORDER BY 2 DESC LIMIT 10) TO
    ↪  '/dev/stdout' WITH (FORMAT 'csv', HEADER)" \
    | uplot bar -d, -H -t "GitHub Events for @dacort"
```


GitHub Events for @dacort
IssueCommentEvent 31.0
IssuesEvent 20.0
PushEvent 19.0
CreateEvent 8.0
PullRequestEvent 7.0
DeleteEvent 3.0
PullRequestReviewCommentEvent 3.0
PullRequestReviewEvent 3.0
ForkEvent 3.0
WatchEvent 2.0

Under the Hood

Internals

Overview of DuckDB Internals

On this page is a brief description of the internals of the DuckDB engine.

Parser

The parser converts a query string into the following tokens:

- `SQLStatement`
- `QueryNode`
- `TableRef`
- `ParsedExpression`

The parser is not aware of the catalog or any other aspect of the database. It will not throw errors if tables do not exist, and will not resolve **any** types of columns yet. It only transforms a query string into a set of tokens as specified.

ParsedExpression The ParsedExpression represents an expression within a SQL statement. This can be e.g. a reference to a column, an addition operator or a constant value. The type of the ParsedExpression indicates what it represents, e.g. a comparison is represented as a `ComparisonExpression`.

ParsedExpressions do **not** have types, except for nodes with explicit types such as CAST statements. The types for expressions are resolved in the Binder, not in the Parser.

TableRef The TableRef represents any table source. This can be a reference to a base table, but it can also be a join, a table-producing function or a subquery.

QueryNode The QueryNode represents either (1) a `SELECT` statement, or (2) a set operation (i.e. `UNION`, `INTERSECT` or `DIFFERENCE`).

SQL Statement The SQLStatement represents a complete SQL statement. The type of the SQL Statement represents what kind of statement it is (e.g. `StatementType::SELECT` represents a `SELECT` statement). A single SQL string can be transformed into multiple SQL statements in case the original query string contains multiple queries.

Binder

The binder converts all nodes into their **bound** equivalents. In the binder phase:

- The tables and columns are resolved using the catalog
- Types are resolved
- Aggregate/window functions are extracted

The following conversions happen:

- SQLStatement -> `BoundStatement`
- QueryNode -> `BoundQueryNode`
- TableRef -> `BoundTableRef`
- ParsedExpression -> `Expression`

Logical Planner

The logical planner creates `LogicalOperator` nodes from the bound statements. In this phase, the actual logical query tree is created.

Optimizer

After the logical planner has created the logical query tree, the optimizers are run over that query tree to create an optimized query plan. The following query optimizers are run:

- **Expression Rewriter**: Simplifies expressions, performs constant folding
- **Filter Pushdown**: Pushes filters down into the query plan and duplicates filters over equivalency sets. Also prunes subtrees that are guaranteed to be empty (because of filters that statically evaluate to false).
- **Join Order Optimizer**: Reorders joins using dynamic programming. Specifically, the DPcpp algorithm from the paper Dynamic Programming Strikes Back is used.
- **Common Sub Expressions**: Extracts common subexpressions from projection and filter nodes to prevent unnecessary duplicate execution.
- **In Clause Rewriter**: Rewrites large static IN clauses to a MARK join or INNER join.

Column Binding Resolver

The column binding resolver converts logical `BoundColumnRefExpresion` nodes that refer to a column of a specific table into `BoundReferenceExpression` nodes that refer to a specific index into the DataChunks that are passed around in the execution engine.

Physical Plan Generator

The physical plan generator converts the resulting logical operator tree into a `PhysicalOperator` tree.

Execution

In the execution phase, the physical operators are executed to produce the query result. The execution model is a vectorized volcano model, where `DataChunks` are pulled from the root node of the physical operator tree. Each PhysicalOperator itself defines how it grants its result. A `PhysicalTableScan` node will pull the chunk from the base tables on disk, whereas a `PhysicalHashJoin` will perform a hash join between the output obtained from its child nodes.

Storage

The DuckDB internal storage format is currently in flux, and is expected to change with each release until we reach v1.0.0.

How to Move Between Storage Formats

When you update DuckDB and open a database file, you might encounter an error message about incompatible storage formats, pointing to this page. To move your database(s) to newer format you only need the older and the newer DuckDB executable.

Open your database file with the older DuckDB and run the SQL statement `"EXPORT DATABASE 'tmp'"`. This allows you to save the whole state of the current database in use inside folder `tmp`. The content of the `tmp` folder will be overridden, so choose an empty/non yet existing location. Then, start the newer DuckDB and execute `"IMPORT DATABASE 'tmp'"` (pointing to the previously populated folder) to load the database, which can be then saved to the file you pointed DuckDB to.

A bash two-liner (to be adapted with the file names and executable locations) is:

```
$ /older/version/duckdb mydata.db -c "EXPORT DATABASE 'tmp'"
$ /newer/duckdb mydata.new.db -c "IMPORT DATABASE 'tmp'"
```

After this `mydata.db` will be untouched with the old format, `mydata.new.db` will contain the same data but in a format accessible from more recent DuckDB, and folder `tmp` will old the same data in an universal format as different files.

Check EXPORT documentation for more details on the syntax.

Storage Header

DuckDB files start with a `uint64_t` which contains a checksum for the main header, followed by four magic bytes (DUCK), followed by the storage version number in a `uint64_t`.

```
$ hexdump -n 20 -C mydata.db
00000000  01 d0 e2 63 9c 13 39 3e  44 55 43 4b 2b 00 00 00  |...c..9>DUCK+...|
00000010  00 00 00 00                                       |....|
00000014
```

A simple example of reading the storage version using Python is below.

```
import struct

pattern = struct.Struct('<8x4sQ')

with open('test/sql/storage_version/storage_version.db', 'rb') as fh:
    print(pattern.unpack(fh.read(pattern.size)))
```

Storage Version Table

For changes in each given release, check out the change log on GitHub. To see the commits that changed each storage version, see the commit log.

Storage version	DuckDB version(s)
64	v0.9.0, v0.9.1
51	v0.8.0, v0.8.1
43	v0.7.0, v0.7.1
39	v0.6.0, v0.6.1
38	v0.5.0, v0.5.1

Storage version	DuckDB version(s)
33	v0.3.3, v0.3.4, v0.4.0
31	v0.3.2
27	v0.3.1
25	v0.3.0
21	v0.2.9
18	v0.2.8
17	v0.2.7
15	v0.2.6
13	v0.2.5
11	v0.2.4
6	v0.2.3
4	v0.2.2
1	v0.2.1 and prior

Disk Usage

The disk usage of DuckDB's format depends on a number of factors, including the data type and the data distribution, the compression methods used, etc. As a rough approximation, loading 100 GB of uncompressed CSV files into a DuckDB database file will require 25 GB of disk space, while loading 100 GB of Parquet files will require 120 GB of disk space.

Execution Format

`Vector` is the container format used to store in-memory data during execution.
`DataChunk` is a collection of Vectors, used for instance to represent a column list in a PhysicalProjection operator.

Data Flow

DuckDB uses a vectorized query execution model.
All operators in DuckDB are optimized to work on Vectors of a fixed size.

This fixed size is commonly referred to in the code as `STANDARD_VECTOR_SIZE`.
The default `STANDARD_VECTOR_SIZE` is 2048 tuples.

Vector Format

Vectors logically represent arrays that contain data of a single type. DuckDB supports different *vector formats*, which allow the system to store the same logical data with a different *physical representation*. This allows for a more compressed representation, and potentially allows for compressed execution throughout the system. Below the list of supported vector formats is shown.

Flat Vectors Flat vectors are physically stored as a contiguous array, this is the standard uncompressed vector format. For flat vectors the logical and physical representations are identical.

Constant Vectors Constant vectors are physically stored as a single constant value.

Constant vectors are useful when data elements are repeated - for example, when representing the result of a constant expression in a function call, the constant vector allows us to only store the value once.

```
select lst || 'duckdb' from range(1000) tbl(lst);
```

Since `duckdb` is a string literal, the value of the literal is the same for every row. In a flat vector, we would have to duplicate the literal 'duckdb' once for every row. The constant vector allows us to only store the literal once.

Constant vectors are also emitted by the storage when decompressing from constant compression.

Dictionary Vectors Dictionary vectors are physically stored as a child vector, and a selection vector that contains indices into the child vector.

Dictionary vectors are emitted by the storage when decompressing from dictionary

Just like constant vectors, dictionary vectors are also emitted by the storage.
When deserializing a dictionary compressed column segment, we store this in a dictionary vector so we can keep the data compressed during query execution.

Sequence Vectors Sequence vectors are physically stored as an offset and an increment value.

Sequence vectors are useful for efficiently storing incremental sequences. They are generally emitted for row identifiers.

Unified Vector Format These properties of the different vector formats are great for optimization purposes, for example you can imagine the scenario where all the parameters to a function are constant, we can just compute the result once and emit a constant vector.
But writing specialized code for every combination of vector types for every function is unfeasible due to the combinatorial explosion of possibilities.

Instead of doing this, whenever you want to generically use a vector regardless of the type, the UnifiedVectorFormat can be used.
This format essentially acts as a generic view over the contents of the Vector. Every type of Vector can convert to this format.

Complex Types

String Vectors To efficiently store strings, we make use of our `string_t` class.

```
struct string_t {
    union {
        struct {
            uint32_t length;
            char prefix[4];
            char *ptr;
        } pointer;
        struct {
            uint32_t length;
            char inlined[12];
        } inlined;
    } value;
};
```

Short strings (`<= 12 bytes`) are inlined into the structure, while larger strings are stored with a pointer to the data in the auxiliary string buffer. The length is used throughout the functions to avoid having to call `strlen` and having to continuously check for null-pointers. The prefix is used for comparisons as an early out (when the prefix does not match, we know the strings are not equal and don't need to chase any pointers).

List Vectors List vectors are stored as a series of *list entries* together with a child Vector. The child vector contains the *values* that are present in the list, and the list entries specify how each individual list is constructed.

```
struct list_entry_t {
    idx_t offset;
    idx_t length;
};
```

The offset refers to the start row in the child Vector, the length keeps track of the size of the list of this row.

List vectors can be stored recursively. For nested list vectors, the child of a list vector is again a list vector.

For example, consider this mock representation of a Vector of type `BIGINT[][]`:

```
{
   "type": "list",
   "data": "list_entry_t",
   "child": {
      "type": "list",
      "data": "list_entry_t",
      "child": {
         "type": "bigint",
         "data": "int64_t"
      }
   }
}
```

Struct Vectors Struct vectors store a list of child vectors. The number and types of the child vectors is defined by the schema of the struct.

Map Vectors Internally map vectors are stored as a `LIST[STRUCT(key KEY_TYPE, value VALUE_TYPE)]`.

Union Vectors Internally `UNION` utilizes the same structure as a `STRUCT`. The first "child" is always occupied by the Tag Vector of the UNION, which records for each row which of the UNION's types apply to that row.

Developer Guides

Building DuckDB from Source

> **Note.** DuckDB binaries are available for stable and nightly builds on the installation page. You should only build DuckDB under specific circumstances, such as when running on a specific architecture or building an unmerged pull request.

Prerequisites DuckDB needs a C++11-compiler and CMake. Additionally, we recommend using the Ninja build system.

Linux Packages Install the required packages with the package manager of your distribution.

Fedora, CentOS, and Red Hat:

```
sudo yum install -y git g++ cmake ninja-build
```

Ubuntu and Debian:

```
sudo apt-get update
sudo apt-get install -y git g++ cmake ninja-build
```

Alpine Linux:

```
apk add g++ git make cmake ninja
```

macOS Install Xcode and Homebrew. Then, install the required packages with:

```
brew install cmake ninja
```

Windows Consult the Windows CI workflow for a list of packages used to build DuckDB on Windows.

Building DuckDB To build DuckDB we use a Makefile which in turn calls into CMake. We also advise using Ninja as the generator for CMake.

```
GEN=ninja make
```

It is not advised to directly call CMake, as the Makefile sets certain variables that are crucial to properly building the package.

Build Type DuckDB can be built in many different settings, most of these correspond directly to CMake but not all of them.

`release`

This build has been stripped of all the assertions and debug symbols and code, optimized for performance.

`debug`

This build runs with all the debug information, including symbols, assertions and DEBUG blocks.
The special debug defines are not automatically set for this build however.

`relassert`

This build does not trigger the `#ifdef DEBUG` code blocks, but still has debug symbols that make it possible to step through the execution with line number information and `D_ASSERT` lines are still checked in this build.

`reldebug`

This build is similar to `relassert` in many ways, only assertions are also stripped in this build.

`benchmark`

This build is a shorthand for `release` with `BUILD_BENCHMARK=1` set.

`tidy-check`

This creates a build and then runs clang tidy to check for issues or style violations through static analysis.
The CI will also run this check, causing it to fail if this check fails.

`format-fix` | `format-changes` | `format-main`

This doesn't actually create a build, but uses the following format checkers to check for style issues:

- clang-format to fix format issues in the code.
- cmake-format to fix format issues in the CMakeLists.txt files.

The CI will also run this check, causing it to fail if this check fails.

Package Flags For every package that is maintained by core DuckDB, there exists a flag in the Makefile to enable building the package.
These can be enabled by either setting them in the current `env`, through set up files like `bashrc` or `zshrc`, or by setting them before the call to `make`, for example:

```
BUILD_PYTHON=1 make debug
```

`BUILD_PYTHON`

When this flag is set, the Python package is built.

`BUILD_SHELL`

When this flag is set, the CLI is built, this is usually enabled by default.

`BUILD_BENCHMARK`

When this flag is set, our in-house Benchmark testing suite is built.
More information about this can be found here.

`BUILD_JDBC`

When this flag is set, the Java package is built.

`BUILD_ODBC`

When this flag is set, the ODBC package is built.

`BUILD_R`

When this flag is set, the R package is built.

`BUILD_NODE`

When this flag is set, the Node package is built.

Extension Flags For every in-tree extension that is maintained by core DuckDB there exists a flag to enable building and statically linking the extension into the build.

`BUILD_AUTOCOMPLETE`

When this flag is set, the AutoComplete extension is built.

`BUILD_ICU`

When this flag is set, the ICU extension is built.

`BUILD_TPCH`

When this flag is set, the TPCH extension is built, this enables TPCH-H data generation and query support using `dbgen`.

`BUILD_TPCDS`

When this flag is set, the TPCDS extension is built, this enables TPC-DS data generation and query support using `dsdgen`.

`BUILD_TPCE`

When this flag is set, the TPCE extension is built, unlike TPC-H and TPC-DS this does not enable data generation and query support, but does enable tests for TPC-E through our test suite.

`BUILD_FTS`

When this flag is set, the Full Text Search extension is built.

`BUILD_VISUALIZER`

When this flag is set, the Visualizer extension is built.

`BUILD_HTTPFS`

When this flag is set, the HTTP File System extension is built.

`BUILD_JSON`

When this flag is set, the JSON extension is built.

`BUILD_INET`

When this flag is set, the INET extension is built.

`BUILD_SQLSMITH`

When this flag is set, the SQLSmith extension is built.

Debug Flags `CRASH_ON_ASSERT`

`D_ASSERT(condition)` is used all throughout the code, these will throw an InternalException in debug builds.
With this flag enabled, when the assertion triggers it will instead directly cause a crash.

`DISABLE_STRING_INLINE`

In our execution format `string_t` has the feature to ”inline” strings that are under a certain length (12 bytes), this means they don't require a separate allocation.
When this flag is set, we disable this and don't inline small strings.

`DISABLE_MEMORY_SAFETY`

Our data structures that are used extensively throughout the non-performance-critical code have extra checks to ensure memory safety, these checks include:

- Making sure `nullptr` is never dereferenced.
- Making sure index out of bounds accesses don't trigger a crash.

With this flag enabled we remove these checks, this is mostly done to check that the performance hit of these checks is negligible.

`DESTROY_UNPINNED_BLOCKS`

When previously pinned blocks in the BufferManager are unpinned, with this flag enabled we destroy them instantly to make sure that there aren't situations where this memory is still being used, despite not being pinned.

`DEBUG_STACKTRACE`

When a crash or assertion hit occurs in a test, print a stack trace.
This is useful when debugging a crash that is hard to pinpoint with a debugger attached.

Miscellaneous Flags `DISABLE_UNITY`

To improve compilation time, we use Unity Build to combine translation units.
This can however hide include bugs, this flag disables using the unity build so these errors can be detected.

`DISABLE_SANITIZER`

In some situations, running an executable that has been built with sanitizers enabled is not support / can cause problems. Julia is an example of this.
With this flag enabled, the sanitizers are disabled for the build.

Troubleshooting

Building the R Package on Linux aarch64 Building the R package on Linux running on an ARM64 architecture (AArch64) may result in the following error message:

```
/usr/bin/ld: /usr/include/c++/10/bits/basic_string.tcc:206: warning: too
↪  many GOT entries for -fpic, please recompile with -fPIC
```

To work around this, create or edit the `~/.R/Makevars` file:

```
ALL_CXXFLAGS =  $(PKG_CXXFLAGS) -fPIC $(SHLIB_CXXFLAGS) $(CXXFLAGS)
```

Building the httpfs Extension and Python Package on macOS **Problem:** The build fails on macOS when both the `httpfs` extension and the Python package are included:

```
GEN=ninja BUILD_PYTHON=1 BUILD_HTTPFS=1 make
```

```
ld: library not found for -lcrypto
clang: error: linker command failed with exit code 1 (use -v to see
 ↪  invocation)
error: command '/usr/bin/clang++' failed with exit code 1
ninja: build stopped: subcommand failed.
make: *** [release] Error 1
```

Solution: In general, we recommended using the nightly builds, available under GitHub main (Bleeding Edge) on the installation page. If you would like to build DuckDB from source, avoid using the `BUILD_PYTHON=1` flag unless you are actively developing the Python library. Instead, first build the `httpfs` extension, then build and install the Python package separately using the `setup.py` script:

```
GEN=ninja BUILD_HTTPFS=1 make
python tools/pythonpkg/setup.py install --user
```

Profiling

Profiling is important to help understand why certain queries exhibit specific performance characteristics. DuckDB contains several built-in features to enable query profiling that will be explained on this page.

For the examples on this page we will use the following example data set:

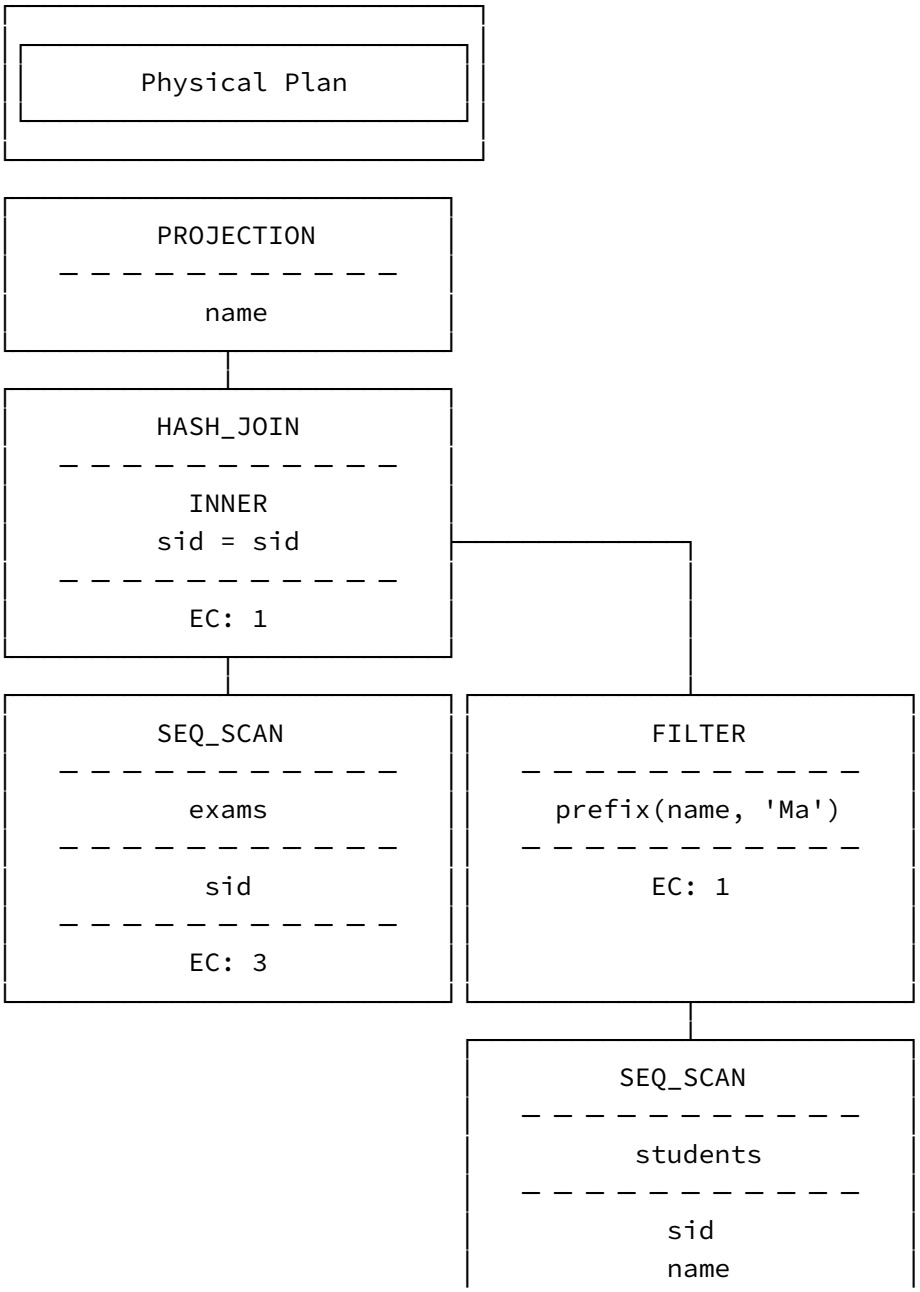
```
CREATE TABLE students(sid INTEGER PRIMARY KEY, name VARCHAR);
CREATE TABLE exams(cid INTEGER, sid INTEGER, grade INTEGER, PRIMARY KEY(cid,
 ↪  sid));

INSERT INTO students VALUES (1, 'Mark'), (2, 'Hannes'), (3, 'Pedro');
INSERT INTO exams VALUES (1, 1, 8), (1, 2, 8), (1, 3, 7), (2, 1, 9), (2, 2,
 ↪  10);
```

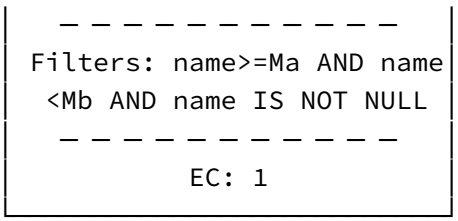
Explain Statement The first step to profiling a database engine is figuring out what execution plan the engine is using. The `EXPLAIN` statement allows you to peek into the query plan and see what is going on under the hood.

The `EXPLAIN` statement displays the physical plan, i.e., the query plan that will get executed.

To demonstrate, see the below example:

```
CREATE TABLE students(name VARCHAR, sid INT);
CREATE TABLE exams(eid INT, subject VARCHAR, sid INT);
INSERT INTO students VALUES ('Mark', 1), ('Joe', 2), ('Matthew', 3);
INSERT INTO exams VALUES (10, 'Physics', 1), (20, 'Chemistry', 2), (30,
 ↪  'Literature', 3);
EXPLAIN SELECT name FROM students JOIN exams USING (sid) WHERE name LIKE
 ↪  'Ma%';
```



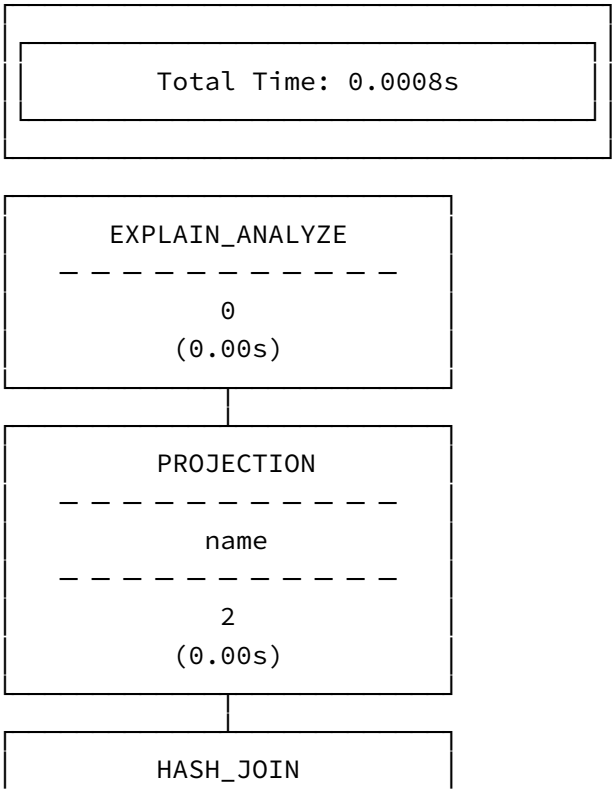


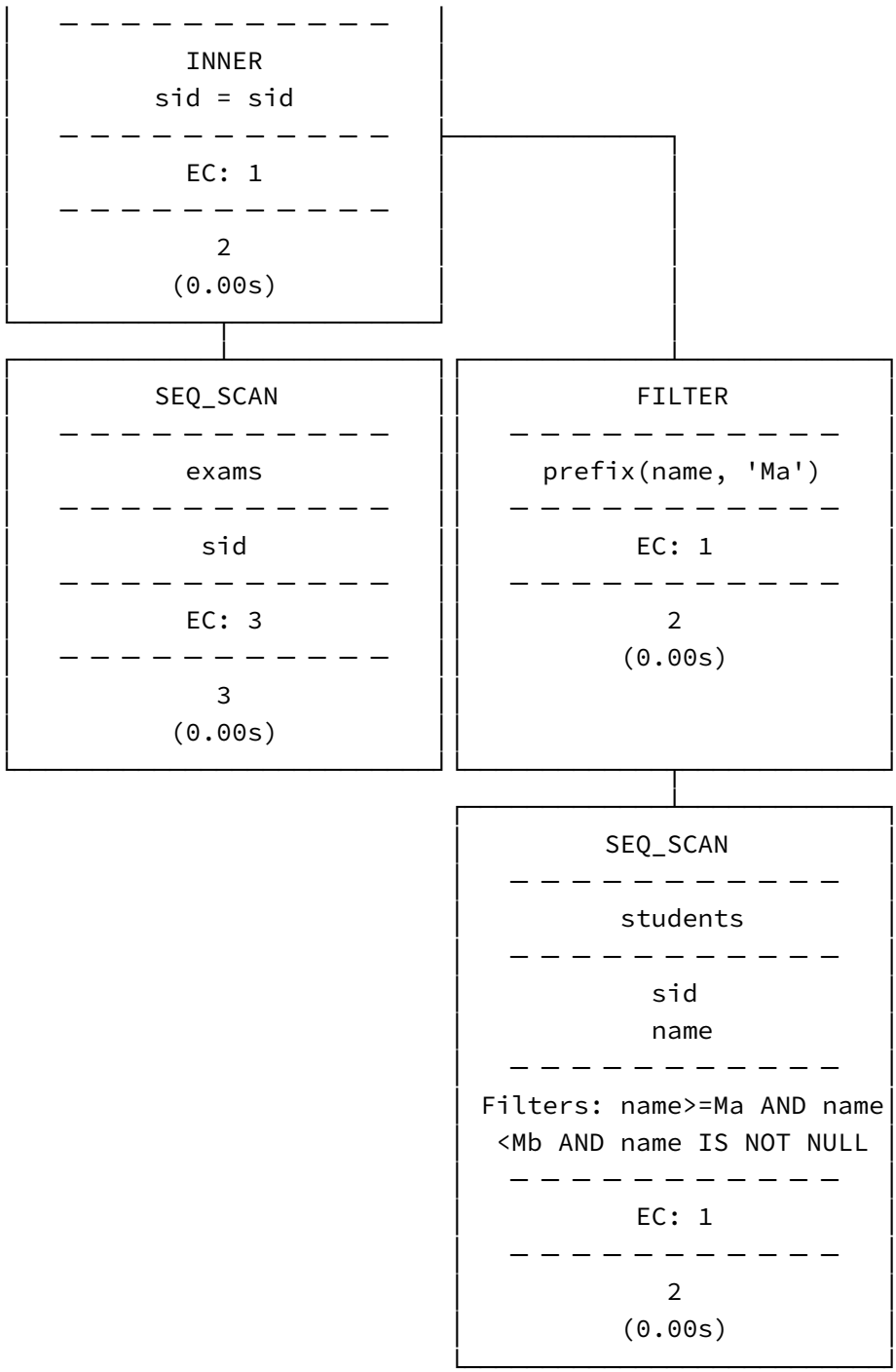
Note that the query is not actually executed – therefore, we can only see the estimated cardinality (EC) for each operator, which is calculated by using the statistics of the base tables and applying heuristics for each operator.

Run-Time Profiling The query plan helps understand the performance characteristics of the system. However, often it is also necessary to look at the performance numbers of individual operators and the cardinalities that pass through them. For this, you can create a query-profile graph.

To create the query graphs it is first necessary to gather the necessary data by running the query. In order to do that, we must first enable the run-time profiling. This can be done by prefixing the query with `EXPLAIN ANALYZE`:

```
EXPLAIN ANALYZE SELECT name FROM students JOIN exams USING (sid) WHERE name
↪  LIKE 'Ma%';
```





The output of `EXPLAIN ANALYZE` contains the estimated cardinality (EC), the actual cardinality, and the execution time for each operator.

It is also possible to save the query plan to a file, e.g., in JSON format:

```
-- All queries performed will be profiled, with output in json format.
```

```
-- By default the result is still printed to stdout.
PRAGMA enable_profiling='json';
-- Instead of writing to stdout, write the profiling output to a specific
 ↪  file on disk.
-- This has no effect for `EXPLAIN ANALYZE` queries, which will *always* be
-- returned as query results.
PRAGMA profile_output='/path/to/file.json';
```

> **Note.** This file is overwritten with each query that is issued. If you want to store the profile output for later it should be copied to a different file.

Now let us run the query that we inspected before:

```
SELECT name FROM students JOIN exams USING (sid) WHERE name LIKE 'Ma%';
```

After the query is completed, the JSON file containing the profiling output has been written to the specified file. We can then render the query graph using the Python script, provided we have the `duckdb` python module installed. This script will generate a HTML file and open it in your web browser.

```
python scripts/generate_querygraph.py /path/to/file.json
```

Testing

Testing is vital to make sure that DuckDB works properly and keeps working properly. For that reason, we put a large emphasis on thorough and frequent testing. We run a batch of small tests on every commit using GitHub Actions, and run a more exhaustive batch of tests on pull requests and commits in the master branch.

It is crucial that any new features that get added have correct tests that not only test the "happy path", but also test edge cases and incorrect usage of the feature. In this section, we describe how DuckDB tests are structured and how to make new tests for DuckDB.

The tests can be run by running the `unittest` program located in the `test` folder. For the default compilations this is located in either `build/release/test/unittest` (release) or `build/debug/test/unittest` (debug).

Writing Tests When testing DuckDB, we aim to route all the tests through SQL. We try to avoid testing components individually because that makes those components more difficult to change later on. As such, almost all of our tests can (and should) be expressed in pure SQL. There are certain exceptions to this, which we will discuss in the section "Catch Tests". However, in most cases you should write your tests in plain SQL.

SQL tests should be written using the SQLLogicTest framework.

C++ tests can be written using the Catch framework.

Client Connector Tests DuckDB also has tests for various client connectors. These are generally written in the relevant client language, and can be found in `tools/*/tests`. They also double as documentation of what should be doable from a given client.

SQLLogicTest

When testing DuckDB, we aim to route all the tests through SQL. We try to avoid testing components individually because that makes those components more difficult to change later on. As such, almost all of our tests can (and should) be expressed in pure SQL. There are certain exceptions to this, which we will discuss in the section "Catch Tests". However, in most cases you should write your tests in plain SQL.

For testing plain SQL we use an extended version of the SQL logic test suite, adopted from SQLite. Every test is a single self-contained file located in the `test/sql` directory. To run tests located outside of the default `test` directory, specify `--test-dir <root_directory>` and make sure provided test file paths are relative to that root directory.

The test describes a series of SQL statements, together with either the expected result, a `statement ok` indicator, or a `statement error` indicator. An example of a test file is shown below:

```
# name: test/sql/projection/test_simple_projection.test
# group [projection]

# enable query verification
statement ok
PRAGMA enable_verification

# create table
statement ok
CREATE TABLE a (i integer, j integer);

# insertion: 1 affected row
statement ok
INSERT INTO a VALUES (42, 84);

query II
SELECT * FROM a;
```

```
----
42	84
```

In this example, three statements are executed. The first statements are expected to succeed (prefixed by `statement ok`). The third statement is expected to return a single row with two columns (indicated by `query II`). The values of the row are expected to be 42 and 84 (separated by a tab character). For more information on query result verification, see the result verification section.

The top of every file should contain a comment describing the name and group of the test. The name of the test is always the relative file path of the file. The group is the folder that the file is in. The name and group of the test are relevant because they can be used to execute *only* that test in the unittest group. For example, if we wanted to execute *only* the above test, we would run the command `unittest test/sql/projection/test_simple_projection.test`. If we wanted to run all tests in a specific directory, we would run the command `unittest "[projection]"`.

Any tests that are placed in the `test` directory are automatically added to the test suite. Note that the extension of the test is significant. SQLLogicTests should either use the `.test` extension, or the `.test_slow` extension. The `.test_slow` extension indicates that the test takes a while to run, and will only be run when all tests are explicitly run using `unittest *`. Tests with the extension `.test` will be included in the fast set of tests.

Query Verification

Many simple tests start by enabling query verification. This can be done through the following PRAGMA statement:

```
statement ok
PRAGMA enable_verification
```

Query verification performs extra validation to ensure that the underlying code runs correctly. The most important part of that is that it verifies that optimizers do not cause bugs in the query. It does this by running both an unoptimized and optimized version of the query, and verifying that the results of these queries are identical.

Query verification is very useful because it not only discovers bugs in optimizers, but also finds bugs in e.g. join implementations. This is because the unoptimized version will typically run using cross products instead. Because of this, query verification can be very slow to do when working with larger data sets. It is therefore recommended to turn on query verification for all unit tests, except those involving larger data sets (more than 10-100~ rows).

Editors & Syntax Highlighting

The SQLLogicTests are not exactly an industry standard, but several other systems have adopted them as well. Parsing sqllogictests is intentionally simple. All statements have to be separated by empty lines. For that reason, writing a syntax highlighter is not extremely difficult.

A syntax highlighter exists for Visual Studio Code. We have also made a fork that supports the DuckDB dialect of the sqllogictests. You can use the fork by installing the original, then copying the `syntaxes/sqllogictest.tmLanguage.json` into the installed extension (on MacOS this is located in `~/.vscode/extensions/benesch.sqllogictest-0.1.1`).

A syntax highlighter is also available for CLion. It can be installed directly on the IDE by searching SQLTest on the marketplace. A GitHub repository is also available, with extensions and bug reports being welcome.

Temporary Files For some tests (e.g., CSV/Parquet file format tests) it is necessary to create temporary files. Any temporary files should be created in the temporary testing directory. This directory can be used by placing the string `__TEST_DIR__` in a query. This string will be replaced by the path of the temporary testing directory.

```
statement ok
COPY csv_data TO '__TEST_DIR__/output_file.csv.gz' (COMPRESSION GZIP);
```

Require & Extensions To avoid bloating the core system, certain functionality of DuckDB is available only as an extension. Tests can be build for those extensions by adding a `require` field in the test. If the extension is not loaded, any statements that occurs after the require field will be skipped. Examples of this are `require parquet` or `require icu`.

Another usage is to limit a test to a specific vector size. For example, adding `require vector_size 512` to a test will prevent the test from being run unless the vector size greater than or equal to 512. This is useful because certain functionality is not supported for low vector sizes, but we run tests using a vector size of 2 in our CI.

SQLLogicTest - Debugging

The purpose of the tests is to figure out when things break. Inevitably changes made to the system will cause one of the tests to fail, and when that happens the test needs to be debugged.

First, it is always recommended to run in debug mode. This can be done by compiling the system using the command `make debug`. Second, it is recommended to only run the test that breaks.

This can be done by passing the filename of the breaking test to the test suite as a command line parameter (e.g., `build/debug/test/unittest test/sql/projection/test_simple_projection.test`). For more options on running a subset of the tests see the Triggering which tests to run section.

After that, a debugger can be attached to the program and the test can be debugged. In the sqllogictests it is normally difficult to break on a specific query, however, we have expanded the test suite so that a function called `query_break` is called with the line number `line` as parameter for every query that is run. This allows you to put a conditional breakpoint on a specific query. For example, if we want to break on line number 43 of the test file we can create the following break point:

```
gdb: break query_break if line==43
lldb: break s -n query_break -c line==43
```

You can also skip certain queries from executing by placing `mode skip` in the file, followed by an optional `mode unskip`. Any queries between the two statements will not be executed.

Triggering Which Tests to Run

When running the unittest program, by default all the fast tests are run. A specific test can be run by adding the name of the test as an argument. For the SQLLogicTests, this is the relative path to the test file.

```
# run only a single test
build/debug/test/unittest test/sql/projection/test_simple_projection.test
```

All tests in a given directory can be executed by providing the directory as a parameter with square brackets.

```
# run all tests in the "projection" directory
build/debug/test/unittest "[projection]"
```

All tests, including the slow tests, can be run by running the tests with an asterisk.

```
# run all tests, including the slow tests
build/debug/test/unittest "*"
```

We can run a subset of the tests using the `--start-offset` and `--end-offset` parameters:

```
# run tests the tests 200..250
build/debug/test/unittest --start-offset=200 --end-offset=250
```

These are also available in percentages:

```
# run tests 10% - 20%
build/debug/test/unittest --start-offset-percentage=10
 ↪  --end-offset-percentage=20
```

The set of tests to run can also be loaded from a file containing one test name per line, and loaded using the `-f` command.

```
$ cat test.list
test/sql/join/full_outer/test_full_outer_join_issue_4252.test
test/sql/join/full_outer/full_outer_join_cache.test
test/sql/join/full_outer/test_full_outer_join.test
# run only the tests labeled in the file
$ build/debug/test/unittest -f test.list
```

SQLLogicTest - Result Verification

The standard way of verifying results of queries is using the `query` statement, followed by the letter `I` times the number of columns that are expected in the result. After the query, four dashes (`----`) are expected followed by the result values separated by tabs. For example,

```
query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
----
42  84
10  20
```

For legacy reasons the letters R and T are also accepted to denote columns.

NULL Values and Empty Strings

Empty lines have special significance for the SQLLogic test runner: they signify an end of the current statement or query. For that reason, empty strings and NULL values have special syntax that must be used in result verification. NULL values should use the string `NULL`, and empty strings should use the string `(empty)`, e.g.:

```
query II
SELECT NULL, ''
----
NULL
(empty)
```

Error Verification

In order to signify that an error is expected, the `statement error` indicator can be used. The `statement error` also takes an optional expected result - which is interpreted as the *expected error message*. Similar to `query`, the expected error should be placed after the four dashes (`----`) following the query. The test passes if the error message *contains* the text under `statement error` - the entire error message does not need to be provided. It is recommended that you only use a subset of the error message, so that the test does not break unnecessarily if the formatting of error messages is changed.

```
statement error
SELECT * FROM non_existent_table;
----
Table with name non_existent_table does not exist!
```

Regex

In certain cases result values might be very large or complex, and we might only be interested in whether or not the result *contains* a snippet of text. In that case, we can use the `<REGEX>:` modifier followed by a certain regex. If the result value matches the regex the test is passed. This is primarily used for query plan analysis.

```
query II
EXPLAIN SELECT tbl.a FROM "data/parquet-testing/arrow/alltypes_
 ↪  plain.parquet" tbl(a) WHERE a=1 OR a=2
----
physical_plan   <REGEX>:.*PARQUET_SCAN.*Filters: a=1 OR a=2.*
```

If we instead want the result *not* to contain a snippet of text, we can use the `<!REGEX>:` modifier.

File

As results can grow quite large, and we might want to re-use results over multiple files, it is also possible to read expected results from files using the `<FILE>` command. The expected result is read from the given file. As convention the file path should be provided as relative to the root of the GitHub repository.

```
query I
PRAGMA tpch(1)
----
<FILE>:extension/tpch/dbgen/answers/sf1/q01.csv
```

Row-Wise vs. Value-Wise Result Ordering

The result values of a query can be either supplied in row-wise order, with the individual values separated by tabs, or in value-wise order. In value wise order the individual *values* of the query must appear in row, column order each on an individual line. Consider the following example in both row-wise and value-wise order:

```
# row-wise
query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
----
42  84
10  20

# value-wise
query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
----
42
84
10
20
```

Hashes and Outputting Values

Besides direct result verification, the sqllogic test suite also has the option of using MD5 hashes for value comparisons. A test using hashes for result verification looks like this:

```
query I
SELECT g, STRING_AGG(x,',') FROM strings GROUP BY g
----
200 values hashing to b8126ea73f21372cdb3f2dc483106a12
```

This approach is useful for reducing the size of tests when results have many output rows. However, it should be used sparingly, as hash values make the tests more difficult to debug if they do break.

After it is ensured that the system outputs the correct result, hashes of the queries in a test file can be computed by adding `mode output_hash` to the test file. For example:

```
mode output_hash

query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
```

```
----
42  84
10  20
```

The expected output hashes for every query in the test file will then be printed to the terminal, as follows:

```
================================================================================
SQL Query
SELECT 42, 84 UNION ALL SELECT 10, 20;
================================================================================
4 values hashing to 498c69da8f30c24da3bd5b322a2fd455
================================================================================
```

In a similar manner, `mode output_result` can be used in order to force the program to print the result to the terminal for every query run in the test file.

Result Sorting

Queries can have an optional field that indicates that the result should be sorted in a specific manner. This field goes in the same location as the connection label. Because of that, connection labels and result sorting cannot be mixed.

The possible values of this field are `nosort`, `rowsort` and `valuesort`. An example of how this might be used is given below:

```
query I rowsort
SELECT 'world' UNION ALL SELECT 'hello'
----
hello
world
```

In general, we prefer not to use this field and rely on `ORDER BY` in the query to generate deterministic query answers. However, existing sqllogictests use this field extensively, hence it is important to know of its existence.

Query Labels

Another feature that can be used for result verification are `query labels`. These can be used to verify that different queries provide the same result. This is useful for comparing queries that are logically equivalent, but formulated differently. Query labels are provided after the connection label or sorting specifier.

Queries that have a query label do not need to have a result provided. Instead, the results of each of the queries with the same label are compared to each other. For example, the following script verifies that the queries `SELECT 42+1` and `SELECT 44-1` provide the same result:

```
query I nosort r43
SELECT 42+1;
----

query I nosort r43
SELECT 44-1;
----
```

SQLLogicTest - Persistent Testing

By default, all tests are run in in-memory mode (unless `--force-storage` is enabled). In certain cases, we want to force the usage of a persistent database. We can initiate a persistent database using the `load` command, and trigger a reload of the database using the `restart` command.

```
# load the DB from disk
load __TEST_DIR__/storage_scan.db

statement ok
CREATE TABLE test (a INTEGER);

statement ok
INSERT INTO test VALUES (11), (12), (13), (14), (15), (NULL)

# ...

restart

query I
SELECT * FROM test ORDER BY a
----
NULL
11
12
13
14
15
```

Note that by default the tests run with `SET wal_autocheckpoint='0KB'` - meaning a checkpoint is triggered after every statement. WAL tests typically run with the following settings to disable

this behavior:

```
statement ok
PRAGMA disable_checkpoint_on_shutdown

statement ok
PRAGMA wal_autocheckpoint='1TB';
```

SQLLogicTest - Loops

Loops can be used in sqllogictests when it is required to execute the same query many times but with slight modifications in constant values. For example, suppose we want to fire off 100 queries that check for the presence of the values `0..100` in a table:

```
# create the table integers with the values 0..100
statement ok
CREATE TABLE integers AS SELECT * FROM range(0, 100, 1) t1(i);

# verify individually that all 100 values are there
loop i 0 100

# execute the query, replacing the value
query I
SELECT COUNT(*) FROM integers WHERE i=${i};
----
1

# end the loop (note that multiple statements can be part of a loop)
endloop
```

Similarly, `foreach` can be used to iterate over a set of values.

```
foreach partcode millennium century decade year quarter month day hour
 ↪  minute second millisecond microsecond epoch

query III
SELECT i, DATE_PART('${partcode}', i) AS p, DATE_PART(['${partcode}'], i) AS
 ↪  st
FROM intervals
WHERE p <> st['${partcode}'];
----

endloop
```

`foreach` also has a number of preset combinations that should be used when required. In this manner, when new combinations are added to the preset, old tests will automatically pick up these new combinations.

Preset	Expansion
`<compression>`` |none uncompressed rle bitpacking dictionary fsst chimp patas| || `tinyint smallint integer bigint hugeint` | | `<unsigned>	`utinyint usmallint uinteger ubigint`
`<integral>`` || || `<integral> float double` | | `<alltypes>	`<numeric> bool interval varchar json`

> **Note.** Use large loops sparingly. Executing hundreds of thousands of SQL statements will slow down tests unnecessarily. Do not use loops for inserting data.

Data Generation without Loops

Loops should be used sparingly. While it might be tempting to use loops for inserting data using insert statements, this will considerably slow down the test cases. Instead, it is better to generate data using the built-in `range` and `repeat` functions.

```
-- create the table integers with the values [0, 1, .., 98,  99]
CREATE TABLE integers AS SELECT * FROM range(0, 100, 1) t1(i);

-- create the table strings with 100X the value "hello"
CREATE TABLE strings AS SELECT 'hello' AS s FROM range(0, 100, 1);
```

Using these two functions, together with clever use of cross products and other expressions, many different types of datasets can be efficiently generated. The `RANDOM()` function can also be used to

generate random data.

An alternative option is to read data from an existing CSV or Parquet file. There are several large CSV files that can be loaded from the directory `test/sql/copy/csv/data/real` using a `COPY INTO` statement or the `read_csv_auto` function.

The TPC-H and TPC-DS extensions can also be used to generate synthetic data, using e.g. `CALL dbgen(sf=1)` or `CALL dsdgen(sf=1)`.

SQLLogicTest - Multiple Connections

For tests whose purpose is to verify that the transactional management or versioning of data works correctly, it is generally necessary to use multiple connections. For example, if we want to verify that the creation of tables is correctly transactional, we might want to start a transaction and create a table in `con1`, then fire a query in `con2` that checks that the table is not accessible yet until committed.

We can use multiple connections in the sqllogictests using `connection labels`. The connection label can be optionally appended to any `statement` or `query`. All queries with the same connection label will be executed in the same connection. A test that would verify the above property would look as follows:

```
statement ok con1
BEGIN TRANSACTION

statement ok con1
CREATE TABLE integers(i INTEGER);

statement error con2
SELECT * FROM integers;
```

Concurrent Connections

Using connection modifiers on the statement and queries will result in testing of multiple connections, but all the queries will still be run *sequentially* on a single thread. If we want to run code from multiple connections *concurrently* over multiple threads, we can use the `concurrentloop` construct. The queries in `concurrentloop` will be run concurrently on separate threads at the same time.

```
concurrentloop i 0 10

statement ok
CREATE TEMP TABLE t2 AS (SELECT 1);
```

```
statement ok
INSERT INTO t2 VALUES (42);

statement ok
DELETE FROM t2

endloop
```

One caveat with `concurrentloop` is that results are often unpredictable - as multiple clients can hammer the database at the same time we might end up with (expected) transaction conflicts. `statement maybe` can be used to deal with these situations. `statement maybe` essentially accepts both a success, and a failure with a specific error message.

```
concurrentloop i 1 10

statement maybe
CREATE OR REPLACE TABLE t2 AS (SELECT
 ↪  -541240333865773480040026564265311535114 FROM t2 LIMIT 70%);
----
write-write conflict

endloop
```

Catch C/C++ Tests

While we prefer the sqllogic tests for testing most functionality, for certain tests only SQL is not sufficient. This typically happens when you want to test the C++ API. When using pure SQL is really not an option it might be necessary to make a C++ test using Catch.

Catch tests reside in the test directory as well. Here is an example of a catch test that tests the storage of the system:

```
#include "catch.hpp"
#include "test_helpers.hpp"

TEST_CASE("Test simple storage", "[storage]") {
    auto config = GetTestConfig();
    unique_ptr<QueryResult> result;
    auto storage_database = TestCreatePath("storage_test");

    // make sure the database does not exist
    DeleteDatabase(storage_database);
    {
```

```
        // create a database and insert values
        DuckDB db(storage_database, config.get());
        Connection con(db);
        REQUIRE_NO_FAIL(con.Query("CREATE TABLE test (a INTEGER, b
↪  INTEGER);"));
        REQUIRE_NO_FAIL(con.Query("INSERT INTO test VALUES (11, 22), (13,
↪  22), (12, 21), (NULL, NULL)"));
        REQUIRE_NO_FAIL(con.Query("CREATE TABLE test2 (a INTEGER);"));
        REQUIRE_NO_FAIL(con.Query("INSERT INTO test2 VALUES (13), (12),
↪  (11)"));
    }
    // reload the database from disk a few times
    for (idx_t i = 0; i < 2; i++) {
        DuckDB db(storage_database, config.get());
        Connection con(db);
        result = con.Query("SELECT * FROM test ORDER BY a");
        REQUIRE(CHECK_COLUMN(result, 0, {Value(), 11, 12, 13}));
        REQUIRE(CHECK_COLUMN(result, 1, {Value(), 22, 21, 22}));
        result = con.Query("SELECT * FROM test2 ORDER BY a");
        REQUIRE(CHECK_COLUMN(result, 0, {11, 12, 13}));
    }
    DeleteDatabase(storage_database);
}
```

The test uses the `TEST_CASE` wrapper to create each test. The database is created and queried using the C++ API. Results are checked using either `REQUIRE_FAIL`/`REQUIRE_NO_FAIL` (corresponding to statement ok and statement error) or `REQUIRE(CHECK_COLUMN(...))` (corresponding to query with a result check). Every test that is created in this way needs to be added to the corresponding `CMakeLists.txt`.

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.