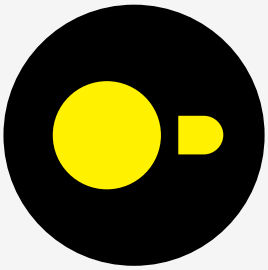
DuckDB Documentation

DuckDB version 0.10.3
Generated on 2024-06-03 at 11:25 UTC



Contents

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

Connect

Connect

Connect or Create a Database

To use DuckDB, you must first create a connection to a database. The exact syntax varies between the client APIs but it typically involves passing an argument to configure persistence.

Persistence

DuckDB can operate in both persistent mode, where the data is saved to disk, and in in-memory mode, where the entire data set is stored in the main memory.

Persistent Database

To create or open a persistent database, set the path of the database file, e.g., `my_database.duckdb`, when creating the connection. 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 an arbitrary extension, but `.db` or `.duckdb` are two common choices.

> **Tip.** Running on a persistent database allows spilling to disk, thus facilitating larger-than-memory workloads (i.e., out-of-core-processing).

Starting with v0.10, DuckDB's storage format is backwards-compatible, i.e., DuckDB is able to read database files produced by an older versions of DuckDB. For example, DuckDB v0.10 can read and operate on files created by the previous DuckDB version, v0.9. For more details on DuckDB's storage format, see the storage page.

In-Memory Database

DuckDB can operate in in-memory mode. In most clients, this can be activated by passing the special value `:memory:` as the database file or omitting the database file argument. In in-memory mode, no data is persisted to disk, therefore, all data is lost when the process finishes.

Concurrency

Handling Concurrency

DuckDB has two configurable options for concurrency:

1. One process can both read and write to the database.
2. Multiple processes can read from the database, but no processes can write (`access_mode = 'READ_ONLY'`).

When using option 1, DuckDB supports multiple writer threads using a combination of MVCC (Multi-Version Concurrency Control) and optimistic concurrency control (see Concurrency within a Single Process), but all within that single writer process. The reason for this concurrency model is to allow for the caching of data in RAM for faster analytical queries, rather than going back and forth to disk during each query. It also allows the caching of functions pointers, the database catalog, and other items so that subsequent queries on the same connection are faster.

> DuckDB is optimized for bulk operations, so executing many small transactions is not a primary design goal.

Concurrency within a Single Process

DuckDB supports concurrency within a single process according to the following rules. As long as there are no write conflicts, multiple concurrent writes will succeed. Appends will never conflict, even on the same table. Multiple threads can also simultaneously update separate tables or separate subsets of the same table. Optimistic concurrency control comes into play when two threads attempt to edit (update or delete) the same row at the same time. In that situation, the second thread to attempt the edit will fail with a conflict error.

Writing to DuckDB from Multiple Processes

Writing to DuckDB from multiple processes is not supported automatically and is not a primary design goal (see Handling Concurrency).

If multiple processes must write to the same file, several design patterns are possible, but would need to be implemented in application logic. For example, each process could acquire a cross-process mutex lock, then open the database in read/write mode and close it when the query is complete. Instead of using a mutex lock, each process could instead retry the connection if another process is already connected to the database (being sure to close the connection upon query completion). Another alternative would be to do multi-process transactions on a MySQL, PostgreSQL, or SQLite database, and use DuckDB's MySQL, PostgreSQL, or SQLite extensions to execute analytical queries on that data periodically.

Additional options include writing data to Parquet files and using DuckDB's ability to read multiple Parquet files, taking a similar approach with CSV files, or creating a web server to receive requests and manage reads and writes to DuckDB.

Optimistic Concurrency Control

DuckDB uses optimistic concurrency control, an approach generally considered to be the best fit for read-intensive analytical database systems as it speeds up read query processing. As a result any transactions that modify the same rows at the same time will cause a transaction conflict error:

```
Transaction conflict: cannot update a table that has been altered!
```

> **Tip.** A common workaround when a transaction conflict is encountered is to rerun the transaction.

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.

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

For a more detailed description, see the page on the `INSERT statement`.

CSV Loading

Data can be efficiently loaded from CSV files using several methods. The simplest is to use the CSV file's name:

```sql
SELECT * FROM 'test.csv';
```

Alternatively, use the `read_csv` function or the COPY statement to pass along options. For example:

```sql
SELECT * FROM read_csv('test.csv', header = false);
```

It is also possible to read data directly from **compressed CSV files** (e.g., compressed with gzip):

```sql
SELECT * FROM 'test.csv.gz';
```

DuckDB can create a table from the loaded data using the `CREATE TABLE ... AS SELECT` statement:

```sql
CREATE TABLE test AS
    SELECT * FROM 'test.csv';
```

For more details, see the page on CSV loading.

Parquet Loading

Parquet files can be efficiently loaded and queried using their filename:

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

Alternatively, use the `read_parquet` function or the COPY statement. For example:

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

For more details, see the page on Parquet loading.

JSON Loading

JSON files can be efficiently loaded and queried using their filename:

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

Alternatively, use the `read_json_auto` function or the COPY statement. For example:

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

For more details, see the page on JSON loading.

Appender

In several APIs (C, C++, Go, Java, and Rust), 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 using SQL statements.

CSV Files

CSV Import

Examples

The following examples use the `flights.csv` file.

Read a CSV file from disk, auto-infer options.

```
SELECT * FROM 'flights.csv';
```

Use the `read_csv` function 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 flights.csv | duckdb -c "SELECT * FROM read_csv('/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';
```

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 true, DELIMITER '|');
```

If we serialize the entire table, we can simply refer to it with its name.

```
COPY ontime TO 'flights.csv' WITH (HEADER true, 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 using the CSV sniffer. 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 `read_csv` function.

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`
`allow_quoted_nulls`	Option to allow the conversion of quoted values to NULL values	BOOL	`true`
`auto_detect`	Enables auto detection of CSV parameters.	BOOL	`true`
`auto_type_candidates`	This option allows you to specify the types that the sniffer will use when detecting CSV column types. The VARCHAR type is always included in the detected types (as a fallback option). See example.	TYPE[]	default types
`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 character 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	`"`
`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 NULLs.	VARCHAR[]	`[]`

Name	Description	Type	Default
`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 or (since v0.10.2) a list of strings that represent a `NULL` value.	`VARCHAR` or `VARCHAR[]`	(empty)
`parallel`	Whether or not the parallel CSV reader is used.	`BOOL`	`true`
`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`

`auto_type_candidates` Details

Usage example:

```
SELECT * FROM read_csv('csv_file.csv', auto_type_candidates = ['BIGINT', 'DATE']);
```

The default value for the `auto_type_candidates` option is `['SQLNULL', 'BOOLEAN', 'BIGINT', 'DOUBLE', 'TIME', 'DATE', 'TIMESTAMP', 'VARCHAR']`.

CSV Functions

The `read_csv` automatically attempts to figure out the correct configuration of the CSV reader using the CSV sniffer. 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('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` to create a persistent table as well:

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

column_name	column_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('flights.csv', sample_size = 20_000);
```

If we set `delim`/`sep`, `quote`, `escape`, or `header` explicitly, we can bypass the automatic detection of this particular parameter:

```
SELECT * FROM read_csv('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.

API Changes

> **Deprecated.** DuckDB v0.10.0 introduced breaking changes to the `read_csv` function. Namely, The `read_csv` function now attempts auto-detecting the CSV parameters, making its behavior identical to the old `read_csv_auto` function. If you would like to use `read_csv` with its old behavior, turn off the auto-detection manually by using `read_csv(..., auto_detect = false)`.

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). COPY detects the CSV's configuration options automatically.

```
CREATE TABLE ontime (
    flightdate DATE,
    uniquecarrier VARCHAR,
    origincityname VARCHAR,
    destcityname VARCHAR
);
COPY ontime FROM 'flights.csv';
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 manually specify the CSV format, we can do so using the configuration options of COPY.

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

Reading Faulty CSV Files

DuckDB supports reading erroneous CSV files. For details, see the Reading Faulty CSV Files page.

Limitations

The CSV reader only supports input files using UTF-8 character encoding. For CSV files using different encodings, use e.g., the `iconv` command-line tool to convert them to UTF-8.

CSV Auto Detection

When using `read_csv`, the system tries to automatically infer how to read the CSV file using the CSV sniffer. 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` with an explicit delimiter (e.g., `read_csv('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.

`sniff_csv` Function

It is possible to run the CSV sniffer as a separate step using the `sniff_csv(filename)` function, which returns the detected CSV properties as a table with a single row. The `sniff_csv` function accepts an optional `sample_size` parameter to configure the number of rows sampled.

```
FROM sniff_csv('my_file.csv');
FROM sniff_csv('my_file.csv', sample_size = 1000);
```

Column name	Description	Example
`Delimiter`	delimiter	`,`
`Quote`	quote character	`"`
`Escape`	escape	`\`
`NewLineDelimiter`	new-line delimiter	`\r\n`
`SkipRow`	number of rows skipped	`1`
`HasHeader`	whether the CSV has a header	`true`
`Columns`	column types encoded as a `LIST` of `STRUCT`s	`({'name': 'VARCHAR', 'age': 'BIGINT'})`
`DateFormat`	date Format	`%d/%m/%Y`
`TimestampFormat`	timestamp Format	`%Y-%m-%dT%H:%M:%S.%f`
`UserArguments`	arguments used to invoke `sniff_csv`	`sample_size = 1000`
`Prompt`	prompt ready to be used to read the CSV	`FROM read_csv('my_file.csv', auto_detect=false, delim=',', ...)`

Prompt

The `Prompt` column contains a SQL command with the configurations detected by the sniffer.

```
-- use line mode in CLI to get the full command
.mode line
SELECT Prompt FROM sniff_csv('my_file.csv');
```

```
Prompt = FROM read_csv('my_file.csv', auto_detect=false, delim=',', quote='"', escape='"', new_line='\n', skip=0, header=true, columns={...});
```

Detection Steps

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`. 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 = [INTEGER, 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': INTEGER}`).

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`
`%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`

Reading Faulty CSV Files

CSV files can come in all shapes and forms, with some presenting many errors that make the process of cleanly reading them inherently difficult. To help users read these files, DuckDB supports detailed error messages, the ability to skip faulty lines, and the possibility of storing faulty lines in a temporary table to assist users with a data cleaning step.

Structural Errors

DuckDB supports the detection and skipping of several different structural errors. In this section, we will go over each error with an example. For the examples, consider the following table:

```
CREATE TABLE people (name VARCHAR, birth_date DATE);
```

DuckDB detects the following error types:

- `CAST`: Casting errors occur when a column in the CSV file cannot be cast to the expected schema value. For example, the line `Pedro,The 90s` would cause an error since the string `The 90s` cannot be cast to a date.
- `MISSING COLUMNS`: This error occurs if a line in the CSV file has fewer columns than expected. In our example, we expect two columns; therefore, a row with just one value, e.g., `Pedro`, would cause this error.
- `TOO MANY COLUMNS`: This error occurs if a line in the CSV has more columns than expected. In our example, any line with more than two columns would cause this error, e.g., `Pedro,01-01-1992,pdet`.
- `UNQUOTED VALUE`: Quoted values in CSV lines must always be unquoted at the end; if a quoted value remains quoted throughout, it will cause an error. For example, assuming our scanner uses `quote='"'`, the line `"pedro"holanda, 01-01-1992` would present an unquoted value error.
- `LINE SIZE OVER MAXIMUM`: DuckDB has a parameter that sets the maximum line size a CSV file can have, which by default is set to `2,097,152` bytes. Assuming our scanner is set to `max_line_size = 25`, the line `Pedro Holanda, 01-01-1992` would produce an error, as it exceeds 25 bytes.
- `INVALID UNICODE`: DuckDB only supports UTF-8 strings; thus, lines containing non-UTF-8 characters will produce an error. For example, the line `pedro\xff\xff, 01-01-1992` would be problematic.

Anatomy of a CSV Error

By default, when performing a CSV read, if any structural errors are encountered, the scanner will immediately stop the scanning process and throw the error to the user. These errors are designed to provide as much information as possible to allow users to evaluate them directly in their CSV file.

This is an example for a full error message:

```
Conversion Error: CSV Error on Line: 5648
Original Line: Pedro,The 90s
Error when converting column "birth_date". date field value out of range: "The 90s", expected format is
(DD-MM-YYYY)

Column date is being converted as type DATE
This type was auto-detected from the CSV file.
Possible solutions:
* Override the type for this column manually by setting the type explicitly, e.g. types={'birth_date':
'VARCHAR'}
* Set the sample size to a larger value to enable the auto-detection to scan more values, e.g. sample_
size=-1
* Use a COPY statement to automatically derive types from an existing table.

  file= people.csv
  delimiter = , (Auto-Detected)
  quote = " (Auto-Detected)
  escape = " (Auto-Detected)
  new_line = \r\n (Auto-Detected)
  header = true (Auto-Detected)
  skip_rows = 0 (Auto-Detected)
  date_format = (DD-MM-YYYY) (Auto-Detected)
  timestamp_format =  (Auto-Detected)
  null_padding=0
  sample_size=20480
  ignore_errors=false
  all_varchar=0
```

The first block provides us with information regarding where the error occurred, including the line number, the original CSV line, and which field was problematic:

```
Conversion Error: CSV Error on Line: 5648
Original Line: Pedro,The 90s
Error when converting column "birth_date". date field value out of range: "The 90s", expected format is
(DD-MM-YYYY)
```

The second block provides us with potential solutions:

```
Column date is being converted as type DATE
This type was auto-detected from the CSV file.
Possible solutions:
* Override the type for this column manually by setting the type explicitly, e.g. types={'birth_date':
'VARCHAR'}
* Set the sample size to a larger value to enable the auto-detection to scan more values, e.g. sample_
size=-1
* Use a COPY statement to automatically derive types from an existing table.
```

Since the type of this field was auto-detected, it suggests defining the field as a VARCHAR or fully utilizing the dataset for type detection.

Finally, the last block presents some of the options used in the scanner that can cause errors, indicating whether they were auto-detected or manually set by the user.

Using the `ignore_errors` Option

There are cases where CSV files may have multiple structural errors, and users simply wish to skip these and read the correct data. Reading erroneous CSV files is possible by utilizing the `ignore_errors` option. With this option set, rows containing data that would otherwise cause the CSV parser to generate an error will be ignored. In our example, we will demonstrate a CAST error, but note that any of the errors described in our Structural Error section would cause the faulty line to be skipped.

For example, consider the following CSV file, `faulty.csv`:

```
Pedro,31
Oogie Boogie, three
```

If you read the CSV file, specifying that the first column is a VARCHAR and the second column is an `INTEGER`, loading the file would fail, as the string `three` cannot be converted to an `INTEGER`.

For example, the following query will throw a casting error.

```
FROM read_csv('faulty.csv', columns = {'name': 'VARCHAR', 'age': 'INTEGER'});
```

However, with `ignore_errors` set, the second row of the file is skipped, outputting only the complete first row. For example:

```
FROM read_csv(
    'faulty.csv',
    columns = {'name': 'VARCHAR', 'age': 'INTEGER'},
    ignore_errors = true
);
```

Outputs:

name	age
Pedro	31

One should note that the CSV Parser is affected by the projection pushdown optimization. Hence, if we were to select only the name column, both rows would be considered valid, as the casting error on the age would never occur. For example:

```
SELECT name
FROM read_csv('faulty.csv', columns = {'name': 'VARCHAR', 'age': 'INTEGER'});
```

Outputs:

name
Pedro
Oogie Boogie

Retrieving Faulty CSV Lines

Being able to read faulty CSV files is important, but for many data cleaning operations, it is also necessary to know exactly which lines are corrupted and what errors the parser discovered on them. For scenarios like these, it is possible to use DuckDB's CSV Rejects Table feature. By default, this feature creates two temporary tables.

1. `reject_scans`: Stores information regarding the parameters of the CSV Scanner
2. `reject_errors`: Stores information regarding each CSV faulty line and in which CSV Scanner they happened.

Note that any of the errors described in our Structural Error section will be stored in the rejects tables. Also, if a line has multiple errors, multiple entries will be stored for the same line, one for each error.

Reject Scans

The CSV Reject Scans Table returns the following information:

Column name	Description	Type
`scan_id`	The internal ID used in DuckDB to represent that scanner	`UBIGINT`
`file_id`	A scanner might happen over multiple files, so the file_id represents a unique file in a scanner	`UBIGINT`
`file_path`	The file path	`VARCHAR`
`delimiter`	The delimiter used e.g., ;	`VARCHAR`
`quote`	The quote used e.g., ”	`VARCHAR`
`escape`	The quote used e.g., ”	`VARCHAR`
`newline_delimiter`	The newline delimiter used e.g., \r\n	`VARCHAR`
`skip_rows`	If any rows were skipped from the top of the file	`UINTEGER`
`has_header`	If the file has a header	`BOOLEAN`
`columns`	The schema of the file (i.e., all column names and types)	`VARCHAR`
`date_format`	The format used for date types	`VARCHAR`
`timestamp_format`	The format used for timestamp types	`VARCHAR`
`user_arguments`	Any extra scanner parameters manually set by the user	`VARCHAR`

Reject Errors

The CSV Reject Errors Table returns the following information:

Column name	Description	Type
`scan_id`	The internal ID used in DuckDB to represent that scanner, used to join with reject scans tables	`UBIGINT`
`file_id`	The file_id represents a unique file in a scanner, used to join with reject scans tables	`UBIGINT`
`line`	Line number, from the CSV File, where the error occured.	`UBIGINT`
`line_byte_position`	Byte Position of the start of the line, where the error occured.	`UBIGINT`
`byte_position`	Byte Position where the error occured.	`UBIGINT`
`column_idx`	If the error happens in a specific column, the index of the column.	`UBIGINT`
`column_name`	If the error happens in a specific column, the name of the column.	`VARCHAR`
`error_type`	The type of the error that happened.	`ENUM`
`csv_line`	The original CSV line.	`VARCHAR`
`error_message`	The error message produced by DuckDB.	`VARCHAR`

Parameters

The parameters listed below are used in the `read_csv` function to configure the CSV Rejects Table.

Name	Description	Type	Default
`store_rejects`	If set to true, any errors in the file will be skipped and stored in the default rejects temporary tables.	`BOOLEAN`	False
`rejects_scan`	Name of a temporary table where the information of the scan information of faulty CSV file are stored.	`VARCHAR`	reject_scans
`rejects_table`	Name of a temporary table where the information of the faulty lines of a CSV file are stored.	`VARCHAR`	reject_errors
`rejects_limit`	Upper limit on the number of faulty records from a CSV file that will be recorded in the rejects table. 0 is used when no limit should be applied.	`BIGINT`	0

To store the information of the faulty CSV lines in a rejects table, the user must simply set the `store_rejects` option to true. For example:

```
FROM read_csv(
    'faulty.csv',
    columns = {'name': 'VARCHAR', 'age': 'INTEGER'},
    store_rejects = true
);
```

You can then query both the `reject_scans` and `reject_errors` tables, to retrieve information about the rejected tuples. For example:

```
FROM reject_scans;
```

Outputs:

scan_id	file_id	file_path	delimiter	quote	escape	newline_delimiter	skip_rows	has_header	columns	date_format	timestamp_format	user_arguments
5	0	faulty.csv	,	”	”	\n	0	false	{'name': 'VARCHAR','age': 'INTEGER'}			store_rejects=true

```
FROM reject_errors;
```

Outputs:

scan_id	file_id	line	line_byte_position	byte_position	column_idx	column_name	error_type	csv_line	error_message
5	0	2	10	23	2	age	CAST	Oogie Boogie, three	Error when converting column ”age”. Could not convert string ” three” to 'INTEGER'

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('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('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('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';
```

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

JSON Files

JSON Loading

Examples

Read a JSON file from disk, auto-infer options.

```
SELECT * FROM 'todos.json';
```

Use the `read_json` function 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
`auto_detect`	Whether to auto-detect detect the names of the keys and data types of the values automatically	`BOOL`	`false`
`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)`
`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 `'uncompressed'`, `'gzip'`, `'zstd'`, and `'auto_detect'`.	`VARCHAR`	`'auto_detect'`
`convert_strings_to_integers`	Whether strings representing integer values should be converted to a numerical type.	`BOOL`	`false`
`dateformat`	Specifies the date format to use when parsing dates. See Date Format	`VARCHAR`	`'iso'`
`filename`	Whether or not an extra `filename` column should be included in the result.	`BOOL`	`false`
`format`	Can be one of `['auto', 'unstructured', 'newline_delimited', 'array']`	`VARCHAR`	`'array'`
`hive_partitioning`	Whether or not to interpret the path as a Hive partitioned path.	`BOOL`	`false`
`ignore_errors`	Whether to ignore parse errors (only possible when `format` is `'newline_delimited'`)	`BOOL`	`false`
`maximum_depth`	Maximum nesting depth to which the automatic schema detection detects types. Set to -1 to fully detect nested JSON types	`BIGINT`	`-1`
`maximum_object_size`	The maximum size of a JSON object (in bytes)	`UINTEGER`	`16777216`
`records`	Can be one of `['auto', 'true', 'false']`	`VARCHAR`	`'records'`
`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`
`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`

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

For more details, see the page on the COPY statement.

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(['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(['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 two 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 three 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(['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

Since v0.10.2, it is possible to specify the name of the filename column using the `filename` argument:

```
SELECT * FROM read_csv(['flights1.csv', 'flights2.csv'], union_by_name = true, filename = "my_filename_
column");
```

FlightDate	UniqueCarrier	OriginCityName	DestCityName	my_filename_column
1988-01-01	AA	New York, NY	Los Angeles, CA	flights1.csv
1988-01-02	AA	New York, NY	Los Angeles, CA	flights1.csv
1988-01-03	AA	New York, NY	Los Angeles, CA	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
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('flights*.csv');
```

Read a set of CSV files combining columns by name:

```
SELECT * FROM read_csv('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(['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 three 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 two specific folders:

```
SELECT * FROM read_parquet(['folder1/*.parquet', 'folder2/*.parquet']);
```

Read over https:

```
SELECT * FROM read_parquet('https://some.url/some_file.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 using the default compression (Snappy):

```
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(100_000))
    TO 'test.parquet'
    (FORMAT 'parquet', COMPRESSION 'zstd', ROW_GROUP_SIZE 100_000);
```

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.

> Parquet data sets differ based on the number of files, the size of individual files, the compression algorithm used row group size, etc. These have a significant effect on performance. Please consult the Performance Guide for details.

`read_parquet` Function

Function	Description	Example
`read_parquet(path_or_list_of_paths)`	Read Parquet file(s)	`SELECT * FROM read_parquet('test.parquet');`
`parquet_scan(path_or_list_of_paths)`	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`
`encryption_config`	Configuration for Parquet encryption.	STRUCT	-
`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`

Name	Description	Type	Default
`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(100_000))
    TO 'row-groups-zstd.parquet'
    (FORMAT PARQUET, COMPRESSION ZSTD, ROW_GROUP_SIZE 100_000);
```

> LZ4 compression is currently only available in the nightly and source builds:

Write a CSV file to an `LZ4_RAW`-compressed Parquet file:

```
COPY
    (FROM generate_series(100_000))
    TO 'result-lz4.parquet'
    (FORMAT PARQUET, COMPRESSION LZ4);
```

Or:

```
COPY
    (FROM generate_series(100_000))
    TO 'result-lz4.parquet'
    (FORMAT PARQUET, COMPRESSION LZ4_RAW);
```

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);
```

Encryption

DuckDB supports reading and writing encrypted Parquet files.

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:

```
INSTALL parquet;
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:

```sql
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
compression	VARCHAR
encodings	VARCHAR
index_page_offset	BIGINT
dictionary_page_offset	BIGINT
data_page_offset	BIGINT
total_compressed_size	BIGINT
total_uncompressed_size	BIGINT
key_value_metadata	MAP(BLOB, BLOB)

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:

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

Fetch the internal schema of a Parquet file:

```sql
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
precision	BIGINT
field_id	BIGINT
logical_type	VARCHAR

Parquet File Metadata

The `parquet_file_metadata` function can be used to query file-level metadata such as the format version and the encryption algorithm used:

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

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

Field	Type
file_name	VARCHAR
created_by	VARCHAR
num_rows	BIGINT
num_row_groups	BIGINT
format_version	BIGINT
encryption_algorithm	VARCHAR
footer_signing_key_metadata	VARCHAR

Parquet Key-Value Metadata

The `parquet_kv_metadata` function can be used to query custom metadata defined as key-value pairs:

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

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

Field	Type
file_name	VARCHAR
key	BLOB
value	BLOB

Parquet Encryption

Starting with version 0.10.0, DuckDB supports reading and writing encrypted Parquet files. DuckDB broadly follows the Parquet Modular Encryption specification with some limitations.

Reading and Writing Encrypted Files

Using the PRAGMA `add_parquet_key` function, named encryption keys of 128, 192, or 256 bits can be added to a session. These keys are stored in-memory:

```
PRAGMA add_parquet_key('key128', '0123456789112345');
PRAGMA add_parquet_key('key192', '012345678911234501234567');
PRAGMA add_parquet_key('key256', '0123456789112345012345678911234 5');
```

Writing Encrypted Parquet Files

After specifying the key (e.g., `key256`), files can be encrypted as follows:

```
COPY tbl TO 'tbl.parquet' (ENCRYPTION_CONFIG {footer_key: 'key256'});
```

Reading Encrpyted Parquet Files

An encrypted Parquet file using a specific key (e.g., `key256`), can then be read as follows:

```
COPY tbl FROM 'tbl.parquet' (ENCRYPTION_CONFIG {footer_key: 'key256'});
```

Or:

```
SELECT *
FROM read_parquet('tbl.parquet', encryption_config = {footer_key: 'key256'});
```

Limitations

DuckDB's Parquet encryption currently has the following limitations.

1. It is not compatible with the encryption of, e.g., PyArrow, until the missing details are implemented.
2. DuckDB encrypts the footer and all columns using the `footer_key`. The Parquet specification allows encryption of individual columns with different keys, e.g.:

   ```
   COPY tbl TO 'tbl.parquet'
       (ENCRYPTION_CONFIG {
           footer_key: 'key256',
           column_keys: {key256: ['col0', 'col1']}
       });
   ```

   However, this is unsupported at the moment and will cause an error to be thrown (for now):

   ```
   Not implemented Error: Parquet encryption_config column_keys not yet implemented
   ```

Performance Implications

Note that encryption has some performance implications. Without encryption, reading/writing the `lineitem` table from TPC-H at SF1, which is 6M rows and 15 columns, from/to a Parquet file takes 0.26 and 0.99 seconds, respectively. With encryption, this takes 0.64 and 2.21 seconds, both approximately 2.5× slower than the unencrypted version.

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(10_000_000))
    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 122,880. 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.

To write a query to a Parquet file with a different row group size, run:

```
COPY
    (FROM generate_series(100_000))
    TO 'row-groups.parquet'
    (FORMAT PARQUET, ROW_GROUP_SIZE 100_000);
```

See the Performance Guide on file formats for more tips.

Partitioning

Hive Partitioning

Examples

Read data from a Hive partitioned data set:

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

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 = true);
```

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 = true)
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:

```
SELECT *
FROM read_parquet(
    'dir/**/*.parquet',
    hive_partitioning = true,
    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.

> **Best practice.** 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 Appender can be used to load bulk data into a DuckDB database. It is currently available in the C, C++, Go, Java, and Rust APIs. 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.

In the C++ API, the Appender works as follows:

```
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 approach)
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)
);
```

Commit Frequency

By default, the appender performs a commits every 204,800 rows. You can change this by explicitly using transactions and surrounding your batches of AppendRow calls by `BEGIN TRANSACTION` and `COMMIT` statements.

Handling Constraint Violations

If the Appender encounters a `PRIMARY KEY` conflict or a `UNIQUE` constraint violation, it fails and returns the following error:

```
Constraint Error: PRIMARY KEY or UNIQUE constraint violated: duplicate key "..."
```

In this case, the entire append operation fails and no rows are inserted.

Appender Support in Other Clients

The Appender is also available in the following client APIs:

- C
- Go
- JDBC (Java)
- Rust

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.

> **Best practice.** 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.

> **Tip.** 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');
```

For a more detailed description together with syntax diagram can be found, see the page on the `INSERT statement`.

Client APIs

Client APIs Overview

There are various client APIs for DuckDB:

- C
- C++
- Go by marcboeker
- Java
- Julia
- Node.js
- Python
- R
- Rust
- WebAssembly/Wasm
- ADBC API
- ODBC API

Additionally, there is a standalone Command Line Interface (CLI) client.

There are also contributed third-party DuckDB wrappers, which currently do not have an official documentation page:

- C# by Giorgi
- Common Lisp by ak-coram
- Crystal by amauryt
- Ruby by suketa
- Zig by karlseguin

C

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.

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);
duckdb_query_progress_type 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. The instantiated database should be closed with 'duckdb_close'.

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` or `duckdb_open_ext`. 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 interrupt

`duckdb_query_progress`

Get progress of the running query

Syntax

```
duckdb_query_progress_type 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(

);
```

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`. The duckdb_config must be destroyed using 'duckdb_destroy_config'

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 object and de-allocates all memory allocated for the object.

Syntax

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

Parameters

- `config`

The configuration object to destroy.

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.

> 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");
}
```

> **Warning.** 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_statement_type duckdb_result_statement_type(duckdb_result result);
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 to 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_result_statement_type`

Returns the statement type of the statement that was executed

Syntax

```
duckdb_statement_type duckdb_result_statement_type(
  duckdb_result result
);
```

Parameters

- `result`

The result object to fetch the statement type from.

- `returns`

duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

`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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Returns the number of rows present in 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.

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);
duckdb_vector duckdb_array_vector_get_child(duckdb_vector vector);
```

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.

Note that the result must be destroyed with `duckdb_destroy_data_chunk`.

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. You may also use this function to assign BLOBs.

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_array_vector_get_child

Retrieves the child vector of a array vector.

The resulting vector is valid as long as the parent vector is valid. The resulting vector has the size of the parent vector multiplied by the array size.

Syntax

```
duckdb_vector duckdb_array_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector

- `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

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);
duckdb_value duckdb_create_struct_value(duckdb_logical_type type, duckdb_value *values);
duckdb_value duckdb_create_list_value(duckdb_logical_type type, duckdb_value *values, idx_t value_count);
duckdb_value duckdb_create_array_value(duckdb_logical_type type, duckdb_value *values, idx_t value_count);
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_create_struct_value

Creates a struct value from a type and an array of values

Syntax

```
duckdb_value duckdb_create_struct_value(
  duckdb_logical_type type,
  duckdb_value *values
);
```

Parameters

- `type`

The type of the struct

- `values`

The values for the struct fields

- `returns`

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

duckdb_create_list_value

Creates a list value from a type and an array of values of length `value_count`

Syntax

```
duckdb_value duckdb_create_list_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`

The type of the list

- `values`

The values for the list

- `value_count`

The number of values in the list

- `returns`

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

duckdb_create_array_value

Creates a array value from a type and an array of values of length `value_count`

Syntax

```
duckdb_value duckdb_create_array_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`

The type of the array

- `values`

The values for the array

- `value_count`

The number of values in the array

- `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

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 (`INTEGER[]`). 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 = 0,
  DUCKDB_TYPE_BOOLEAN = 1,
  DUCKDB_TYPE_TINYINT = 2,
  DUCKDB_TYPE_SMALLINT = 3,
  DUCKDB_TYPE_INTEGER = 4,
  DUCKDB_TYPE_BIGINT = 5,
  DUCKDB_TYPE_UTINYINT = 6,
  DUCKDB_TYPE_USMALLINT = 7,
  DUCKDB_TYPE_UINTEGER = 8,
  DUCKDB_TYPE_UBIGINT = 9,
  DUCKDB_TYPE_FLOAT = 10,
  DUCKDB_TYPE_DOUBLE = 11,
  DUCKDB_TYPE_TIMESTAMP = 12,
  DUCKDB_TYPE_DATE = 13,
  DUCKDB_TYPE_TIME = 14,
  DUCKDB_TYPE_INTERVAL = 15,
  DUCKDB_TYPE_HUGEINT = 16,
  DUCKDB_TYPE_UHUGEINT = 32,
  DUCKDB_TYPE_VARCHAR = 17,
  DUCKDB_TYPE_BLOB = 18,
  DUCKDB_TYPE_DECIMAL = 19,
  DUCKDB_TYPE_TIMESTAMP_S = 20,
  DUCKDB_TYPE_TIMESTAMP_MS = 21,
  DUCKDB_TYPE_TIMESTAMP_NS = 22,
  DUCKDB_TYPE_ENUM = 23,
  DUCKDB_TYPE_LIST = 24,
  DUCKDB_TYPE_STRUCT = 25,
  DUCKDB_TYPE_MAP = 26,
  DUCKDB_TYPE_ARRAY = 33,
  DUCKDB_TYPE_UUID = 27,
```

```
  DUCKDB_TYPE_UNION = 28,
  DUCKDB_TYPE_BIT = 29,
  DUCKDB_TYPE_TIME_TZ = 30,
  DUCKDB_TYPE_TIMESTAMP_TZ = 31,
} 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.

> `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);
duckdb_result_type duckdb_result_return_type(duckdb_result result);
```

Date/Time/Timestamp Helpers

```
duckdb_date_struct duckdb_from_date(duckdb_date date);
duckdb_date duckdb_to_date(duckdb_date_struct date);
bool duckdb_is_finite_date(duckdb_date date);
duckdb_time_struct duckdb_from_time(duckdb_time time);
duckdb_time_tz duckdb_create_time_tz(int64_t micros, int32_t offset);
duckdb_time_tz_struct duckdb_from_time_tz(duckdb_time_tz micros);
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);
bool duckdb_is_finite_timestamp(duckdb_timestamp ts);
```

Hugeint Helpers

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

Decimal Helpers

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

Logical Type Interface

```
duckdb_logical_type duckdb_create_logical_type(duckdb_type type);
char *duckdb_logical_type_get_alias(duckdb_logical_type type);
duckdb_logical_type duckdb_create_list_type(duckdb_logical_type type);
duckdb_logical_type duckdb_create_array_type(duckdb_logical_type type, idx_t array_size);
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_enum_type(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_array_type_child_type(duckdb_logical_type type);
idx_t duckdb_array_type_array_size(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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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_result_return_type`

Returns the return_type of the given result, or DUCKDB_RETURN_TYPE_INVALID on error

Syntax

```
duckdb_result_type duckdb_result_return_type(
  duckdb_result result
);
```

Parameters

- `result`

The result object

- `returns`

The return_type

`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_is_finite_date`

Test a `duckdb_date` to see if it is a finite value.

Syntax

```
bool duckdb_is_finite_date(
  duckdb_date date
);
```

Parameters

- `date`

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

- `returns`

True if the date is finite, false if it is ±infinity.

`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_create_time_tz

Create a duckdb_time_tz object from micros and a timezone offset.

Syntax

```
duckdb_time_tz duckdb_create_time_tz(
  int64_t micros,
  int32_t offset
);
```

Parameters

- micros

The microsecond component of the time.

- offset

The timezone offset component of the time.

- returns

The duckdb_time_tz element.

duckdb_from_time_tz

Decompose a TIME_TZ objects into micros and a timezone offset.

Use duckdb_from_time to further decompose the micros into hour, minute, second and microsecond.

Syntax

```
duckdb_time_tz_struct duckdb_from_time_tz(
  duckdb_time_tz micros
);
```

Parameters

- micros

The time object, as obtained from a DUCKDB_TYPE_TIME_TZ column.

- out_micros

The microsecond component of the time.

- out_offset

The timezone offset component of the time.

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_is_finite_timestamp

Test a duckdb_timestamp to see if it is a finite value.

Syntax

```
bool duckdb_is_finite_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- ts

The timestamp object, as obtained from a DUCKDB_TYPE_TIMESTAMP column.

- returns

True if the timestamp is finite, false if it is ±infinity.

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_logical_type_get_alias

Returns the alias of a duckdb_logical_type, if one is set, else NULL. The result must be destroyed with duckdb_free.

Syntax

```
char *duckdb_logical_type_get_alias(
  duckdb_logical_type type
);
```

Parameters

- type

The logical type to return the alias of

- returns

The alias or NULL

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_array_type`

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

Syntax

```
duckdb_logical_type duckdb_create_array_type(
  duckdb_logical_type type,
  idx_t array_size
);
```

Parameters

- `type`

The child type of array type to create.

- `array_size`

The number of elements in the array.

- `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_enum_type

Creates an ENUM type from the passed member name array. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

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

Parameters

- `enum_name`

The name of the enum.

- `member_names`

The array of names that the enum should consist of.

- `member_count`

The number of elements that were specified in the array.

- `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 enum 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_array_type_child_type`

Retrieves the child type of the given array type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_array_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

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

`duckdb_array_type_array_size`

Retrieves the array size of the given array type.

Syntax

```
idx_t duckdb_array_type_array_size(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The fixed number of elements the values of this array type can store.

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.

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.

> **Warning.** 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_statement_type duckdb_prepared_statement_type(duckdb_prepared_statement statement);
```

`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_prepared_statement_type`

Returns the statement type of the statement to be executed

Syntax

```
duckdb_statement_type duckdb_prepared_statement_type(
  duckdb_prepared_statement statement
);
```

Parameters

- `statement`

The prepared statement.

- `returns`

duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

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

```c
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

```c
duckdb_state duckdb_appender_create(duckdb_connection connection, const char *schema, const char *table,
duckdb_appender *out_appender);
idx_t duckdb_appender_column_count(duckdb_appender appender);
duckdb_logical_type duckdb_appender_column_type(duckdb_appender appender, idx_t col_idx);
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_uhugeint(duckdb_appender appender, duckdb_uhugeint 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.

Note that the object must be destroyed with `duckdb_appender_destroy`.

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_column_count

Returns the number of columns in the table that belongs to the appender.

- appender The appender to get the column count from.

Syntax

```
idx_t duckdb_appender_column_count(
  duckdb_appender appender
);
```

Parameters

- `returns`

The number of columns in the table.

`duckdb_appender_column_type`

Returns the type of the column at the specified index.

Note: The resulting type should be destroyed with `duckdb_destroy_logical_type`.

- appender The appender to get the column type from.
- col_idx The index of the column to get the type of.

Syntax

```
duckdb_logical_type duckdb_appender_column_type(
  duckdb_appender appender,
  idx_t col_idx
);
```

Parameters

- `returns`

The duckdb_logical_type of the column.

`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. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. It is not possible to append more values. Call duckdb_appender_error to obtain the error message followed by duckdb_appender_destroy to destroy the invalidated 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`

Closes the appender by flushing all intermediate states and closing it for further appends. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. Call duckdb_appender_error to obtain the error message followed by duckdb_appender_destroy to destroy the invalidated appender.

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`

Closes the appender by flushing all intermediate states to the table and destroying it. By destroying it, this function de-allocates all memory associated with the appender. If flushing the data triggers a constraint violation, then all data is invalidated, and this function returns DuckDBError. Due to the destruction of the appender, it is no longer possible to obtain the specific error message with duckdb_appender_error. Therefore, call duckdb_appender_close before destroying the appender, if you need insights into the specific error.

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_uhugeint

Append a duckdb_uhugeint value to the appender.

Syntax

```
duckdb_state duckdb_append_uhugeint(
  duckdb_appender appender,
  duckdb_uhugeint 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.

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

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. 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

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);
duckdb_query_progress_type 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_statement_type duckdb_result_statement_type(duckdb_result result);
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);
duckdb_result_type duckdb_result_return_type(duckdb_result result);
```

Safe fetch functions

```
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_uhugeint duckdb_value_uhugeint(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);
duckdb_string duckdb_value_string(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);
bool duckdb_is_finite_date(duckdb_date date);
duckdb_time_struct duckdb_from_time(duckdb_time time);
duckdb_time_tz duckdb_create_time_tz(int64_t micros, int32_t offset);
duckdb_time_tz_struct duckdb_from_time_tz(duckdb_time_tz micros);
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);
bool duckdb_is_finite_timestamp(duckdb_timestamp ts);
```

Hugeint Helpers

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

Unsigned Hugeint Helpers

```
double duckdb_uhugeint_to_double(duckdb_uhugeint val);
duckdb_uhugeint duckdb_double_to_uhugeint(double val);
```

Decimal Helpers

```
duckdb_decimal duckdb_double_to_decimal(double val, uint8_t width, uint8_t scale);
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_statement_type duckdb_prepared_statement_type(duckdb_prepared_statement statement);
```

Bind Values to Prepared Statements

```
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_uhugeint(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_
uhugeint 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);
```

Execute Prepared Statements

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

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_pending_state duckdb_pending_execute_check_state(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);
duckdb_value duckdb_create_struct_value(duckdb_logical_type type, duckdb_value *values);
duckdb_value duckdb_create_list_value(duckdb_logical_type type, duckdb_value *values, idx_t value_count);
duckdb_value duckdb_create_array_value(duckdb_logical_type type, duckdb_value *values, idx_t value_count);
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);
char *duckdb_logical_type_get_alias(duckdb_logical_type type);
duckdb_logical_type duckdb_create_list_type(duckdb_logical_type type);
duckdb_logical_type duckdb_create_array_type(duckdb_logical_type type, idx_t array_size);
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_enum_type(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_array_type_child_type(duckdb_logical_type type);
idx_t duckdb_array_type_array_size(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);
duckdb_vector duckdb_array_vector_get_child(duckdb_vector vector);
```

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);
idx_t duckdb_appender_column_count(duckdb_appender appender);
duckdb_logical_type duckdb_appender_column_type(duckdb_appender appender, idx_t col_idx);
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_uhugeint(duckdb_appender appender, duckdb_uhugeint 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);
void duckdb_result_arrow_array(duckdb_result result, duckdb_data_chunk chunk, duckdb_arrow_array *out_
array);
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);
void duckdb_destroy_arrow_stream(duckdb_arrow_stream *stream_p);
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);
```

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_data_chunk duckdb_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. The instantiated database should be closed with 'duckdb_close'.

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` or `duckdb_open_ext`. 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 interrupt

duckdb_query_progress

Get progress of the running query

Syntax

```
duckdb_query_progress_type 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`. The duckdb_config must be destroyed using 'duckdb_destroy_config'

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 object 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 to 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_result_statement_type`

Returns the statement type of the statement that was executed

Syntax

```
duckdb_statement_type duckdb_result_statement_type(
  duckdb_result result
);
```

Parameters

- `result`

The result object to fetch the statement type from.

- `returns`

duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

`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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Returns the number of rows present in 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

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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_result_return_type

Returns the return_type of the given result, or DUCKDB_RETURN_TYPE_INVALID on error

Syntax

```
duckdb_result_type duckdb_result_return_type(
  duckdb_result result
);
```

Parameters

- `result`

The result object

- `returns`

The return_type

duckdb_value_boolean

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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_uhugeint`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Syntax

```
duckdb_uhugeint duckdb_value_uhugeint(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- `returns`

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

`duckdb_value_decimal`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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_string

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Syntax

```
duckdb_string duckdb_value_string(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

Parameters

- returns

The string value at the specified location. Attempts to cast the result value to string.

- No support for nested types, and for other complex types.
- The resulting field "string.data" 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

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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 field "blob.data" must be freed with `duckdb_free`.

duckdb_value_is_null

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`, `duckdb_value_blob`, or `duckdb_value_string`.

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_is_finite_date

Test a duckdb_date to see if it is a finite value.

Syntax

```
bool duckdb_is_finite_date(
  duckdb_date date
);
```

Parameters

- date

The date object, as obtained from a DUCKDB_TYPE_DATE column.

- returns

True if the date is finite, false if it is ±infinity.

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_create_time_tz

Create a duckdb_time_tz object from micros and a timezone offset.

Syntax

```
duckdb_time_tz duckdb_create_time_tz(
  int64_t micros,
  int32_t offset
);
```

Parameters

- `micros`

The microsecond component of the time.

- `offset`

The timezone offset component of the time.

- `returns`

The `duckdb_time_tz` element.

`duckdb_from_time_tz`

Decompose a TIME_TZ objects into micros and a timezone offset.

Use `duckdb_from_time` to further decompose the micros into hour, minute, second and microsecond.

Syntax

```
duckdb_time_tz_struct duckdb_from_time_tz(
  duckdb_time_tz micros
);
```

Parameters

- `micros`

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

- `out_micros`

The microsecond component of the time.

- `out_offset`

The timezone offset component of the time.

`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_is_finite_timestamp

Test a duckdb_timestamp to see if it is a finite value.

Syntax

```
bool duckdb_is_finite_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- `ts`

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

- `returns`

True if the timestamp is finite, false if it is ±infinity.

`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_uhugeint_to_double`

Converts a duckdb_uhugeint object (as obtained from a `DUCKDB_TYPE_UHUGEINT` column) into a double.

Syntax

```
double duckdb_uhugeint_to_double(
  duckdb_uhugeint val
);
```

Parameters

- `val`

The uhugeint value.

- `returns`

The converted `double` element.

`duckdb_double_to_uhugeint`

Converts a double value to a duckdb_uhugeint object.

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

Syntax

```
duckdb_uhugeint duckdb_double_to_uhugeint(
  double val
);
```

Parameters

- `val`

The double value.

- `returns`

The converted `duckdb_uhugeint` 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_prepared_statement_type

Returns the statement type of the statement to be executed

Syntax

```
duckdb_statement_type duckdb_prepared_statement_type(
  duckdb_prepared_statement statement
);
```

Parameters

- `statement`

The prepared statement.

- `returns`

duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

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_uhugeint

Binds an duckdb_uhugeint value to the prepared statement at the specified index.

Syntax

```
duckdb_state duckdb_bind_uhugeint(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_uhugeint 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.

Note that the result must be freed with `duckdb_destroy_result`.

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_streaming

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Executes the prepared statement with the given bound parameters, and returns an optionally-streaming query result. To determine if the resulting query was in fact streamed, use `duckdb_result_is_streaming`

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

Note that the result must be freed with `duckdb_destroy_result`.

Syntax

```
duckdb_state duckdb_execute_prepared_streaming(
  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_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

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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_pending_execute_check_state`

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_check_state 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_check_state(
  duckdb_pending_result pending_result
);
```

Parameters

- `pending_result`

The pending result.

- `returns`

The state of the pending result.

`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.

Note that the result must be freed with `duckdb_destroy_result`.

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_create_struct_value`

Creates a struct value from a type and an array of values

Syntax

```
duckdb_value duckdb_create_struct_value(
  duckdb_logical_type type,
  duckdb_value *values
);
```

Parameters

- `type`

The type of the struct

- `values`

The values for the struct fields

- `returns`

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

`duckdb_create_list_value`

Creates a list value from a type and an array of values of length `value_count`

Syntax

```
duckdb_value duckdb_create_list_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`

The type of the list

- `values`

The values for the list

- `value_count`

The number of values in the list

- `returns`

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

`duckdb_create_array_value`

Creates a array value from a type and an array of values of length `value_count`

Syntax

```
duckdb_value duckdb_create_array_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`

The type of the array

- `values`

The values for the array

- `value_count`

The number of values in the array

- `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_logical_type_get_alias

Returns the alias of a duckdb_logical_type, if one is set, else NULL. The result must be destroyed with duckdb_free.

Syntax

```
char *duckdb_logical_type_get_alias(
  duckdb_logical_type type
);
```

Parameters

- type

The logical type to return the alias of

- returns

The alias or NULL

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_array_type`

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

Syntax

```
duckdb_logical_type duckdb_create_array_type(
  duckdb_logical_type type,
  idx_t array_size
);
```

Parameters

- `type`

The child type of array type to create.

- `array_size`

The number of elements in the array.

- `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_enum_type

Creates an ENUM type from the passed member name array. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

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

Parameters

- `enum_name`

The name of the enum.

- `member_names`

The array of names that the enum should consist of.

- `member_count`

The number of elements that were specified in the array.

- `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 enum 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_array_type_child_type`

Retrieves the child type of the given array type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
duckdb_logical_type duckdb_array_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

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

`duckdb_array_type_array_size`

Retrieves the array size of the given array type.

Syntax

```
idx_t duckdb_array_type_array_size(
  duckdb_logical_type type
);
```

Parameters

- `type`

The logical type object

- `returns`

The fixed number of elements the values of this array type can store.

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.

Note that the result must be destroyed with `duckdb_destroy_data_chunk`.

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. You may also use this function to assign BLOBs.

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_array_vector_get_child`

Retrieves the child vector of a array vector.

The resulting vector is valid as long as the parent vector is valid. The resulting vector has the size of the parent vector multiplied by the array size.

Syntax

```
duckdb_vector duckdb_array_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`

The vector

- `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. 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.

Note that the object must be destroyed with `duckdb_appender_destroy`.

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_column_count`

Returns the number of columns in the table that belongs to the appender.

- appender The appender to get the column count from.

Syntax

```
idx_t duckdb_appender_column_count(
  duckdb_appender appender
);
```

Parameters

- `returns`

The number of columns in the table.

`duckdb_appender_column_type`

Returns the type of the column at the specified index.

Note: The resulting type should be destroyed with `duckdb_destroy_logical_type`.

- appender The appender to get the column type from.
- col_idx The index of the column to get the type of.

Syntax

```
duckdb_logical_type duckdb_appender_column_type(
  duckdb_appender appender,
  idx_t col_idx
);
```

Parameters

- `returns`

The duckdb_logical_type of the column.

`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. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. It is not possible to append more values. Call duckdb_appender_error to obtain the error message followed by duckdb_appender_destroy to destroy the invalidated 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`

Closes the appender by flushing all intermediate states and closing it for further appends. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. Call duckdb_appender_error to obtain the error message followed by duckdb_appender_destroy to destroy the invalidated appender.

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`

Closes the appender by flushing all intermediate states to the table and destroying it. By destroying it, this function de-allocates all memory associated with the appender. If flushing the data triggers a constraint violation, then all data is invalidated, and this function returns DuckDBError. Due to the destruction of the appender, it is no longer possible to obtain the specific error message with duckdb_appender_error. Therefore, call duckdb_appender_close before destroying the appender, if you need insights into the specific error.

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_uhugeint

Append a duckdb_uhugeint value to the appender.

Syntax

```
duckdb_state duckdb_append_uhugeint(
  duckdb_appender appender,
  duckdb_uhugeint 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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Fetch the internal arrow schema from the arrow result. Remember to call release on the respective ArrowSchema object.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Fetch the internal arrow schema from the prepared statement. Remember to call release on the respective ArrowSchema object.

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_result_arrow_array`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Convert a data chunk into an arrow struct array. Remember to call release on the respective ArrowArray object.

Syntax

```
void duckdb_result_arrow_array(
  duckdb_result result,
  duckdb_data_chunk chunk,
  duckdb_arrow_array *out_array
);
```

Parameters

- `result`

The result object the data chunk have been fetched from.

- `chunk`

The data chunk to convert.

- `out_array`

The output array.

`duckdb_query_arrow_array`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Fetch an internal arrow struct array from the arrow result. Remember to call release on the respective ArrowArray object.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Returns the number of columns present in 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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Returns the number of rows present in 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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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`

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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 error from.

- `returns`

The error of the result.

duckdb_destroy_arrow

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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_destroy_arrow_stream

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Releases the arrow array stream and de-allocates its memory.

Syntax

```
void duckdb_destroy_arrow_stream(
  duckdb_arrow_stream *stream_p
);
```

Parameters

- `stream`

The arrow array stream to destroy.

duckdb_execute_prepared_arrow

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Executes the prepared statement with the given bound parameters, and returns an arrow query result. Note that after running `duckdb_execute_prepared_arrow`, `duckdb_destroy_arrow` must be called on the result object.

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

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

Scans the Arrow array and creates a view with the given name. Note that after running `duckdb_arrow_array_scan`, `duckdb_destroy_arrow_stream` must be called on the out stream.

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_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` must be called on the result.

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 the 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

DEPRECATION NOTICE: This method is scheduled for removal in a future release.

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.

`duckdb_fetch_chunk`

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

The result must be destroyed with `duckdb_destroy_data_chunk`.

It is not known beforehand how many chunks will be returned by this result.

Syntax

```
duckdb_data_chunk duckdb_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`.

> 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)");

auto result = con.Query("SELECT * FROM integers");
if (result->HasError()) {
    cerr << result->GetError() << endl;
} else {
    cout << result->ToString() << endl;
}
```

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);
```

> **Warning.** 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

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).

> DuckDB has a tldr page that summarizes the most common uses of the CLI client. If you have tldr installed, you can display it by running `tldr duckdb`.

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.

Usage

The typical usage of the `duckdb` command is the following:

```
duckdb [OPTIONS] [FILENAME]
```

Options

The `[OPTIONS]` part encodes arguments for the CLI client. Common options include:

- `-csv`: sets the output mode to CSV
- `-json`: sets the output mode to JSON
- `-readonly`: open the database in read-only mode (see concurrency in DuckDB)

For a full list of options, see the command line arguments page.

In-Memory vs. Persistent Database

When no `[FILENAME]` argument is provided, the DuckDB CLI will open a temporary in-memory database. You will see DuckDB's version number, the information on the connection and a prompt starting with a D.

```
duckdb
```

```
v0.10.2 1601d94f94
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
D
```

To open or create a persistent database, simply include a path as a command line argument like `duckdb path/to/my_database.duckdb` or `duckdb my_database.db`.

Running SQL Statements in the CLI

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

The CLI supports all of DuckDB's rich SQL syntax including `SELECT`, `CREATE`, and `ALTER` statements.

Editor Features

The CLI supports autocompletion, and has sophisticated editor features and syntax highlighting on certain platforms.

Exiting the CLI

To exit the CLI, press `Ctrl+D` if your platform supports it. Otherwise, press `Ctrl+C` or use the `.exit` command. If used a persistent database, DuckDB 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.

Dot Commands

In addition to SQL syntax, special dot commands may be entered into 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.

Frequently-used configurations can be stored in the file `~/.duckdbrc`, which will be loaded when starting the CLI client. See the Configuring the CLI section below for further information on these options.

Below, we summarize a few important dot commands. To see all available commands, see the dot commands page or use the `.help` command.

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
```

> **Warning.** `.open` closes the current database. To keep the current database, while adding a new database, use the ATTACH statement.

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
```

Output Formats

The `.mode` dot command may be used to change the appearance of the tables returned in the terminal output. These include the default `duckbox` mode, `csv` and `json` mode for ingestion by other tools, `markdown` and `latex` for documents, and `insert` mode for generating SQL statements.

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. For details, see the documentation for the output dot command.

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 can 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

Several dot commands can be used to configure the CLI. On startup, the CLI reads and executes all commands in the file `~/.duckdbrc`, including dot commands and SQL statements. This allows you to store the configuration state of the CLI. You may also point to a different initialization file using the `-init`.

Setting a Custom Prompt

As an example, a file in the same directory as the DuckDB CLI named `prompt.sql` 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 work in all terminal environments (e.g., in Windows, unless running with WSL and using the Windows Terminal).

```
.prompt '⚫◗ '
```

To invoke that file on initialization, use this command:

```
duckdb -init prompt.sql
```

This outputs:

```
-- Loading resources from prompt.sql
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
```

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"
```

Loading Extensions

To load extensions, use DuckDB's SQL `INSTALL` and `LOAD` commands as you would other SQL statements.

```
INSTALL fts;
LOAD fts;
```

For details, see the Extension docs.

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('/dev/stdin')"
```

woot
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('/dev/stdin')) TO '/dev/stdout' WITH (FORMAT 'csv', HEADER)"
```

```
woot
42
43
```

Reading Environment Variables

The `getenv` function can read environment variables.

Examples

To retrieve the home directory's path from the `HOME` environment variable, use:

```
SELECT getenv('HOME') AS home;
```

home
/Users/user_name

The output of the `getenv` function can be used to set configuration options. For example, to set the `NULL` order based on the environment variable `DEFAULT_NULL_ORDER`, use:

```
SET default_null_order = getenv('DEFAULT_NULL_ORDER');
```

Restrictions for Reading Environment Variables

The `getenv` function can only be run when the `enable_external_access` is set to `true` (the default setting). It is only available in the CLI client and is not supported in other DuckDB clients.

Prepared Statements

The DuckDB CLI supports executing prepared statements in addition to regular `SELECT` statements. To create and execute a prepared statement in the CLI client, use the `PREPARE` clause and the `EXECUTE` statement.

Command Line Arguments

The table below summarizes DuckDB's command line options. To list all command line options, use the command `duckdb -help`. Fot a list of dot commands available in the CLI shell, see the Dot Commands page.

Argument	Description
`-append`	Append the database to the end of the file
`-ascii`	Set output mode to `ascii`
`-bail`	Stop after hitting an error
`-batch`	Force batch I/O
`-box`	Set output mode to `box`
`-column`	Set output mode to `column`
`-cmd COMMAND`	Run `COMMAND` before reading `stdin`
`-c COMMAND`	Run `COMMAND` and exit
`-csv`	Set output mode to `csv`
`-echo`	Print commands before execution
`-init FILENAME`	Run the script in `FILENAME` upon startup (instead of `~./duckdbrc`)
`-header`	Turn headers on
`-help`	Show this message
`-html`	Set output mode to HTML
`-interactive`	Force interactive I/O
`-json`	Set output mode to `json`
`-line`	Set output mode to `line`
`-list`	Set output mode to `list`
`-markdown`	Set output mode to `markdown`
`-newline SEP`	Set output row separator. Default: `\n`
`-nofollow`	Refuse to open symbolic links to database files
`-noheader`	Turn headers off
`-no-stdin`	Exit after processing options instead of reading stdin
`-nullvalue TEXT`	Set text string for `NULL` values. Default: empty string
`-quote`	Set output mode to `quote`
`-readonly`	Open the database read-only

Argument	Description
`-s COMMAND`	Run COMMAND and exit
`-separator SEP`	Set output column separator to SEP. Default: |
`-stats`	Print memory stats before each finalize
`-table`	Set output mode to `table`
`-unsigned`	Allow loading of unsigned extensions
`-version`	Show DuckDB version

Dot Commands

Dot commands are available in the DuckDB 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.

Dot Commands

Command	Description
`.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
`.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
`.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

Command	Description
`.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 `NAME`
`.timer on|off`	Turn SQL timer on or off
`.width NUM1 NUM2 ...`	Set minimum column widths for columnar output

Using the `.help` Commmand

The `.help` text may be filtered by passing in a text string as the second argument.

```
.help m
```

```
.maxrows COUNT      Sets the maximum number of rows for display (default: 40). 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
```

`.output`: 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:

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);
```

Configuring the Syntax Highlighter

By default the shell includes support for syntax highlighting. The CLI's syntax highlighter can be configured using the following commands.

To turn off the highlighter:

```
.highlight on
```

To turn on the highlighter:

```
.highlight off
```

To configure the color used to highlight constants:

```
.constant
[red|green|yellow|blue|magenta|cyan|white|brightblack|brightred|brightgreen|brightyellow|brightblue|brightmagenta|br

.constantcode [terminal_code]
```

To configure the color used to highlight keywords:

```
.keyword
[red|green|yellow|blue|magenta|cyan|white|brightblack|brightred|brightgreen|brightyellow|brightblue|brightmagenta|br

.keywordcode [terminal_code]
```

Importing Data from CSV

> **Deprecated.** This feature is only included for compatibility reasons and may be removed in the future. Use the `read_csv` function or the COPY statement to load CSV files.

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` 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 INTEGER, col_2 INTEGER);
.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
```

Output Formats

The `.mode` dot 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. 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 and the `latex` mode is useful for writing academic papers.

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
`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
```

Editing

> The linenoise-based CLI editor is currently only available for macOS and Linux.

DuckDB's CLI uses a line-editing library based on linenoise, which has shortcuts that are based on Emacs mode of readline. Below is a list of available commands.

Moving

Key	Action
`Left`	Move back a character
`Right`	Move forward a character
Up	Move up a line. When on the first line, move to previous history entry
Down	Move down a line. When on last line, move to next history entry
Home	Move to beginning of buffer
End	Move to end of buffer
`Ctrl+Left`	Move back a word
`Ctrl+Right`	Move forward a word
`Ctrl+A`	Move to beginning of buffer
`Ctrl+B`	Move back a character
`Ctrl+E`	Move to end of buffer
`Ctrl+F`	Move forward a character
`Alt+Left`	Move back a word
`Alt+Right`	Move forward a word

History

Key	Action
`Ctrl+P`	Move to previous history entry
`Ctrl+N`	Move to next history entry
`Ctrl+R`	Search the history
`Ctrl+S`	Search the history
`Alt+<`	Move to first history entry
`Alt+>`	Move to last history entry
`Alt+N`	Search the history
`Alt+P`	Search the history

Changing Text

Key	Action
`Backspace`	Delete previous character
`Delete`	Delete next character
`Ctrl+D`	Delete next character. When buffer is empty, end editing
`Ctrl+H`	Delete previous character
`Ctrl+K`	Delete everything after the cursor
`Ctrl+T`	Swap current and next character
`Ctrl+U`	Delete all text
`Ctrl+W`	Delete previous word
`Alt+C`	Convert next word to titlecase
`Alt+D`	Delete next word
`Alt+L`	Convert next word to lowercase
`Alt+R`	Delete all text
`Alt+T`	Swap current and next word
`Alt+U`	Convert next word to uppercase
`Alt+Backspace`	Delete previous word
`Alt+\`	Delete spaces around cursor

Completing

Key	Action
`Tab`	Autocomplete. When autocompleting, cycle to next entry
`Shift+Tab`	When autocompleting, cycle to previous entry
`Esc+Esc`	When autocompleting, revert autocompletion

Miscellaneous

Key	Action
`Enter`	Execute query. If query is not complete, insert a newline at the end of the buffer
`Ctrl+J`	Execute query. If query is not complete, insert a newline at the end of the buffer
`Ctrl+C`	Cancel editing of current query
`Ctrl+G`	Cancel editing of current query
`Ctrl+L`	Clear screen
`Ctrl+O`	Cancel editing of current query
`Ctrl+X`	Insert a newline after the cursor
`Ctrl+Z`	Suspend CLI and return to shell, use `fg` to re-open

Using Read-Line

If you prefer, you can use `rlwrap` to use read-line directly with the shell. Then, use `Shift+Enter` to insert a newline and `Enter` to execute the query:

```
rlwrap --substitute-prompt="D " duckdb -batch
```

Autocomplete

The shell offers context-aware autocomplete of SQL queries through the `autocomplete` extension. autocomplete is triggered by pressing `Tab`.

Multiple autocomplete suggestions can be present. You can cycle forwards through the suggestions by repeatedly pressing `Tab`, or `Shift+Tab` to cycle backwards. autocompletion can be reverted by pressing ESC twice.

The shell autocompletes four different groups:

- Keywords
- Table names and table functions
- Column names and scalar functions
- File names

The shell looks at the position in the SQL statement to determine which of these autocompletions to trigger. For example:

```
SELECT s -> student_id
```

```
SELECT student_id F -> FROM
```

```
SELECT student_id FROM g -> grades
```

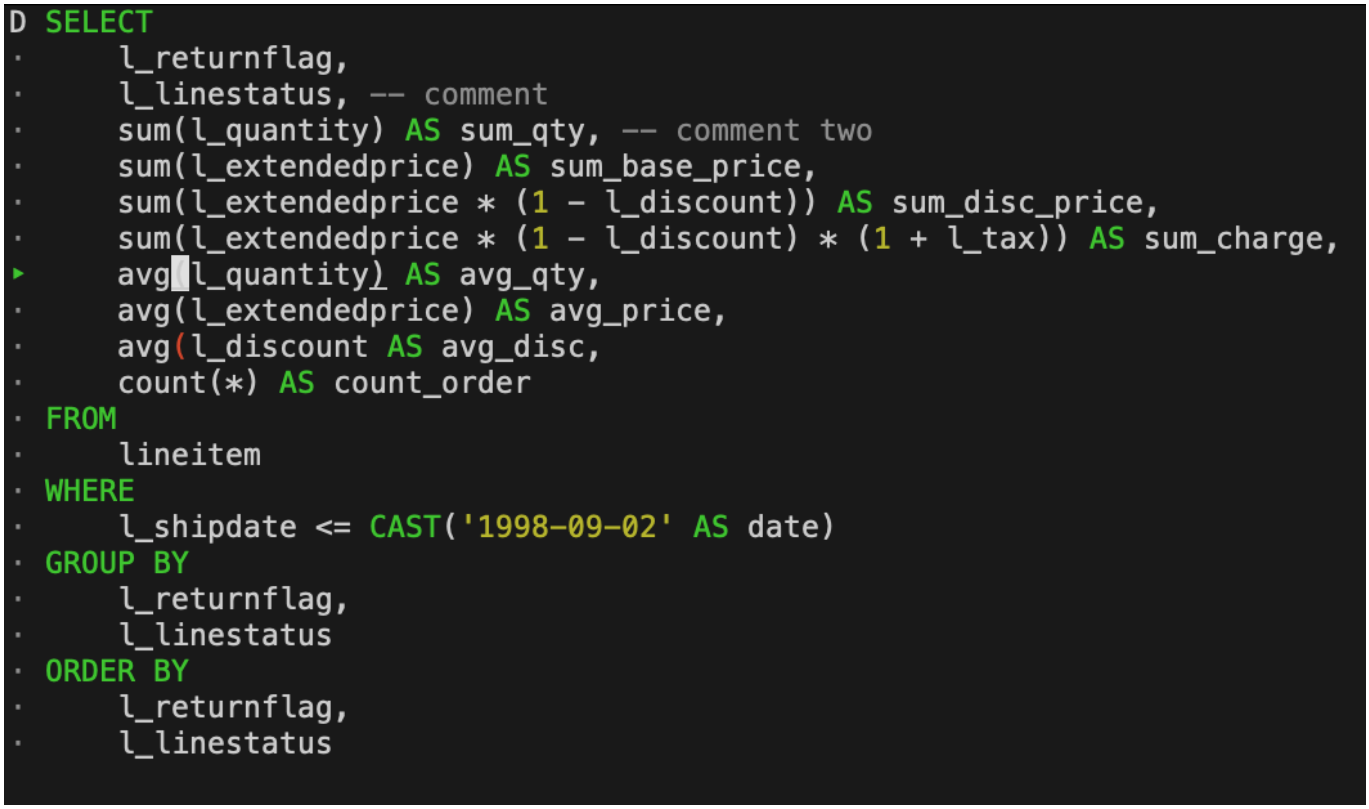
```
SELECT student_id FROM 'd -> data/
```

```
SELECT student_id FROM 'data/ -> data/grades.csv
```

Syntax Highlighting

> Syntax highlighting in the CLI is currently only available for macOS and Linux.

SQL queries that are written in the shell are automatically highlighted using syntax highlighting.



There are several components of a query that are highlighted in different colors. The colors can be configured using dot commands. Syntax highlighting can also be disabled entirely using the `.highlight off` command.

Below is a list of components that can be configured.

Type	Command	Default Color
Keywords	.keyword	green
Constants ad literals	.constant	yellow
Comments	.comment	brightblack
Errors	.error	red
Continuation	.cont	brightblack
Continuation (Selected)	.cont_sel	green

The components can be configured using either a supported color name (e.g., `.keyword red`), or by directly providing a terminal code to use for rendering (e.g., `.keywordcode \033[31m`). Below is a list of supported color names and their corresponding terminal codes.

Color	Terminal Code
red	`\033[31m`
green	`\033[32m`
yellow	`\033[33m`

Color	Terminal Code
blue	`\033[34m`
magenta	`\033[35m`
cyan	`\033[36m`
white	`\033[37m`
brightblack	`\033[90m`
brightred	`\033[91m`
brightgreen	`\033[92m`
brightyellow	`\033[93m`
brightblue	`\033[94m`
brightmagenta	`\033[95m`
brightcyan	`\033[96m`
brightwhite	`\033[97m`

For example, here is an alternative set of syntax highlighting colors:

```
.keyword brightred
.constant brightwhite
.comment cyan
.error yellow
.cont blue
.cont_sel brightblue
```

If you wish to start up the CLI with a different set of colors every time, you can place these commands in the `~/.duckdbrc` file that is loaded on start-up of the CLI.

Error Highlighting

The shell has support for highlighting certain errors. In particular, mismatched brackets and unclosed quotes are highlighted in red (or another color if specified). This highlighting is automatically disabled for large queries. In addition, it can be disabled manually using the `.render_errors off` command.

Go

The DuckDB Go driver, `go-duckdb`, allows using DuckDB via the `database/sql` interface. For examples on how to use this interface, see the official documentation and tutorial.

> The Go client is a third-party library and its repository is hosted https://github.com/marcboeker/go-duckdb.

Installation

To install the `go-duckdb` client, run:

```
go get github.com/marcboeker/go-duckdb
```

Importing

To import the DuckDB Go package, add the following entries to your imports:

```
import (
    "database/sql"
    _ "github.com/marcboeker/go-duckdb"
)
```

Appender

The DuckDB Go client supports the DuckDB Appender API for bulk inserts. You can obtain a new Appender by supplying a DuckDB connection to `NewAppenderFromConn()`. For example:

```
connector, err := duckdb.NewConnector("test.db", nil)
if err != nil {
  ...
}
conn, err := connector.Connect(context.Background())
if err != nil {
  ...
}
defer conn.Close()

// Retrieve appender from connection (note that you have to create the table 'test' beforehand).
appender, err := NewAppenderFromConn(conn, "", "test")
if err != nil {
  ...
}
defer appender.Close()

err = appender.AppendRow(...)
if err != nil {
  ...
}

// Optional, if you want to access the appended rows immediately.
err = appender.Flush()
```

```
if err != nil {
  ...
}
```

Examples

Simple Example

An example for using the Go API is as follows:

```
package main

import (
    "database/sql"
    "errors"
    "fmt"
    "log"

    _ "github.com/marcboeker/go-duckdb"
)

func main() {
    db, err := sql.Open("duckdb", "")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    _, err = db.Exec(` CREATE TABLE people (id INTEGER, name VARCHAR)`)
    if err != nil {
        log.Fatal(err)
    }
    _, err = db.Exec(` INSERT INTO people VALUES (42, 'John')`)
    if err != nil {
        log.Fatal(err)
    }

    var (
        id   int
        name string
    )
    row := db.QueryRow(` SELECT id, name FROM people`)
    err = row.Scan(&id, &name)
    if errors.Is(err, sql.ErrNoRows) {
        log.Println("no rows")
    } else if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("id: %d, name: %s\n", id, name)
}
```

More Examples

For more examples, see the examples in the duckdb-go repository.

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 using the following statement:

```
Class.forName("org.duckdb.DuckDBDriver");
```

To create a DuckDB connection, call `DriverManager` with the `jdbc:duckdb:` JDBC URL prefix, like so:

```
import java.sql.Connection;
import java.sql.DriverManager;

Connection conn = DriverManager.getConnection("jdbc:duckdb:");
```

To use DuckDB-specific features such as the Appender, cast the object to a `DuckDBConnection`:

```
import java.sql.DriverManager;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) 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 readOnlyProperty = new Properties();
readOnlyProperty.setProperty("duckdb.read_only", "true");
Connection conn = DriverManager.getConnection("jdbc:duckdb:/tmp/my_database", readOnlyProperty);
```

Additional connections can be created using the `DriverManager`. A more efficient mechanism is to call the `DuckDBConnection#duplicate()` method:

```
Connection conn2 = ((DuckDBConnection) conn).duplicate();
```

Multiple connections are allowed, but mixing read-write and read-only connections is unsupported.

Configuring Connections

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.

```
Properties connectionProperties = new Properties();
connectionProperties.setProperty("temp_directory", "/path/to/temp/dir/");
Connection conn = DriverManager.getConnection("jdbc:duckdb:/tmp/my_database", connectionProperties);
```

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.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

Connection conn = DriverManager.getConnection("jdbc:duckdb:");

// 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));
    }
}
stmt.close();
```

```
jeans
1
hammer
2
```

DuckDB also supports prepared statements as per the JDBC API:

```
import java.sql.PreparedStatement;

try (PreparedStatement stmt = conn.prepareStatement("INSERT INTO items VALUES (?, ?, ?);")) {
    stmt.setString(1, "chainsaw");
    stmt.setDouble(2, 500.0);
    stmt.setInt(3, 42);
    stmt.execute();
    // more calls to execute() possible
}
```

> **Warning.** 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 stmt = conn.prepareStatement("SELECT * FROM generate_series(2000)");
    var resultset = (DuckDBResultSet) 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"));
        }
    }
    stmt.close();
}
```

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 binding
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 setup
    try (var conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:")) {
        conn.registerArrowStream("asdf", arrow_array_stream);

        // run a query
        try (var stmt = conn.createStatement();
             var rs = (DuckDBResultSet) stmt.executeQuery("SELECT count(*) FROM asdf")) {
            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);
```

Appender

The Appender is available in the DuckDB JDBC driver via the `org.duckdb.DuckDBAppender` class. The constructor of the class requires the schema name and the table name it is applied to. The Appender is flushed when the `close()` method is called.

Example:

```
import java.sql.DriverManager;
import java.sql.Statement;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
Statement stmt = conn.createStatement();
stmt.execute("CREATE TABLE tbl (x BIGINT, y FLOAT, s VARCHAR)");

// using try-with-resources to automatically close the appender at the end of the scope
try (var appender = conn.createAppender(DuckDBConnection.DEFAULT_SCHEMA, "tbl")) {
    appender.beginRow();
    appender.append(10);
    appender.append(3.2);
    appender.append("hello");
    appender.endRow();
    appender.beginRow();
    appender.append(20);
    appender.append(-8.1);
    appender.append("world");
    appender.endRow();
}
stmt.close();
```

Batch Writer

The DuckDB JDBC driver offers batch write functionality. The batch writer supports prepared statements to mitigate the overhead of query parsing.

> The preferred method for bulk inserts is to use the Appender due to its higher performance. However, when using the Appender is not possbile, the batch writer is available as alternative.

Batch Writer with Prepared Statements

```
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
PreparedStatement stmt = conn.prepareStatement("INSERT INTO test (x, y, z) VALUES (?, ?, ?);");

stmt.setObject(1, 1);
stmt.setObject(2, 2);
stmt.setObject(3, 3);
stmt.addBatch();

stmt.setObject(1, 4);
stmt.setObject(2, 5);
stmt.setObject(3, 6);
stmt.addBatch();

stmt.executeBatch();
stmt.close();
```

Batch Writer with Vanilla Statements

The batch writer also supports vanilla SQL statements:

```
import java.sql.DriverManager;
import java.sql.Statement;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
Statement stmt = conn.createStatement();

stmt.execute("CREATE TABLE test (x INTEGER, y INTEGER, z INTEGER)");

stmt.addBatch("INSERT INTO test (x, y, z) VALUES (1, 2, 3);");
stmt.addBatch("INSERT INTO test (x, y, z) VALUES (4, 5, 6);");

stmt.executeBatch();
stmt.close();
```

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, enter the package manager using the] key, and issue the following command:

```
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)
```

Appender API

The DuckDB Julia package also supports the Appender API, which is much faster than using prepared statements or individual `INSERT INTO` statements. Appends are made in row-wise format. For every column, an `append()` call should be made, after which the row should be finished by calling `flush()`. After all rows have been appended, `close()` should be used to finalize the Appender and clean up the resulting memory.

```
using DuckDB, DataFrames, Dates
db = DuckDB.DB()
# create a table
DBInterface.execute(db, "CREATE OR REPLACE
                         TABLE data(id INTEGER PRIMARY KEY, value FLOAT,
                         timestamp TIMESTAMP, date DATE)")
# create data to insert
len = 100
df = DataFrames.DataFrame(
        id = collect(1:len),
        value = rand(len),
        timestamp = Dates.now() + Dates.Second.(1:len),
        date = Dates.today() + Dates.Day.(1:len)
    )
# append data by row
appender = DuckDB.Appender(db, "data")
for i in eachrow(df)
    for j in i
        DuckDB.append(appender, j)
    end
    DuckDB.end_row(appender)
end
# flush the appender after all rows
DuckDB.flush(appender)
DuckDB.close(appender)
```

Concurrency

Within a Julia process, tasks are able to concurrently read and write to the database, as long as each task maintains its own connection to the database. In the example below, a single task is spawned to periodically read the database and many tasks are spawned to write to the database using both `INSERT` statements as well as the Appender API.

```
using Dates, DataFrames, DuckDB
db = DuckDB.DB()
DBInterface.connect(db)
DBInterface.execute(db, "CREATE OR REPLACE TABLE data (date TIMESTAMP, id INTEGER)")

function run_reader(db)
    # create a DuckDB connection specifically for this task
    conn = DBInterface.connect(db)
    while true
        println(DBInterface.execute(conn,
                "SELECT id, count(date) AS count, max(date) AS max_date
```

```
                FROM data GROUP BY id ORDER BY id") |> DataFrames.DataFrame)
        Threads.sleep(1)
    end
    DBInterface.close(conn)
end
# spawn one reader task
Threads.@spawn run_reader(db)

function run_inserter(db, id)
    # create a DuckDB connection specifically for this task
    conn = DBInterface.connect(db)
    for i in 1:1000
        Threads.sleep(0.01)
        DuckDB.execute(conn, "INSERT INTO data VALUES (current_timestamp, ?)"; id);
    end
    DBInterface.close(conn)
end
# spawn many insert tasks
for i in 1:100
    Threads.@spawn run_inserter(db, 1)
end

function run_appender(db, id)
    # create a DuckDB connection specifically for this task
    appender = DuckDB.Appender(db, "data")
    for i in 1:1000
        Threads.sleep(0.01)
        row = (Dates.now(Dates.UTC), id)
        for j in row
            DuckDB.append(appender, j);
        end
        DuckDB.end_row(appender);
        DuckDB.flush(appender);
    end
    DuckDB.close(appender);
end
# spawn many appender tasks
for i in 1:100
    Threads.@spawn run_appender(db, 2)
end
```

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.

For TypeScript wrappers, see the duckdb-async project.

Initializing

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);
  }
});
```

Running a Query

The following code snippet runs a simple query using the `Database.all()` method.

```
db.all('SELECT 42 AS fortytwo', function(err, res) {
  if (err) {
    console.warn(err);
    return;
  }
  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) {
    console.warn(err);
    return;
  }
  console.log(res[0].fortytwo)
  console.log(res[0].hello)
});
```

Connections

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) {
    console.warn(err);
    return;
  }
  console.log(res[0].fortytwo)
});
```

Prepared Statements

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) {
    console.warn(err);
  } else {
    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) {
    console.warn(err);
  } else {
    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) {
      console.warn(err);
    } else {
      console.log(res[0].fortytwo)
    }
  });
});
```

Inserting Data via Arrow

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) {
        console.warn(err);
        return;
    }

    const arrowTable = arrow.tableFromJSON(jsonData);
    db.register_buffer("jsonDataTable", [arrow.tableToIPC(arrowTable)], true, (err, res) => {
        if (err) {
            console.warn(err);
            return;
        }

        // `SELECT * FROM jsonDataTable` would return the entries in `jsonData`
    });
});
```

Loading Unsigned Extensions

To load unsigned extensions, instantiate the database as follows:

```
db = new duckdb.Database(':memory:', {"allow_unsigned_extensions": "true"});
```

Node.js 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
    - .stream(sql, ...params)
    - .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
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	*
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
- .stream(sql, ...params)
- .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	
...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.stream(sql, ...params)

Kind: instance method of Database

Param	Type
sql	
...params	*

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	*

Param	Type
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

Param
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
[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`.

Python version: DuckDB requires 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 directly query Pandas DataFrames, Polars DataFrames and Arrow tables. Note that these are read-only, i.e., editing these tables via `INSERT` or `UPDATE` statements is not possible.

```python
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.

```python
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.

```python
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
```

Connection Options

Applications can open a new DuckDB connection via the `duckdb.connect()` method.

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. Invoking the `duckdb.connect()` method without arguments returns a connection, which also uses an in-memory database:

```python
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 reconnecting 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
```

Configuration

The `duckdb.connect()` accepts a `config` dictionary, where configuration options can be specified. For example:

```
import duckdb

con = duckdb.connect(config = {'threads': 1})
```

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.

> 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.

Using Connections in Parallel Python Programs

The `DuckDBPyConnection` object is not thread-safe. If you would like to write to the same database from multiple threads, create a cursor for each thread with the `DuckDBPyConnection.cursor()` method.

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:

```python
import duckdb

con = duckdb.connect()
con.install_extension("spatial")
con.load_extension("spatial")
```

To load unsigned extensions, use the `config = {"allow_unsigned_extensions": "true"}` argument to the `duckdb.connect()` method.

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('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")
# read a Parquet over https
duckdb.read_parquet("https://some.url/some_file.parquet")
# read a list of Parquet files
duckdb.read_parquet(["file1.parquet", "file2.parquet", "file3.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')")
```

Directly Accessing DataFrames and Arrow Objects

DuckDB is automatically able to query certain Python variables by referring to their variable name (as if it was a table). These types include the following: Pandas DataFrame, Polars DataFrame, Polars LazyFrame, NumPy arrays, relations, and Arrow objects. Accessing these is made possible by replacement scans.

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 100k
duckdb.execute("SET GLOBAL pandas_analyze_sample = 100_000")
```

Registering Objects

You can register Python objects as DuckDB tables using the `DuckDBPyConnection.register()` function.

The precedence of objects with the same name is as follows:

- Objects explicitly registered via `DuckDBPyConnection.register()`
- Native DuckDB tables and views
- Replacement scans

Conversion between DuckDB and Python

This page documents the rules for converting Python objects to DuckDB and DuckDB results to Python.

Object Conversion: Python Object to DuckDB

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}
```

> The names of the fields matter 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}
```

> 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 Results to Python

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.

File-Based Connection

If the database file does not exist, it will be created (the file extension may be `.db`, `.duckdb`, or anything else).

In-Memory Connection

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.

Default Connection

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` or by using `duckdb.default_connection`.

```
import duckdb

duckdb.execute("CREATE TABLE tbl AS SELECT 42 a")
con = duckdb.connect(":default:")
con.sql("SELECT * FROM tbl")
# or
duckdb.default_connection.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)")
```

```python
# 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.

Prepared Statements

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. Values are converted according to the conversion rules.

Here are some examples:

```python
# 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')]
```

> **Warning.** Do *not* use `executemany` to insert large amounts of data into DuckDB. See the data ingestion page for better options.

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 is the following:

```python
import duckdb

res = duckdb.execute("""
    SELECT
        $my_param,
        $other_param,
        $also_param
    """,
```

```
    {
        "my_param": 5,
        "other_param": "DuckDB",
        "also_param": [42]
    }
).fetchall()
print(res)
# [(5, 'DuckDB', [42])]
```

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(10_000_000_000) 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 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(1_000_000) 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(1_000_000) 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(1_000_000) 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 int64	id 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(1_000_000) 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(1_000_000) 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(1_000_000) 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) from 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 generate_random_name():
    fake = Faker()
    return fake.name()
```

```
duckdb.create_function("random_name", generate_random_name, [], VARCHAR)
res = duckdb.sql("SELECT random_name()").fetchall()
print(res)
# [('Gerald Ashley',)]
```

Creating Functions

To register a Python UDF, use the `create_function` method from a DuckDB connection. Here is the syntax:

```
import duckdb
con = duckdb.connect()
con.create_function(name, function, parameters, return_type)
```

The `create_function` method takes 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. **parameters**: Scalar functions can operate on one or more columns. This parameter takes a list of column types used as input.
4. **return_type**: Scalar functions return one element per row. This parameter specifies the return type of the function.
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 setting 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 `parameters`.

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()
print(res)
# [(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.remove_function("my_counter")
count.counter = 0
con.create_function("my_counter", count, side_effects = True)
res = con.sql("SELECT my_counter() FROM range(10)").fetchall()
print(res)
# [(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 Built-ins

The table below shows the mapping of Python Built-in types to DuckDB type.

Built-in types	DuckDB type
bool	BOOLEAN
bytearray	BLOB
bytes	BLOB
float	DOUBLE
int	BIGINT
str	VARCHAR

Numpy DTypes

The table below shows the mapping of Numpy DType to DuckDB type.

Type	DuckDB type
bool	BOOLEAN
float32	FLOAT
float64	DOUBLE
int16	SMALLINT
int32	INTEGER
int64	BIGINT
int8	TINYINT
uint16	USMALLINT
uint32	UINTEGER
uint64	UBIGINT
uint8	UTINYINT

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.

```python
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.

```python
import duckdb

duckdb.typing.DuckDBPyType({'a': str, 'b': int})
# STRUCT(a VARCHAR, b BIGINT)
```

Union[<type_1>, ... <type_n>]

`typing.Union` objects map to a UNION type of the provided types.

```python
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 built-in types, you can use the constants defined in `duckdb.typing`:

DuckDB type
BIGINT
BIT
BLOB
BOOLEAN
DATE
DOUBLE
FLOAT
HUGEINT
INTEGER
INTERVAL
SMALLINT
SQLNULL
TIME_TZ
TIME
TIMESTAMP_MS
TIMESTAMP_NS
TIMESTAMP_S
TIMESTAMP_TZ

DuckDB type
TIMESTAMP
TINYINT
UBIGINT
UHUGEINT
UINTEGER
USMALLINT
UTINYINT
UUID
VARCHAR

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 Expression 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],
    'b': [True, None, False, True],
    'c': [42, 21, 13, 14]
})

# selecting a single column
col = duckdb.ColumnExpression('a')
res = duckdb.df(df).select(col).fetchall()
print(res)
# [(1,), (2,), (3,), (4,)]

# selecting multiple columns
col_list = [
        duckdb.ColumnExpression('a') * 10,
        duckdb.ColumnExpression('b').isnull(),
        duckdb.ColumnExpression('c') + 5
    ]
res = duckdb.df(df).select(*col_list).fetchall()
print(res)
# [(10, False, 47), (20, True, 26), (30, False, 18), (40, False, 19)]
```

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.

```python
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.

```python
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` and it can be set using `.else(value = ...)`. Additional `WHEN (...) THEN (...)` blocks can be added with `.when(condition = ..., value = ...)`.

```python
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.

Operation	Description
`.alias(name: str)`	Applies an alias to the expression.
`.cast(type: DuckDBPyType)`	Applies a cast to the provided type on the expression.
`.isin(*exprs: Expression)`	Creates an `IN` expression against the provided expressions as the list.
`.isnotin(*exprs: Expression)`	Creates a `NOT IN` expression against the provided expressions as the list.
`.isnotnull()`	Checks whether the expression is not `NULL`.
`.isnull()`	Checks whether the expression is `NULL`.

Order Operations

When expressions are provided to `DuckDBPyRelation.order()`, the following order operations can be applied.

Operation	Description
`.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.

> **Warning.** 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')
]
```

Contribution Guidelines

Contributions to the experimental Spark API are welcome. When making a contribution, please follow these guidelines:

- Instead of using temporary files, use our `pytest` testing framework.
- When adding new functions, ensure that method signatures comply with those in the PySpark API.

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]:
┌───────────────┬──────────────────────────────────────────────
│  explain_key  │                  explain_value
│    varchar    │                     varchar
├───────────────┼──────────────────────────────────────────────
│ physical_plan │ ┌───────────────────────────┐\n│       PROJECTION        │\n│   ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─
│\n│             x   …   │
└───────────────┴──────────────────────────────────────────────
```

To work around this, `print` the output of the `explain()` function:

```
In [2]: print(duckdb.sql("SELECT 42 AS x").explain())
```

```
Out[2]:
┌───────────────────────────┐
│         PROJECTION        │
│   ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─   │
│             x             │
└─────────────┬─────────────┘
┌─────────────┴─────────────┐
│         DUMMY_SCAN        │
└───────────────────────────┘
```

Please also check out the Jupyter guide for tips on using Jupyter with JupySQL.

Error When Importing the DuckDB Python Package on Windows

When importing DuckDB on Windows, the Python runtime may return the following error:

```
import duckdb
```

```
ImportError: DLL load failed while importing duckdb: The specified module could not be found.
```

The solution is to install the Microsoft Visual C++ Redistributable package.

R API

Installation

duckdb: R API

The DuckDB R API can be installed using `install.packages("duckdb")`. Please see the installation page for details.

duckplyr: dplyr API

DuckDB offers a dplyr-compatible API via the `duckplyr` package. It can be installed using `install.packages("duckplyr")`. For details, see the `duckplyr` documentation.

Reference Manual

The reference manual for the DuckDB R API is available at R.duckdb.org.

Basic API Usage

The standard DuckDB R API implements the DBI interface for R. If you are not familiar with DBI yet, see here for an introduction.

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
```

> **Warning.** 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
```

> 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 = true)") |>
  filter(month == "3") |>
  summarise(delay = mean(dep_time, na.rm = TRUE)) |>
  collect()
```

Memory Limit

You can use the `memory_limit` configuration option to limit the memory use of DuckDB, e.g.:

```
SET memory_limit = '2GB';
```

Note that this limit is only applied to the memory DuckDB uses and it does not affect the memory use of other R libraries. Therefore, the total memory used by the R process may be higher than the configured `memory_limit`.

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.

```
#[derive(Debug)]
struct Person {
    id: i32,
    name: String,
    data: Option<Vec<u8>>,
}

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());
}
```

Appender

The Rust client supports the DuckDB Appender API for bulk inserts. For example:

```rust
fn insert_rows(conn: &Connection) -> Result<()> {
    let mut app = conn.appender("foo")?;
    app.append_rows([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])?;
    Ok(())
}
```

Swift API

DuckDB offers a Swift API. See the announcement post for details.

Instantiating DuckDB

DuckDB supports both in-memory and persistent databases. To work with an in-memory datatabase, run:

```
let database = try Database(store: .inMemory)
```

To work with a persistent database, run:

```
let database = try Database(store: .file(at: "test.db"))
```

Queries can be issued through a database connection.

```
let connection = try database.connect()
```

DuckDB supports multiple connections per database.

Application Example

The rest of the page is based on the example of our announcement post, which uses raw data from NASA's Exoplanet Archive loaded directly into DuckDB.

Creating an Application-Specific Type

We first create an application-specific type that we'll use to house our database and connection and through which we'll eventually define our app-specific queries.

```
import DuckDB

final class ExoplanetStore {

  let database: Database
  let connection: Connection

  init(database: Database, connection: Connection) {
    self.database = database
    self.connection = connection
  }
}
```

Loading a CSV File

We load the data from NASA's Exoplanet Archive:

```
wget https://exoplanetarchive.ipac.caltech.edu/TAP/sync?query=select+pl_name+,+disc_
year+from+pscomppars&format=csv -O downloaded_exoplanets.csv
```

Once we have our CSV downloaded locally, we can use the following SQL command to load it as a new table to DuckDB:

```
CREATE TABLE exoplanets AS
    SELECT * FROM read_csv('downloaded_exoplanets.csv');
```

Let's package this up as a new asynchronous factory method on our `ExoplanetStore` type:

```
import DuckDB
import Foundation

final class ExoplanetStore {

  // Factory method to create and prepare a new ExoplanetStore
  static func create() async throws -> ExoplanetStore {

  // Create our database and connection as described above
    let database = try Database(store: .inMemory)
    let connection = try database.connect()

  // Download the CSV from the exoplanet archive
  let (csvFileURL, _) = try await URLSession.shared.download(
    from: URL(string: "https://exoplanetarchive.ipac.caltech.edu/TAP/sync?query=select+pl_name+,+disc_
year+from+pscomppars&format=csv")!)

  // Issue our first query to DuckDB
  try connection.execute("""
      CREATE TABLE exoplanets AS
          SELECT * FROM read_csv('\(csvFileURL.path)');
  """)

  // Create our pre-populated ExoplanetStore instance
    return ExoplanetStore(
    database: database,
      connection: connection
  )
  }

  // Let's make the initializer we defined previously
  // private. This prevents anyone accidentally instantiating
  // the store without having pre-loaded our Exoplanet CSV
  // into the database
  private init(database: Database, connection: Connection) {
  ...
  }
}
```

Querying the Database

The following example queires DuckDB from within Swift via an async function. This means the callee won't be blocked while the query is executing. We'll then cast the result columns to Swift native types using DuckDB's `ResultSet` `cast(to:)` family of methods, before finally wrapping them up in a `DataFrame` from the TabularData framework.

```
...

import TabularData

extension ExoplanetStore {

  // Retrieves the number of exoplanets discovered by year
  func groupedByDiscoveryYear() async throws -> DataFrame {

  // Issue the query we described above
    let result = try connection.query("""
      SELECT disc_year, count(disc_year) AS Count
        FROM exoplanets
```

```
        GROUP BY disc_year
        ORDER BY disc_year
      """)

    // Cast our DuckDB columns to their native Swift
    // equivalent types
    let discoveryYearColumn = result[0].cast(to: Int.self)
    let countColumn = result[1].cast(to: Int.self)

    // Use our DuckDB columns to instantiate TabularData
    // columns and populate a TabularData DataFrame
    return DataFrame(columns: [
      TabularData.Column(discoveryYearColumn).eraseToAnyColumn(),
      TabularData.Column(countColumn).eraseToAnyColumn(),
    ])
  }
}
```

Complete Project

For the complete example project, clone the DuckDB Swift repo and open up the runnable app project located in `Examples/SwiftUI/ExoplanetE`

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

Instantiation

DuckDB-Wasm has multiple ways to be instantiated depending on the use case.

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);
```

Statically Served

It is possible to 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-flavored 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';

// EOS signal according to Arrorw IPC streaming format
// See https://arrow.apache.org/docs/format/Columnar.html#ipc-streaming-format
const EOS = new Uint8Array([255, 255, 255, 255, 0, 0, 0, 0]);

const arrowTable = tableFromArrays({
  id: [1, 2, 3],
  name: ['John', 'Jane', 'Jack'],
  age: [20, 21, 22],
});

await c.insertArrowTable(arrowTable, { name: 'arrow_table' });
// Write EOS
await c.insertArrowTable(EOS, { 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' }));
}

// Write EOS
streamInserts.push(c.insertArrowFromIPCStream(EOS, { 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 c.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-flavored)

```
// ..., by specifying URLs in the SQL text
await c.query(`
    CREATE TABLE direct AS
        SELECT * FROM 'https://origin/remote.parquet'
`);
```

> **Tip.** If you encounter a Network Error (`"Failed to execute 'send' on 'XMLHttpRequest')"` when you try to query files from S3, configure the S3 permission CORS header. For example:

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "GET",
            "HEAD"
        ],
        "AllowedOrigins": [
            "*"
        ],
        "ExposeHeaders": [],
        "MaxAgeSeconds": 3000
    }
]
```

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.

> 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	

Extension name	Description	Aliases
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 GitHub	Adds support for reading SQLite database files	sqlite, sqlite3
sqlsmith		
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 0.7 of ADBC.

Please refer to the ADBC documentation page for a more extensive discussion on ADBC and a detailed API explanation.

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)`
`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)`

Function name	Description	Arguments	Example
`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)`
`StatementSetSubstraitPlan`	Set a substrait plan to execute. The query can then be executed with StatementExecuteQuery.	`(AdbcStatement*, const uint8_t*, size_t, AdbcError*)`	`AdbcStatementSetSubstraitPlan(&adbc_statement, substrait_plan, length, &adbc_error)`
`StatementExecuteQuery`	Execute a statement and get the results.	`(AdbcStatement*, ArrowArrayStream*, int64_t*, AdbcError*)`	`AdbcStatementExecuteQuery(&adbc_statement, &arrow_stream, &rows_affected, &adbc_error)`

Function name	Description	Arguments	Example
`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
```

> For details on the `adbc_driver_manager` package, see the `adbc_driver_manager` package documentation.

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 Driver Manager 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.

The ODBC driver is available for all operating systems. Visit the installation page for direct links.

ODBC API on Linux

Driver Manager

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:

On Debian-based distributions (Ubuntu, Mint, etc.), run:

```
sudo apt-get install unixodbc odbcinst
```

On Fedora-based distributions (Amazon Linux, RHEL, CentOS, etc.), run:

```
sudo yum install unixODBC
```

Setting Up the Driver

1. Download the ODBC Linux Asset corresponding to your architecture:
   - x86_64 (AMD64)
   - arm64
2. The package contains the following files:
   - `libduckdb_odbc.so`: the DuckDB driver.
   - `unixodbc_setup.sh`: a setup script to aid the configuration on Linux.

   To extract them, run:

```
mkdir duckdb_odbc && unzip duckdb_odbc-linux-amd64.zip -d duckdb_odbc
```

3. The `unixodbc_setup.sh` script performs 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`.

   Run the following commands with either option `-u` or `-s` to configure DuckDB ODBC.

   The `-u` option based on the user home directory to setup the ODBC init files.

   ```
   ./unixodbc_setup.sh -u
   ```

   The `-s` option changes the system level files that will be visible for all users, because of that it requires root privileges.

   ```
   sudo ./unixodbc_setup.sh -s
   ```

   The option `--help` shows the usage of `unixodbc_setup.sh` prints the help.

   ```
   ./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
   ```

4. The ODBC setup on Linux is based on the `.odbc.ini` and `.odbcinst.ini` files.

   These files can be placed to the user home directory `/home/<username>` or in the system `/etc` directory. The Driver Manager prioritizes the user configuration files over the system files.

   For the details of the configuration parameters, see the ODBC configuration page.

ODBC API on Windows

Using the DuckDB ODBC API on Windows requires the following steps:

1. The Microsoft Windows requires an ODBC Driver Manager to manage communication between applications and the ODBC drivers. The Driver Manager on Windows is provided in a DLL file `odbccp32.dll`, and other files and tools. For detailed information check out the Common ODBC Component Files.
2. DuckDB releases the ODBC driver as an asset. For Windows, download it from the Windows ODBC asset (x86_64/AMD64).
3. The archive 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.

   Decompress the archive to a directory (e.g., `duckdb_odbc`). For example, run:

   ```
   mkdir duckdb_odbc && unzip duckdb_odbc-windows-amd64.zip -d duckdb_odbc
   ```

4. The `odbc_install.exe` binary performs 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 are required. In case of a non-administrator, a User Account Control prompt will occur.

5. `odbc_install.exe` adds a default DSN configuration into the ODBC registries with a default database `:memory:`.

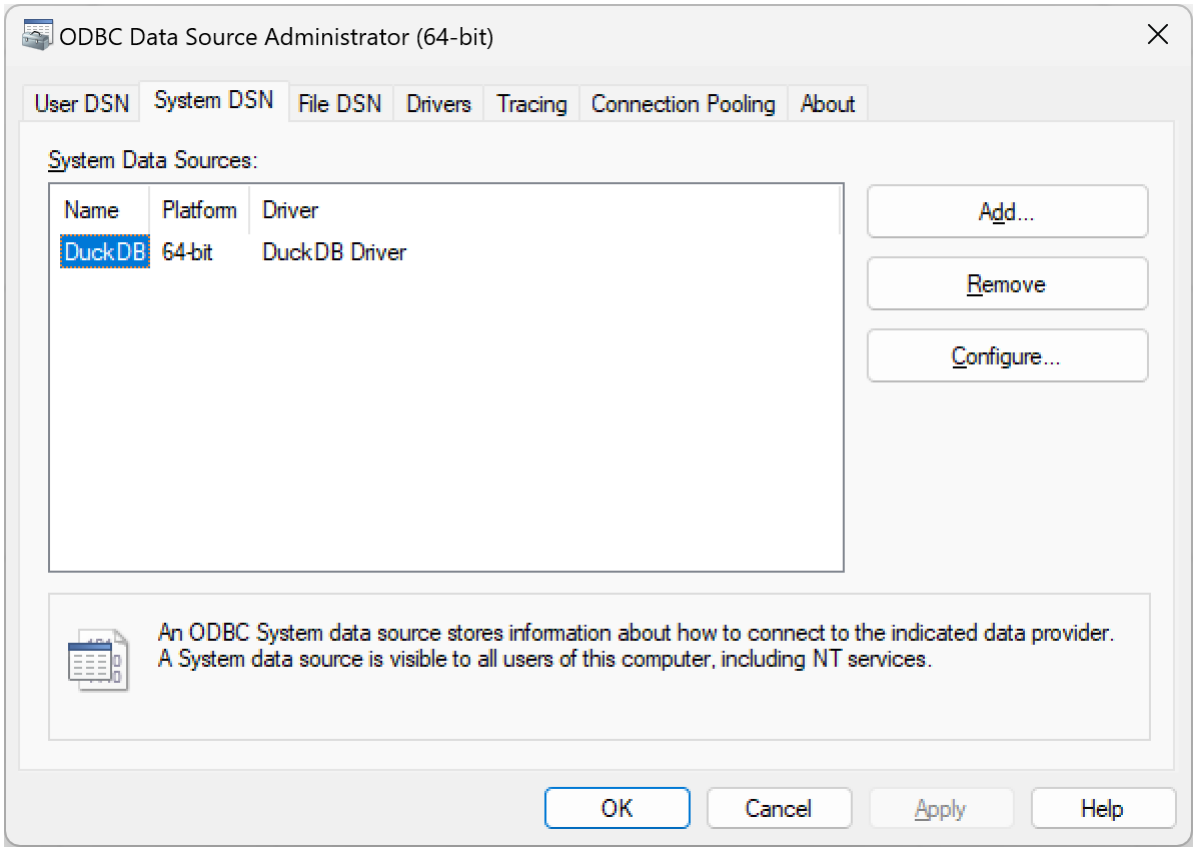
DSN Windows Setup

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:

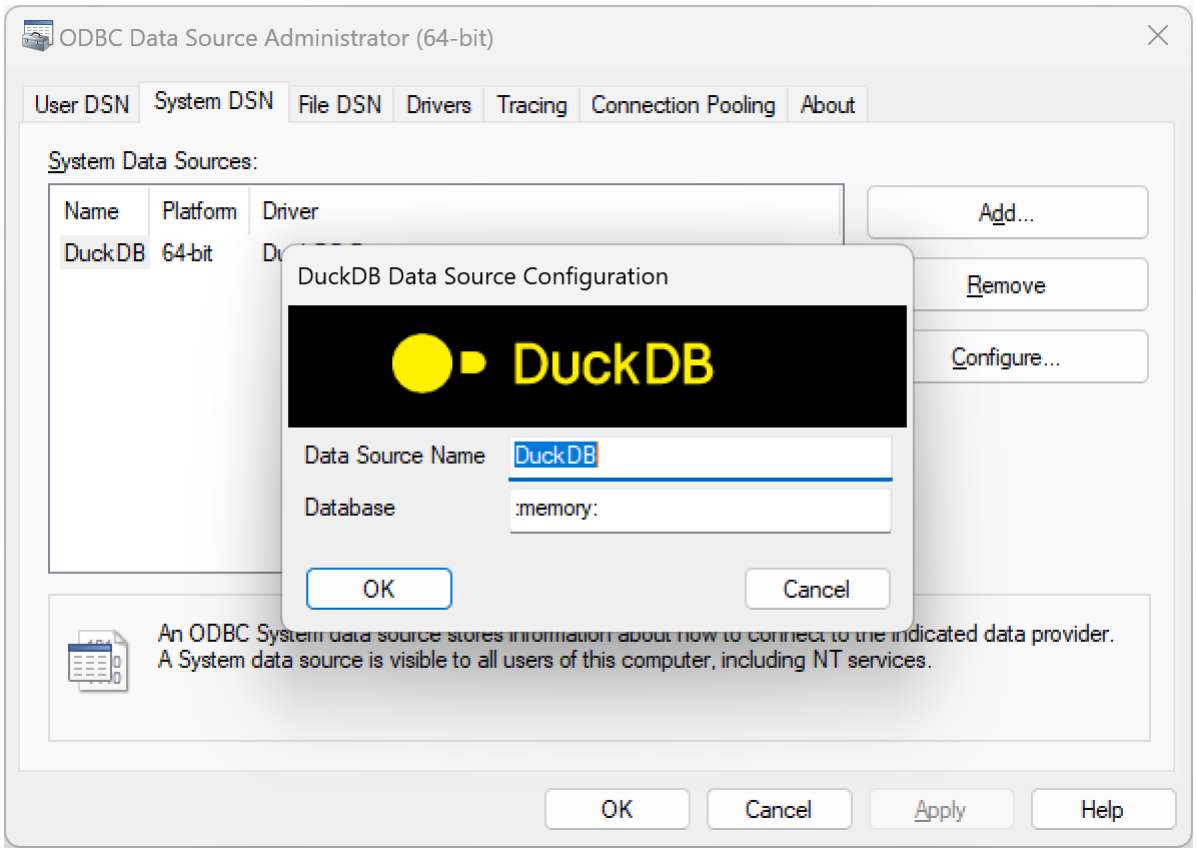
Default DuckDB DSN

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

There are two ways to configure the ODBC driver, either by altering the registry keys as detailed below, or by connecting with `SQLDriverConnect`. A combination of the two is also possible.

Furthermore, the ODBC driver supports all the configuration options included in DuckDB.

> If a configuration is set in both the connection string passed to `SQLDriverConnect` and in the `odbc.ini` file, the one passed to `SQLDriverConnect` will take precedence.

For the details of the configuration parameters, see the ODBC configuration page.

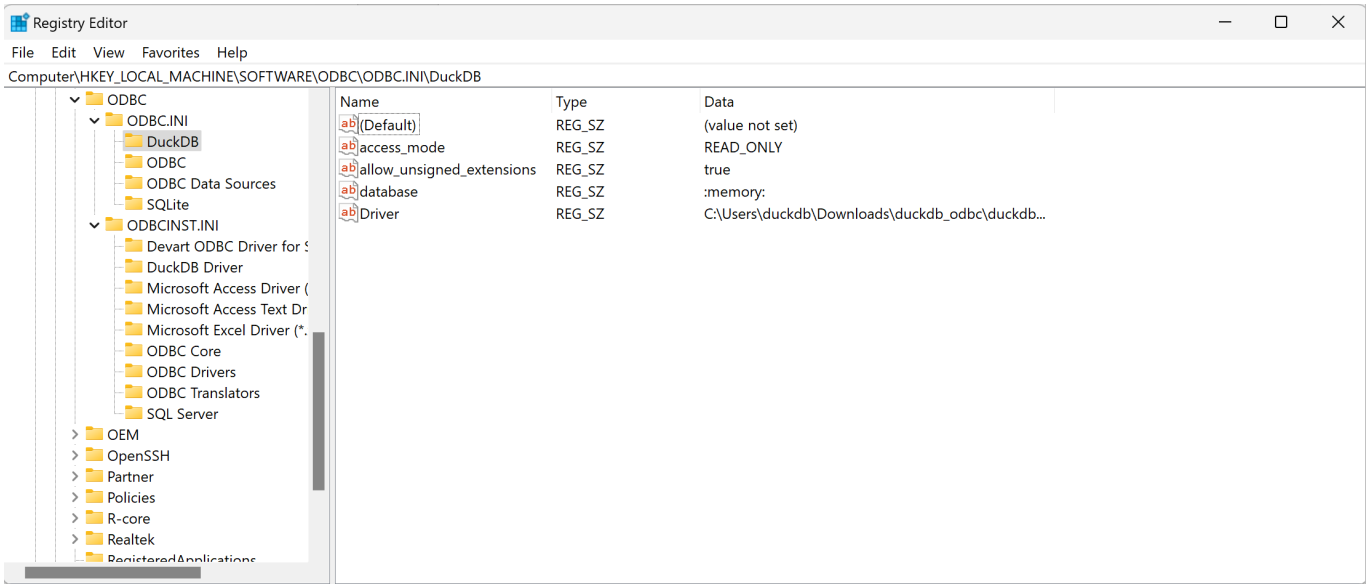
Registry Keys

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.

Updating the ODBC Driver

When a new version of the ODBC driver is released, installing the new version will overwrite the existing one. However, the installer doesn't always update the version number in the registry. To ensure the correct version is used, check that `HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\DuckDB Driver` has the most recent version, and `HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\DuckDB\Driver` has the correct path to the new driver.

ODBC API on macOS

1. A driver manager is required to manage communication between applications and the ODBC driver. DuckDB supports `unixODBC`, which is a complete ODBC driver manager for macOS and Linux. Users can install it from the command line via Homebrew:

   ```
   brew install unixodbc
   ```

2. DuckDB releases a universal ODBC driver for macOS (supporting both Intel and Apple Silicon CPUs). To download it, run:

   ```
   wget https://github.com/duckdb/duckdb/releases/download/v{{ site.currentduckdbversion }}/duckdb_odbc-osx-universal.zip
   ```

3. The archive contains the `libduckdb_odbc.dylib` artifact. To extract it to a directory, run:

   ```
   mkdir duckdb_odbc && unzip duckdb_odbc-osx-universal.zip -d duckdb_odbc
   ```

4. There are two ways to configure the ODBC driver, either by initializing via the configuration files, or by connecting with `SQLDriverConnect`. A combination of the two is also possible.

   Furthermore, the ODBC driver supports all the configuration options included in DuckDB.

   > If a configuration is set in both the connection string passed to `SQLDriverConnect` and in the `odbc.ini` file, the one passed to `SQLDriverConnect` will take precedence.

   For the details of the configuration parameters, see the ODBC configuration page.

5. After the configuration, to validate the installation, it is possible to use an ODBC client. unixODBC uses 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
```

ODBC Configuration

This page documents the files using the ODBC configuration, `odbc.ini` and `odbcinst.ini`. These are either placed in the home directory as dotfiles (`.odbc.ini` and `.odbcinst.ini`, respectively) or in a system directory. For platform-specific details, see the pages for Linux, macOS, and Windows.

`odbc.ini` and `.odbc.ini`

The `odbc.ini` file contains the DSNs for the drivers, which can have specific knobs. An example of `odbc.ini` with DuckDB:

```
[DuckDB]
Driver = DuckDB Driver
Database = :memory:
access_mode = read_only
allow_unsigned_extensions = true
```

The lines correspond to the following parameters:

- `[DuckDB]`: between the brackets is a DSN for the DuckDB.
- `Driver`: Describes the driver's name, as well as where to find the configurations in the `odbcinst.ini`.
- `Database`: Describes the database name used by DuckDB, can also be a file path to a `.db` in the system.
- `access_mode`: The mode in which to connect to the database.
- `allow_unsigned_extensions`: Allow the use of unsigned extensions.

`odbcinst.ini` and `.odbcinst.ini`

The `odbcinst.ini` file 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.

Example of `odbcinst.ini` with the DuckDB:

```
[ODBC]
Trace = yes
TraceFile = /tmp/odbctrace
```

```
[DuckDB Driver]
Driver = /path/to/libduckdb_odbc.dylib
```

The lines correspond to the following parameters:

- `[ODBC]`: The DM configuration section.
- `Trace`: 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. Change to match your configuration.

Configuration

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 be reset to their original values using the RESET statement. The values of configuration options can be queried via the `current_setting()` scalar function or using the `duckdb_settings()` table function.

Examples

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.

```
SET default_null_order = 'nulls_last';
```

Return the current value of a specific setting.

```
SELECT current_setting('threads') AS threads;
```

threads
10

Query a specific setting.

```
SELECT *
FROM duckdb_settings()
WHERE name = 'threads';
```

name	value	description	input_type	scope
threads	1	The number of total threads used by the system.	BIGINT	GLOBAL

Show a list of all available settings.

```
SELECT *
FROM duckdb_settings();
```

Reset the memory limit of the system back to the default.

```
RESET memory_limit;
```

Secrets Manager

DuckDB has a Secrets manager, which provides a unified user interface for secrets across all backends (e.g., AWS S3) that use them.

Configuration Reference

Configuration options come with different default scopes: `GLOBAL` and `LOCAL`. Below is a list of all available configuration options by scope.

Global Configuration Options

Name	Description	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`	`128.0 MiB`
`allow_community_extensions`	Allow to load community built extensions	`BOOLEAN`	`true`
`allow_extensions_metadata_mismatch`	Allow to load extensions with not compatible metadata	`BOOLEAN`	`false`
`allow_persistent_secrets`	Allow the creation of persistent secrets, that are stored and loaded on restarts	`BOOLEAN`	`true`
`allow_unredacted_secrets`	Allow printing unredacted secrets	`BOOLEAN`	`false`
`allow_unsigned_extensions`	Allow to load extensions with invalid or missing signatures	`BOOLEAN`	`false`
`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`	
`ca_cert_file`	Path to a custom certificate file for self-signed certificates.	`VARCHAR`	
`checkpoint_threshold`, `wal_autocheckpoint`	The WAL size threshold at which to automatically trigger a checkpoint (e.g., 1GB)	`VARCHAR`	`16.0 MiB`
`custom_extension_repository`	Overrides the custom endpoint for remote extension installation	`VARCHAR`	

Name	Description	Type	Default value
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
default_secret_storage	Allows switching the default storage for secrets	VARCHAR	local_file
disabled_filesystems	Disable specific file systems preventing access (e.g., LocalFileSystem)	VARCHAR	
duckdb_api	DuckDB API surface	VARCHAR	cli
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_macro_dependencies	Enable created MACROs to create dependencies on the referenced objects (such as tables)	BOOLEAN	false
enable_object_cache	Whether or not object cache is used to cache e.g., Parquet metadata	BOOLEAN	false
enable_server_cert_verification	Enable server side certificate verification.	BOOLEAN	false
enable_view_dependencies	Enable created VIEWs to create dependencies on the referenced objects (such as tables)	BOOLEAN	false
extension_directory	Set the directory to store extensions in	VARCHAR	
external_threads	The number of external threads that work on DuckDB tasks.	BIGINT	1
force_download	Forces upfront download of file	BOOLEAN	false
http_keep_alive	Keep alive connections. Setting this to false can help when running into connection failures	BOOLEAN	true
http_retries	HTTP retries on I/O error	UBIGINT	3
http_retry_backoff	Backoff factor for exponentially increasing retry wait time	FLOAT	4
http_retry_wait_ms	Time between retries	UBIGINT	100
http_timeout	HTTP timeout read/write/connection/retry	UBIGINT	30000
immediate_transaction_mode	Whether transactions should be started lazily when needed, or immediately when BEGIN TRANSACTION is called	BOOLEAN	false
lock_configuration	Whether or not the configuration can be altered	BOOLEAN	false
max_memory, memory_limit	The maximum memory of the system (e.g., 1GB)	VARCHAR	80% of RAM
max_temp_directory_size	The maximum amount of data stored inside the 'temp_directory' (when set) (e.g., 1GB)	VARCHAR	0 bytes
old_implicit_casting	Allow implicit casting to/from VARCHAR	BOOLEAN	false

Name	Description	Type	Default value
password	The password to use. Ignored for legacy compatibility.	VARCHAR	NULL
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
s3_access_key_id	S3 Access Key ID	VARCHAR	
s3_endpoint	S3 Endpoint	VARCHAR	
s3_region	S3 Region	VARCHAR	us-east-1
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)	VARCHAR	800GB
s3_uploader_max_parts_per_file	S3 Uploader max parts per file (between 1 and 10000)	UBIGINT	10000
s3_uploader_thread_limit	S3 Uploader global thread limit	UBIGINT	50
s3_url_compatibility_mode	Disable Globs and Query Parameters on S3 URLs	BOOLEAN	false
s3_url_style	S3 URL style	VARCHAR	vhost
s3_use_ssl	S3 use SSL	BOOLEAN	true
secret_directory	Set the directory to which persistent secrets are stored	VARCHAR	~/.duckdb/stored_secrets
storage_compatibility_version	Serialize on checkpoint with compatibility for a given duckdb version	VARCHAR	v0.10.2
temp_directory	Set the directory to which to write temp files	VARCHAR	<database_name>.tmp or .tmp (in in-memory mode)
threads, worker_threads	The number of total threads used by the system.	BIGINT	# CPU cores
username, user	The username to use. Ignored for legacy compatibility.	VARCHAR	NULL

Local Configuration Options

Name	Description	Type	Default value
enable_http_logging	Enables HTTP logging	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

Name	Description	Type	Default value
`errors_as_json`	Output error messages as structured **JSON** instead of as a raw string	`BOOLEAN`	`false`
`explain_output`	Output of EXPLAIN statements (**ALL**, **OPTIMIZED_ONLY**, **PHYSICAL_ONLY**)	`VARCHAR`	`physical_only`
`file_search_path`	A comma separated list of directories to search for input files	`VARCHAR`	
`home_directory`	Sets the home directory used by the system	`VARCHAR`	
`http_logging_output`	The file to which HTTP logging output should be saved, or empty to print to the terminal	`VARCHAR`	
`integer_division`	Whether or not the / operator defaults to integer division, or to floating point division	`BOOLEAN`	`false`
`log_query_path`	Specifies the path to which queries should be logged (default: NULL, 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`
`ordered_aggregate_threshold`	The number of rows to accumulate before sorting, used for tuning	`UBIGINT`	`262144`
`partitioned_write_flush_threshold`	The threshold in number of rows after which we flush a thread state when writing using **PARTITION_BY**	`BIGINT`	`524288`
`perfect_ht_threshold`	Threshold in bytes for when to use a perfect hash table	`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	`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`
`profile_output`, `profiling_output`	The file to which profile output should be saved, or empty to print to the terminal	`VARCHAR`	
`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`
`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`	

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.

PRAGMA statements that assign a value to an option can also be issued using the SET statement and the value of an option can be retrieved using `SELECT current_setting(option_name)`.

For DuckDB's built in configuration options, see the Configuration Reference. DuckDB extensions may register additional configuration options. These are documented in the respective extensions' documentation pages.

This page contains the supported PRAGMA settings.

Metadata

Schema Information

List all databases:

```
PRAGMA database_list;
```

List all tables:

```
PRAGMA show_tables;
```

List all tables, with extra information, similarly to DESCRIBE:

```
PRAGMA show_tables_expanded;
```

To list all functions:

```
PRAGMA functions;
```

Table Information

Get info for a specific table:

```
PRAGMA table_info('table_name');
CALL pragma_table_info('table_name');
```

`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
```

To also show table structure, but in a slightly different format (included for compatibility):

```
PRAGMA show('table_name');
```

Database Size

Get the file and memory size of each database:

```
SET 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
```

Storage Information

To get storage information:

```
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
`block_offset`	`BIGINT`	empty unless persistent

See Storage for more information.

Show Databases

The following statement is equivalent to the `SHOW DATABASES` statement:

```
PRAGMA show_databases;
```

Resource Management

Memory Limit

Set the memory limit for the buffer manager:

```
SET memory_limit = '1GB';
SET max_memory = '1GB';
```

> **Warning.** The specified memory limit is only applied to the buffer manager. For most queries, the buffer manager handles the majority of the data processed. However, certain in-memory data structures such as vectors and query results are allocated outside of the buffer manager. Additionally, aggregate functions with complex state (e.g., `list`, `mode`, `quantile`, `string_agg`, and `approx` functions) use memory outside of the buffer manager. Therefore, the actual memory consumption can be higher than the specified memory limit.

Threads

Set the amount of threads for parallel query execution:

```
SET threads = 4;
```

Collations

List all available collations:

```
PRAGMA collations;
```

Set the default collation to one of the available ones:

```
SET default_collation = 'nocase';
```

Default Ordering for NULLs

Set the default ordering for NULLs to be either `NULLS FIRST` or `NULLS LAST`:

```
SET default_null_order = 'NULLS FIRST';
SET default_null_order = 'NULLS LAST';
```

Set the default result set ordering direction to `ASCENDING` or `DESCENDING`:

```
SET default_order = 'ASCENDING';
SET default_order = 'DESCENDING';
```

Implicit Casting to VARCHAR

Prior to version 0.10.0, DuckDB would automatically allow any type to be implicitly cast to VARCHAR during function binding. As a result it was possible to e.g., compute the substring of an integer without using an explicit cast. For version v0.10.0 and later an explicit cast is needed instead. To revert to the old behaviour that performs implicit casting, set the `old_implicit_casting` variable to `true`:

```
SET old_implicit_casting = true;
```

Information on DuckDB

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, e.g., `osx_arm64`. The format of this identifier matches the platform name as described on the extension loading explainer:

```
PRAGMA platform;
CALL pragma_platform();
```

User Agent

The following statement returns the user agent information, e.g., `duckdb/v0.10.0(osx_arm64)`:

```
PRAGMA user_agent;
```

Metadata Information

The following statement returns information on the metadata store (`block_id`, `total_blocks`, `free_blocks`, and `free_list`):

```
PRAGMA metadata_info;
```

Progress Bar

Show progress bar when running queries:

```
PRAGMA enable_progress_bar;
```

Or:

```
PRAGMA enable_print_progress_bar;
```

Don't show a progress bar for running queries:

```
PRAGMA disable_progress_bar;
```

Or:

```
PRAGMA disable_print_progress_bar;
```

Profiling Queries

Explain Plan Output

The output of `EXPLAIN` output can be configured to show only the physical plan. This is the default configuration:

```
SET explain_output = 'physical_only';
```

To only show the optimized query plan:

```
SET explain_output = 'optimized_only';
```

To show all query plans:

```
SET explain_output = 'all';
```

Profiling

Enable Profiling

To enable profiling:

```
PRAGMA enable_profiling;
```

Or:

```
PRAGMA enable_profile;
```

Profiling Format

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.

To return the logical query plan as JSON:

```
SET enable_profiling = 'json';
```

To return the logical query plan:

```
SET enable_profiling = 'query_tree';
```

To return the physical query plan:

```
SET enable_profiling = 'query_tree_optimizer';
```

Disable Profiling

To disable profiling:

```
PRAGMA disable_profiling;
```

Or:

```
PRAGMA disable_profile;
```

Profiling Output

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.

> **Warning.** 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:

```
SET profiling_output = '/path/to/file.json';
SET profile_output = '/path/to/file.json';
```

Profiling Mode

By default, a limited amount of profiling information is provided (`standard`). For more details, use the detailed profiling mode by setting `profiling_mode` to `detailed`. The output of this mode shows how long it takes to apply certain optimizers on the query tree and how long physical planning takes:

```
SET profiling_mode = 'detailed';
```

Query Optimization

Optimizer

To disable the query optimizer:

```
PRAGMA disable_optimizer;
```

To enable the query optimizer:

```
PRAGMA enable_optimizer;
```

Selectively Disabling Optimizers

The `disabled_optimizers` option allows selectively disabling optimization steps. For example, to disable `filter_pushdown` and `statistics_propagation`, run:

```
SET disabled_optimizers = 'filter_pushdown,statistics_propagation';
```

The available optimizations can be queried using the `duckdb_optimizers()` table function.

> **Warning.** The `disabled_optimizers` option should only be used for debugging performance issues and should be avoided in production.

Logging

Set a path for query logging:

```
SET log_query_path = '/tmp/duckdb_log/';
```

Disable query logging:

```
SET log_query_path = '';
```

Full-Text Search Indexes

The `create_fts_index` and `drop_fts_index` options are only available when the `fts` extension is loaded. Their usage is documented on the Full-Text Search extension page.

Verification

Verification of External Operators

Enable verification of external operators:

```
PRAGMA verify_external;
```

Disable verification of external operators:

```
PRAGMA disable_verify_external;
```

Verification of Round-Trip Capabilities

Enable verification of round-trip capabilities for supported logical plans:

```
PRAGMA verify_serializer;
```

Disable verification of round-trip capabilities:

```
PRAGMA disable_verify_serializer;
```

Object Cache

Enable caching of objects for e.g., Parquet metadata:

```
PRAGMA enable_object_cache;
```

Disable caching of objects:

```
PRAGMA disable_object_cache;
```

Checkpointing

Force Checkpoint

When `CHECKPOINT` is called when no changes are made, force a checkpoint regardless:

```
PRAGMA force_checkpoint;
```

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

By default, DuckDB uses a temporary directory named `<database_file_name>.tmp` to spill to disk, located in the same directory as the database file. To change this, use:

```
SET temp_directory = '/path/to/temp_dir.tmp/';
```

Returning Errors as JSON

The `errors_as_json` option can be set to obtain error information in raw JSON format. For certain errors, extra information or decomposed information is provided for easier machine processing. For example:

```
SET errors_as_json = true;
```

Then, running a query that results in an error produces a JSON output:

```
SELECT * FROM nonexistent_tbl;
```

```
{
   "exception_type":"Catalog",
   "exception_message":"Table with name nonexistent_tbl does not exist!\nDid you mean \"temp.information_
   schema.tables\"?",
   "name":"nonexistent_tbl",
   "candidates":"temp.information_schema.tables",
   "position":"14",
   "type":"Table",
   "error_subtype":"MISSING_ENTRY"
}
```

Query Verification (for Development)

The following PRAGMAs are mostly used for development and internal testing.

Enable query verification:

```
PRAGMA enable_verification;
```

Disable query verification:

```
PRAGMA disable_verification;
```

Enable force parallel query processing:

```
PRAGMA verify_parallelism;
```

Disable force parallel query processing:

```
PRAGMA disable_verify_parallelism;
```

Secrets Manager

The **Secrets manager** provides a unified user interface for secrets across all backends that use them. Secrets can be scoped, so different storage prefixes can have different secrets, allowing for example to join data across organizations in a single query. Secrets can also be persisted, so that they do not need to be specified every time DuckDB is launched.

> **Warning.** Persistent secrets are stored in unencrypted binary format on the disk.

Secrets

Types of Secrets

Secrets are typed, their type identifies which service they are for. Currently, the following cloud services are available:

- AWS S3 (S3), through the `httpfs` extension
- Azure Blob Storage (AZURE), through the `azure` extension
- Cloudflare R2 (R2), through the `httpfs` extension
- Google Cloud Storage (GCS), through the `httpfs` extension
- Hugging Face (HUGGINGFACE), through the `httpfs` extension

For each type, there are one or more "secret providers" that specify how the secret is created. Secrets can also have an optional scope, which is a file path prefix that the secret applies to. When fetching a secret for a path, the secret scopes are compared to the path, returning the matching secret for the path. In the case of multiple matching secrets, the longest prefix is chosen.

Creating a Secret

Secrets can be created using the `CREATE SECRET` SQL statement. Secrets can be **temporary** or **persistent**. Temporary secrets are used by default – and are stored in-memory for the life span of the DuckDB instance similar to how settings worked previously. Persistent secrets are stored in **unencrypted binary format** in the `~/.duckdb/stored_secrets` directory. On startup of DuckDB, persistent secrets are read from this directory and automatically loaded.

Secret Providers

To create a secret, a **Secret Provider** needs to be used. A Secret Provider is a mechanism through which a secret is generated. To illustrate this, for the S3, GCS, R2, and AZURE secret types, DuckDB currently supports two providers: `CONFIG` and `CREDENTIAL_CHAIN`. The `CONFIG` provider requires the user to pass all configuration information into the `CREATE SECRET`, whereas the `CREDENTIAL_CHAIN` provider will automatically try to fetch credentials. When no Secret Provider is specified, the `CONFIG` provider is used. For more details on how to create secrets using different providers check out the respective pages on httpfs and azure.

Temporary Secrets

To create a temporary unscoped secret to access S3, we can now use the following:

```
CREATE SECRET my_secret (
    TYPE S3,
    KEY_ID 'my_secret_key',
    SECRET 'my_secret_value',
```

```
    REGION 'my_region'
);
```

Note that we implicitly use the default `CONFIG` secret provider here.

Persistent Secrets

In order to persist secrets between DuckDB database instances, we can now use the `CREATE PERSISTENT SECRET` command, e.g.:

```
CREATE PERSISTENT SECRET my_persistent_secret (
    TYPE S3,
    KEY_ID 'my_secret_key',
    SECRET 'my_secret_value'
);
```

This will write the secret (unencrypted) to the `~/.duckdb/stored_secrets` directory.

Deleting Secrets

Secrets can be deleted using the DROP SECRET statement, e.g.:

```
DROP PERSISTENT SECRET my_persistent_secret;
```

Creating Multiple Secrets for the Same Service Type

If two secrets exist for a service type, the scope can be used to decide which one should be used. For example:

```
CREATE SECRET secret1 (
    TYPE S3,
    KEY_ID 'my_secret_key1',
    SECRET 'my_secret_value1',
    SCOPE 's3://my-bucket'
);
```

```
CREATE SECRET secret2 (
    TYPE S3,
    KEY_ID 'my_secret_key2',
    SECRET 'my_secret_value2',
    SCOPE 's3://my-other-bucket'
);
```

Now, if the user queries something from `s3://my-other-bucket/something`, secret `secret2` will be chosen automatically for that request. To see which secret is being used, the `which_secret` scalar function can be used, which takes a path and a secret type as parameters:

```
SELECT which_secret('s3://my-other-bucket/file.parquet', 's3');
```

Listing Secrets

Secrets can be listed using the built-in table-producing function, e.g., by using the `duckdb_secrets()` table function:

```
FROM duckdb_secrets();
```

Sensitive information will be redacted.

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.

> DuckDB's SQL dialect closely follows the conventions of the PostgreSQL dialect. The few exceptions to this are listed on the PostgreSQL compatibility page.

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.

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 keywords and identifiers. When returning identifiers, their original cases are preserved.

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);
```

> **Tip.** 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.

Alternatively, you can use the `COPY` statement. This is faster for large amounts of data because the `COPY` command is optimized for bulk loading 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	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0.0	1994-11-29
Hayward	37	54	NULL	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	temp_avg	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	temp_lo	temp_hi	prcp	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	temp_lo	temp_hi	prcp	date
Hayward	37	54	NULL	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
San Francisco
Hayward

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
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
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
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	NULL	1994-11-29	NULL	NULL	NULL

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)
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
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)
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)
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 in the pattern matching page.

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
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0.0	1994-11-29
Hayward	37	54	NULL	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	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0.0	1994-11-29

One should be cautious when issuing statements of the following form:

```
DELETE FROM <table_name>;
```

> **Warning.** 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

ANALYZE Statement

The ANALYZE statement recomputes the statistics on DuckDB's tables.

Usage

The statistics recomputed by the ANALYZE statement are only used for join order optimization. It is therefore recommended to recompute these statistics for improved join orders, especially after performing large updates (inserts and/or deletes).

To recompute the statistics, run:

```
ANALYZE;
```

ALTER TABLE Statement

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.

> **Warning.** 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

> The ADD CONSTRAINT and DROP CONSTRAINT clauses are not yet supported in DuckDB.

ALTER VIEW Statement

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 Statement

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 the `sqlite` extension for more information):

```
ATTACH 'sqlite_file.db' AS sqlite_db (TYPE SQLITE);
```

Attach the database `file.db` if inferred database alias `file` does not yet exist:

```
ATTACH IF NOT EXISTS 'file.db';
```

Attach the database `file.db` if explicit database alias `file.db` does not yet exist:

```
ATTACH IF NOT EXISTS 'file.db' AS file_db;
```

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;
```

Attach

Attach 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.

Note that it is not possible to detach from the default database: if you would like to do so, issue the USE statement to change the default database to another one. For example, if you are connected to a persistent database, you may change to an in-memory database by issuing:

```
ATTACH ':memory:' AS memory_db;
USE memory_db;
```

> **Warning.** Closing the connection, e.g., invoking the `close()` function in Python, does not release the locks held on the database files as the file handles are held by the main DuckDB instance (in Python's case, the `duckdb` module).

Detach Syntax

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.

Also note the rules on identifiers and database names in particular.

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 Statement

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');
```

Select only the functions where the name starts with ST_:

```
SELECT function_name, parameters, parameter_types, return_type
FROM duckdb_functions()
WHERE function_name LIKE 'ST_%';
```

Syntax

CHECKPOINT Statement

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 option for further behavior modification.

Reclaiming Space

When performing a checkpoint (automatic or otherwise), the space occupied by deleted rows is partially reclaimed. Note that this does not remove all deleted rows, but rather merges row groups that have a significant amount of deletes together. In the current implementation this requires ~25% of rows to be deleted in adjacent row groups.

When running in in-memory mode, checkpointing has no effect, hence it does not reclaim space after deletes in in-memory databases.

> **Warning.** The VACUUM statement does *not* trigger vacuuming deletes and hence does not reclaim space.

COMMENT ON Statement

The `COMMENT ON` statement allows adding metadata to catalog entries (tables, columns, etc.). It follows the PostgreSQL syntax.

Examples

Create a comment on a `TABLE`:

```
COMMENT ON TABLE test_table IS 'very nice table';
```

Create a comment on a `COLUMN`:

```
COMMENT ON COLUMN test_table.test_table_column IS 'very nice column';
```

Create a comment on a `VIEW`:

```
COMMENT ON VIEW test_view IS 'very nice view';
```

Create a comment on an `INDEX`:

```
COMMENT ON INDEX test_index IS 'very nice index';
```

Create a comment on a `SEQUENCE`:

```
COMMENT ON SEQUENCE test_sequence IS 'very nice sequence';
```

Create a comment on a `TYPE`:

```
COMMENT ON TYPE test_type IS 'very nice type';
```

Create a comment on a `MACRO`:

```
COMMENT ON MACRO test_macro IS 'very nice macro';
```

Create a comment on a `MACRO TABLE`:

```
COMMENT ON MACRO TABLE test_table_macro IS 'very nice table macro';
```

To unset a comment, set it to `NULL`, e.g.:

```
COMMENT ON TABLE test_table IS NULL;
```

Reading Comments

Comments can be read by querying the `comment` column of the respective metadata functions:

List comments on `TABLE`s:

```
SELECT comment FROM duckdb_tables();
```

List comments on `COLUMN`s:

```
SELECT comment FROM duckdb_columns();
```

List comments on `VIEW`s:

```
SELECT comment FROM duckdb_views();
```

List comments on `INDEX`s:

```
SELECT comment FROM duckdb_indexes();
```

List comments on `SEQUENCE`s:

```
SELECT comment FROM duckdb_sequences();
```

List comments on `TYPE`s:

```
SELECT comment FROM duckdb_types();
```

List comments on `MACRO`s:

```
SELECT comment FROM duckdb_functions();
```

List comments on `MACRO TABLE`s:

```
SELECT comment FROM duckdb_functions();
```

Limitations

The `COMMENT ON` statement currently has the following limitations:

- It is not possible to comment on schemas or databases.
- It is not possible to comment on things that have a dependency (e.g., a table with an index).

Syntax

COPY Statement

Examples

Read a CSV file into the `lineitem` table, using auto-detected CSV options:

```
COPY lineitem FROM 'lineitem.csv';
```

Read a CSV file into the `lineitem` table, using manually specified CSV options:

```
COPY lineitem FROM 'lineitem.csv' (DELIMITER '|');
```

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);
```

Read a CSV file into the `lineitem` table, using double quotes:

```
COPY lineitem FROM "lineitem.csv";
```

Read a CSV file into the `lineitem` table, omitting quotes:

```
COPY lineitem FROM lineitem.csv;
```

Write a table to a CSV file:

```
COPY lineitem TO 'lineitem.csv' (FORMAT CSV, DELIMITER '|', HEADER);
```

Write a table to a CSV file, using double quotes:

```
COPY lineitem TO "lineitem.csv";
```

Write a table to a CSV file, omitting quotes:

```
COPY lineitem TO lineitem.csv;
```

Write the result of a query to a Parquet file:

```
COPY (SELECT l_orderkey, l_partkey FROM lineitem) TO 'lineitem.parquet' (COMPRESSION ZSTD);
```

Copy the entire content of database `db1` to database `db2`:

```
COPY FROM DATABASE db1 TO db2;
```

Copy only the schema (catalog elements) but not any data:

```
COPY FROM DATABASE db1 TO db2 (SCHEMA);
```

Overview

`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 a CSV file with a header:

```
COPY lineitem TO 'lineitem.csv';
```

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 '|');
```

Use tab separators to create a TSV file without a header:

```
COPY lineitem TO 'lineitem.tsv' (DELIMITER '\t', HEADER false);
```

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' (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);
```

`COPY ... TO` 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
`file_size_bytes`	If this parameter is set, the COPY process creates a directory which will contain the exported files. If a file exceeds the set limit (specified as bytes such as `1000` or in human-readable format such as `1k`), the process creates a new file in the directory. This parameter works in combination with `per_thread_output`. Note that the size is used as an approximation, and files can be occasionally slightly over the limit.	VARCHAR or BIGINT	(empty)
`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
`overwrite_or_ignore`	Whether or not to allow overwriting a directory if one already exists. Only has an effect when used with `partition_by`.	BOOL	`false`
`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`

Syntax

COPY FROM DATABASE ... TO

The `COPY FROM DATABASE ... TO` statement copies the entire content from one attached database to another attached database. This includes the schema, including constraints, indexes, sequences, macros, and the data itself.

```
ATTACH 'db1.db' AS db1;
CREATE TABLE db1.tbl AS SELECT 42 AS x, 3 AS y;
CREATE MACRO db1.two_x_plus_y(x, y) AS 2 * x + y;

ATTACH 'db2.db' AS db2;
COPY FROM DATABASE db1 TO db2;
SELECT db2.two_x_plus_y(x, y) AS z FROM db2.tbl;
```

z
87

To only copy the **schema** of `db1` to `db2` but omit copying the data, add `SCHEMA` to the statement:

```
COPY FROM DATABASE db1 TO db2 (SCHEMA);
```

Syntax

Format-Specific Options

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`
`dateformat`	Specifies the date format to use when writing dates. See Date Format	VARCHAR	(empty)
`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	`"`
`force_quote`	The list of columns to always add quotes to, even if not required.	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`
`field_ids`	The `field_id` for each column. Pass `auto` to attempt to infer automatically.	STRUCT	(empty)
`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`
`row_group_size`	The target size, i.e., number of rows, of each row group.	BIGINT	122880

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
`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`
`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)

Limitations

COPY does not support copying between tables. To copy between tables, use an `INSERT statement`:

```
INSERT INTO tbl2
    FROM tbl1;
```

CREATE MACRO Statement

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

Scalar Macros

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. Note that parameter names get priority over column names. To work around this, disambiguate using the table name.

```
CREATE MACRO plus_one(a) AS (WITH cte AS (SELECT 1 AS a) SELECT cte.a + a FROM cte);
```

Macros are schema-dependent, and have an alias, `FUNCTION`:

```
CREATE FUNCTION main.my_avg(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;
```

Pass an argument as a list: SELECT * FROM get_users([1, 5]):

```
CREATE MACRO get_users(i) AS TABLE
    SELECT * FROM users WHERE uid IN (SELECT unnest(i));
```

Syntax

Macros allow you to create shortcuts for combinations of expressions.

```
CREATE MACRO add(a) AS a + b;
```

```
Binder Error: Referenced column "b" not found in FROM clause!
```

This works:

```
CREATE MACRO add(a, b) AS a + b;
```

Usage example:

```
SELECT add(1, 2) AS x;
```

x
3

However, this fails:

```
SELECT add('hello', 3);
```

```
Binder Error: Could not choose a best candidate function for the function call "+(STRING_LITERAL, INTEGER_
LITERAL)". In order to select one, please add explicit type casts.
    Candidate functions:
    +(DATE, INTEGER) -> DATE
    +(INTEGER, INTEGER) -> INTEGER
```

Macros 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);
```

The following will throw an error:

```
SELECT add_default(40, 2);
```

```
Binder Error: Macro function 'add_default(a)' requires a single positional argument, but 2 positional
arguments were provided.
```

Default parameters must used by assigning them like the following:

```
SELECT add_default(40, b := 2) AS x;
```

x
42

However, the following fails:

```
SELECT add_default(b := 2, 40);
```

```
Binder Error: Positional parameters cannot come after parameters with a default value!
```

The order of default parameters does not matter:

```
CREATE MACRO triple_add(a, b := 5, c := 10) AS a + b + c;
```

```
SELECT triple_add(40, c := 1, b := 1) AS x;
```

x
42

When macros 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) AS x;
```

Internally, add is replaced with its definition of `a + b`:

```
SELECT a + b; AS x
```

Then, the parameters are replaced by the supplied arguments:

```
SELECT 40 + 2 AS x;
```

Limitations

Using Named Parameters

Currently, positional macro parameters can only be used positionally, and named parameters can only be used by supplying their name. Therefore, the following will not work:

```
CREATE MACRO my_macro(a, b := 42) AS (a + b);
SELECT my_macro(32, 52);
```

```
Error: Binder Error: Macro function 'my_macro(a)' requires a single positional argument, but 2 positional arguments were provided.
```

Using Subquery Macros

If a MACRO is defined as a subquery, it cannot be invoked in a table function. DuckDB will return the following error:

```
Binder Error: Table function cannot contain subqueries
```

CREATE SCHEMA Statement

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 SECRET Statement

The `CREATE SECRET` statement creates a new secret in the Secrets Manager.

Syntax for CREATE SECRET

Syntax for DROP SECRET

CREATE SEQUENCE Statement

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 error, e.g.:

```
Sequence Error: nextval: 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;
```

Creating and Dropping Sequences

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;
```

Using Sequences for Primary Keys

Sequences can provide an integer primary key for a table. For example:

```
CREATE SEQUENCE id_sequence START 1;
CREATE TABLE tbl (id INTEGER DEFAULT nextval('id_sequence'), s VARCHAR);
INSERT INTO tbl (s) VALUES ('hello'), ('world');
SELECT * FROM tbl;
```

The script results in the following table:

id	s
1	hello
2	world

Sequences can also be added using the `ALTER TABLE` statement. The following example adds an `id` column and fills it with values generated by the sequence:

```
CREATE TABLE tbl (s VARCHAR);
INSERT INTO tbl VALUES ('hello'), ('world');
CREATE SEQUENCE id_sequence START 1;
ALTER TABLE tbl ADD COLUMN id INTEGER DEFAULT nextval('id_sequence');
SELECT * FROM tbl;
```

This script results in the same table as the previous example.

Selecting the Next Value

To select the next number from a sequence, use `nextval`:

```
CREATE SEQUENCE serial START 1;
SELECT nextval('serial') AS nextval;
```

nextval
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.

```
CREATE SEQUENCE serial START 1;
SELECT nextval('serial') AS nextval;
SELECT currval('serial') AS currval;
```

currval
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
`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.
`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
`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.
`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.
`name`	The name (optionally schema-qualified) of the sequence to be created.
`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.
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.

> Sequences are based on `BIGINT` arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807).

CREATE TABLE Statement

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 table with `CREATE TABLE ... AS SELECT` (CTAS):

```
CREATE TABLE t1 AS
    SELECT 42 AS i, 84 AS j;
```

Create a table from a CSV file (automatically detecting column names and types):

```sql
CREATE TABLE t1 AS
    SELECT *
    FROM read_csv('path/file.csv');
```

We can use the FROM-first syntax to omit `SELECT *`:

```sql
CREATE TABLE t1 AS
    FROM read_csv('path/file.csv');
```

Copy the schema of `t2` to `t1`:

```sql
CREATE TABLE t1 AS
    FROM t2
    LIMIT 0;
```

Temporary Tables

Temporary tables can be created using the `CREATE TEMP TABLE` or the `CREATE TEMPORARY 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 (automatically detecting column names and types):

```sql
CREATE TEMP TABLE t1 AS
    SELECT *
    FROM read_csv('path/file.csv');
```

Allow temporary tables to off-load excess memory to disk:

```sql
SET temp_directory = '/path/to/directory/';
```

Temporary tables are part of the `temp.main` schema. While discouraged, their names can overlap with the names of the regular database tables. In these cases, use their fully qualified name, e.g., `temp.main.t1`, for disambiguation.

`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:

```sql
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:

```sql
CREATE TABLE IF NOT EXISTS t1 (i INTEGER, j INTEGER);
```

CREATE TABLE ... AS SELECT (CTAS)

DuckDB supports the `CREATE TABLE ... AS SELECT` syntax, also known as "CTAS":

```
CREATE TABLE nums AS
    SELECT i
    FROM range(0, 3) t(i);
```

This syntax can be used in combination with the CSV reader, the shorthand to read directly from CSV files without specifying a function, the FROM-first syntax, and the HTTP(S) support, yielding concise SQL commands such as the following:

```
CREATE TABLE flights AS
    FROM 'https://duckdb.org/data/flights.csv';
```

The CTAS construct also works with the `OR REPLACE` modifier, yielding `CREATE OR REPLACE TABLE ... AS` statements:

```
CREATE OR REPLACE TABLE flights AS
    FROM 'https://duckdb.org/data/flights.csv';
```

Note that it is not possible to create tables using CTAS statements with constraints (primary keys, check constraints, etc.).

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)
);
```

Limitations

Foreign keys have the following limitations.

Foreign keys with cascading deletes (`FOREIGN KEY ... REFERENCES ... ON DELETE CASCADE`) are not supported.

Inserting into tables with self-referencing foreign keys is currently not supported and will result in the following error:

```
Constraint Error: Violates foreign key constraint because key "..." does not exist in the referenced table.
```

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 Statement

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;
```

The SQL query behind an existing view can be read using the `duckdb_views()` function like this:

```
SELECT sql FROM duckdb_views() WHERE view_name = 'v1';
```

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 Statement

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

The `CREATE TYPE` clause defines a new data type available to this DuckDB instance. These new types can then be inspected in the `duckdb_types` table.

Limitations

Extending types to support custom operators (such as the PostgreSQL && operator) is not possible via plain SQL. Instead, it requires adding additional C++ code. 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;
```

The `TRUNCATE` statement removes all rows from a table, acting as an alias for `DELETE FROM` without a `WHERE` clause:

```
TRUNCATE 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.

Limitations on Reclaiming Memory and Disk Space

Running `DELETE` does not mean space is reclaimed. In general, rows are only marked as deleted. DuckDB reclaims space upon performing a `CHECKPOINT`. `VACUUM` currently does not reclaim space.

DESCRIBE Statement

The `DESCRIBE` statement shows the schema of a table, view or query.

Usage

```
DESCRIBE tbl;
```

In order to summarize a query, prepend `DESCRIBE` to a query.

```
DESCRIBE SELECT * FROM tbl;
```

Alias

The `SHOW` statement is an alias for `DESCRIBE`.

See Also

For more examples, see the guide on `DESCRIBE`.

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;
```

Drop function `fn`:

```
DROP FUNCTION fn;
```

Drop index `idx`:

```
DROP INDEX idx;
```

Drop schema `sch`:

```
DROP SCHEMA sch;
```

Drop sequence `seq`:

```
DROP SEQUENCE seq;
```

Drop macro `mcr`:

```
DROP MACRO mcr;
```

Drop macro table `mt`:

```
DROP MACRO TABLE mt;
```

Drop type `typ`:

```
DROP TYPE typ;
```

Syntax

Dependencies of Dropped Objects

DuckDB performs limited dependency tracking for some object types. 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);
DROP SCHEMA myschema;
```

```
Dependency Error: Cannot drop entry `myschema` because there are entries that depend on it.
Use DROP...CASCADE to drop all dependents.
```

The CASCADE modifier drops both myschema and `myschema.t1`:

```
CREATE SCHEMA myschema;
CREATE TABLE myschema.t1 (i INTEGER);
DROP SCHEMA myschema CASCADE;
```

The following dependencies are tracked and thus will raise an error if the user tries to drop the depending object without the CASCADE modifier.

Depending object type	Dependandt object type
SCHEMA	FUNCTION
SCHEMA	INDEX
SCHEMA	MACRO TABLE
SCHEMA	MACRO
SCHEMA	SCHEMA
SCHEMA	SEQUENCE
SCHEMA	TABLE
SCHEMA	TYPE
SCHEMA	VIEW
TABLE	INDEX

Limitations

Dependencies on Views

Currently, dependencies are not tracked for views. For example, if a view is created that references a table and the table is dropped, then the view will be in an invalid state:

```
CREATE TABLE tbl (i INTEGER);
CREATE VIEW v AS
    SELECT i FROM tbl;
DROP TABLE tbl RESTRICT;
SELECT * FROM v;
```

```
Catalog Error: Table with name tbl does not exist!
```

Limitations on Reclaiming Disk Space

Running `DROP TABLE` should free the memory used by the table, but not always disk space. Even if disk space does not decrease, the free blocks will be marked as `free`. For example, if we have a 2 GB file and we drop a 1 GB table, the file might still be 2 GB, but it should have 1 GB of free blocks in it. To check this, use the following PRAGMA and check the number of `free_blocks` in the output:

```
PRAGMA database_size;
```

EXPORT/IMPORT DATABASE Statements

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 'target_directory' as CSV files:

```
EXPORT DATABASE 'target_directory';
```

Export to directory 'target_directory', using the given options for the CSV serialization:

```
EXPORT DATABASE 'target_directory' (FORMAT CSV, DELIMITER '|');
```

Export to directory 'target_directory', tables serialized as Parquet:

```
EXPORT DATABASE 'target_directory' (FORMAT PARQUET);
```

Export to directory 'target_directory', tables serialized as Parquet, compressed with ZSTD, with a row_group_size of 100,000:

```
EXPORT DATABASE 'target_directory' (
    FORMAT PARQUET,
    COMPRESSION ZSTD,
    ROW_GROUP_SIZE 100_000
);
```

Reload the database again:

```
IMPORT DATABASE 'source_directory';
```

Alternatively, use a PRAGMA:

```
PRAGMA import_database('source_directory');
```

For details regarding the writing of Parquet files, see the Parquet Files page in the Data Import section and the COPY Statement page.

EXPORT DATABASE

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.

Syntax

IMPORT DATABASE

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.

Syntax

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.

`INSERT INTO ... [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. That is, the values supplied by the `VALUES` clause or query are associated with the column list left-to-right. This is the default option, that can be explicitly specified using the `BY POSITION` option. For example:

```
CREATE TABLE tbl (a INTEGER, b INTEGER);
INSERT INTO tbl
    VALUES (5, 42);
```

Specifying "BY POSITION" is optional and is equivalent to the default behavior:

```
INSERT INTO tbl
    BY POSITION
    VALUES (5, 42);
```

To use a different order, column names can be provided as part of the target, for example:

```
CREATE TABLE tbl (a INTEGER, b INTEGER);
INSERT INTO tbl (b, a)
    VALUES (5, 42);
```

Adding `BY POSITION` results in the same behavior:

```
INSERT INTO tbl
    BY POSITION (b, a)
    VALUES (5, 42);
```

This will insert 5 into b and 42 into a.

`INSERT INTO ... BY NAME`

Using the `BY NAME` modifier, 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. This allows inserting even in cases when the order of the columns in the table differs from the order of the values in the `SELECT` statement or certain columns are missing.

For example:

```
CREATE TABLE tbl (a INTEGER, b INTEGER);
INSERT INTO tbl BY NAME (SELECT 42 AS b, 32 AS a);
INSERT INTO tbl BY NAME (SELECT 22 AS b);
SELECT * FROM tbl;
```

a	b
32	42
NULL	22

It's important to note that when using `INSERT INTO ... 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. Columns that are missing from the `SELECT` statement 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. An example for such a conflict is shown in the following example:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl
    VALUES (1, 84);
```

This raises as an error:

```
Constraint Error: Duplicate key "i: 1" violates primary key constraint.
```

The table will contain the row that was first inserted:

```
SELECT * FROM tbl;
```

i	j
1	42

These error messages can be avoided by explicitly handling conflicts. DuckDB supports two such clauses: `ON CONFLICT DO NOTHING` and `ON CONFLICT DO UPDATE SET ...`.

`DO NOTHING` Clause

The `DO NOTHING` clause causes the error(s) to be ignored, and the values are not inserted or updated. For example:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl
    VALUES (1, 84)
    ON CONFLICT DO NOTHING;
```

These statements finish successfully and leaves the table with the row `<i: 1, j: 42>`.

Shorthand for `DO NOTHING`

The `INSERT OR IGNORE INTO ...` statement is a shorter syntax alternative to `INSERT INTO ... ON CONFLICT DO NOTHING`. For example, the following statements are equivalent:

```
INSERT OR IGNORE INTO tbl
    VALUES (1, 84);
INSERT INTO tbl
    VALUES (1, 84) ON CONFLICT DO NOTHING;
```

DO UPDATE Clause (Upsert)

The DO UPDATE clause 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. The expressions can use the special virtual table EXCLUDED, which contains the conflicting values for the row. 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.

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.

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl VALUES (1, 42);
INSERT INTO tbl VALUES (1, 52), (1, 62) ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
```

Examples

An example using DO UPDATE is the following:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl
    VALUES (1, 84)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
SELECT * FROM tbl;
```

i	j
1	84

Rearranging columns and using BY NAME is also possible:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl (j, i)
    VALUES (168, 1)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
INSERT INTO tbl
    BY NAME (SELECT 1 AS i, 336 AS j)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
SELECT * FROM tbl;
```

i	j
1	336

Shorthand

The `INSERT OR REPLACE INTO ...` statement is a shorter syntax alternative to `INSERT INTO ... DO UPDATE SET c1 = EXCLUDED.c1, c2 = EXCLUDED.c2, ...`. That is, it updates every column of the **existing** row to the new values of the **to-be-inserted** row. For example, given the following input table:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
```

These statements are equivalent:

```
INSERT OR REPLACE INTO tbl
    VALUES (1, 84);
INSERT INTO tbl
    VALUES (1, 84)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
INSERT INTO tbl (j, i)
    VALUES (84, 1)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
INSERT INTO tbl BY NAME
    (SELECT 84 AS j, 1 AS i)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
```

Limitations

When the `ON CONFLICT ... DO UPDATE` clause is used and a conflict occurs, DuckDB internally assigns `NULL` values to the row's columns that are unaffected by the conflict, then re-assigns their values. If the affected columns use a `NOT NULL` constraint, this will trigger a `NOT NULL constraint failed` error. For example:

```
CREATE TABLE t1 (id INTEGER PRIMARY KEY, val1 DOUBLE, val2 DOUBLE NOT NULL);
CREATE TABLE t2 (id INTEGER PRIMARY KEY, val1 DOUBLE);
INSERT INTO t1
    VALUES (1, 2, 3);
INSERT INTO t2
    VALUES (1, 5);

INSERT INTO t1 BY NAME (SELECT id, val1 FROM t2)
    ON CONFLICT DO UPDATE
    SET val1 = EXCLUDED.val1;
```

This fails with the following error:

```
Constraint Error: NOT NULL constraint failed: t1.val2
```

Defining a Conflict Target

A conflict target may be provided as `ON CONFLICT (confict_target)`. This is a group of columns that an index or uniqueness/key constraint is defined on. If the conflict target is omitted, or `PRIMARY KEY` constraint(s) on the table are targeted.

Specifying a conflict target is optional unless using a `DO UPDATE` and there are multiple unique/primary key constraints on the table.

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER UNIQUE, k INTEGER);
INSERT INTO tbl
    VALUES (1, 20, 300);
SELECT * FROM tbl;
```

i	j	k
1	20	300

```
INSERT INTO tbl
    VALUES (1, 40, 700)
    ON CONFLICT (i) DO UPDATE SET k = 2 * EXCLUDED.k;
```

i	j	k
1	20	1400

```
INSERT INTO tbl
    VALUES (1, 20, 900)
    ON CONFLICT (j) DO UPDATE SET k = 5 * EXCLUDED.k;
```

i	j	k
1	20	4500

When a conflict target is provided, you can further filter this with a `WHERE` clause, that should be met by all conflicts.

```
INSERT INTO tbl
    VALUES (1, 40, 700)
    ON CONFLICT (i) DO UPDATE SET k = 2 * EXCLUDED.k WHERE k < 100;
```

Multiple Tuples Conflicting on the Same Key

Limitations

Currently, DuckDB's `ON CONFLICT DO UPDATE` feature is limited to enforce constraints between committed and newly inserted (transaction-local) data. In other words, having multiple tuples conflicting on the same key is not supported. If the newly inserted data has duplicate rows, an error message will be thrown, or unexpected behavior can occur. This also includes conflicts **only** within the newly inserted data.

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl
    VALUES (1, 84), (1, 168)
    ON CONFLICT DO UPDATE SET j = j + EXCLUDED.j;
```

This returns the following message.

```
Error: Invalid Input Error: ON CONFLICT DO UPDATE can not update the same row twice in the same command.
Ensure that no rows proposed for insertion within the same command have duplicate constrained values
```

To work around this, enforce uniqueness using `DISTINCT ON`. For example:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl
    SELECT DISTINCT ON(i) i, j FROM VALUES (1, 84), (1, 168) AS t (i, j)
    ON CONFLICT DO UPDATE SET j = j + EXCLUDED.j;
SELECT * FROM tbl;
```

i	j
1	126

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 `*`.

For example:

```
CREATE TABLE t1 (i INTEGER);
INSERT INTO t1
    SELECT 42
    RETURNING *;
```

i
42

A more complex example that includes an expression in the `RETURNING` clause:

```
CREATE TABLE t2 (i INTEGER, j INTEGER);
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

The next 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 page.

```
CREATE TABLE t3 (i INTEGER PRIMARY KEY, j INTEGER);
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.

> 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 <columns>
USING <values>
GROUP BY <rows>
ORDER BY <columns_with_order_directions>
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 INTEGER, Population INTEGER);
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
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
US	8579	8015	8783	8175	9510	8772
NL	1005	1005	1065	1065	1158	1158

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 PIVOT Statements

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>
    FOR
        <column_1> IN (<in_list>)
        <column_2> IN (<in_list>)
        ...
    GROUP BY <rows>
);
```

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.

Profiling Queries

DuckDB supports profiling queries via the `EXPLAIN` and `EXPLAIN ANALYZE` statements.

`EXPLAIN`

To see the query plan of a query without executing it, run:

```
EXPLAIN <query>;
```

The output of `EXPLAIN` contains the estimated cardinalities for each operator.

`EXPLAIN ANALYZE`

To profile a query, run:

```
EXPLAIN ANALYZE <query>;
```

The `EXPLAIN ANALYZE` statement runs the query, and shows the actual cardinalities for each operator, as well as the cumulative wall-clock time spent in each operator.

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_expression>
WHERE <condition>
GROUP BY <groups>
HAVING <group_filter>
    WINDOW <window_expression>
    QUALIFY <qualify_filter>
ORDER BY <order_expression>
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.

> 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 and HAVING Clauses

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 and OFFSET Clauses

ORDER BY, LIMIT and OFFSET are output modifiers. Logically they are applied at the very end of the query. The ORDER BY clause sorts the rows on the sorting criteria in either ascending or descending order. The LIMIT clause restricts the amount of rows fetched, while the OFFSET clause indicates at which position to start reading the values.

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 INTEGER, 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*.

> **Tip.** The `rowid` values are stable within a transaction.

> 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 Statements

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 = 1;
```

Or use the TO keyword:

```
SET threads TO 1;
```

Change configuration option to default value:

```
RESET threads;
```

Retrieve configuration value:

```
SELECT current_setting('threads');
```

Set the default catalog search path globally:

```
SET GLOBAL search_path = 'db1,db2'
```

Set the default collation for the session:

```
SET SESSION default_collation = 'nocase';
```

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

Configuration options can have different scopes:

- GLOBAL: Configuration value is used (or reset) across the entire DuckDB instance.
- SESSION: Configuration value is used (or reset) only for the current session attached to a DuckDB instance.
- LOCAL: Not yet implemented.

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.

SUMMARIZE Statement

The SUMMARIZE statement returns summary statistics for a table, view or a query.

Usage

```
SUMMARIZE tbl;
```

In order to summarize a query, prepend SUMMARIZE to a query.

```
SUMMARIZE SELECT * FROM tbl;
```

See Also

For more examples, see the guide on SUMMARIZE.

Transaction Management

DuckDB supports ACID database transactions. Transactions provide isolation, i.e., changes made by a transaction are not visible from concurrent transactions until it is committed. A transaction can also be aborted, which discards any changes it made so far.

Statements

DuckDB provides the following statements for transaction management.

Starting a Transaction

To start a transaction, run:

```
BEGIN TRANSACTION;
```

Committing a Transaction

You can commit a transaction to make it visible to other transactions and to write it to persistent storage (if using DuckDB in persistent mode). To commit a transaction, run:

```
COMMIT;
```

If you are not in an active transaction, the `COMMIT` statement will fail.

Rolling Back a Transaction

You can abort a transaction. This operation, also known as rolling back, will discard any changes the transaction made to the database. To abort a transaction, run:

```
ROLLBACK;
```

You can also use the abort command, which has an identical behavior:

```
ABORT;
```

If you are not in an active transaction, the `ROLLBACK` and `ABORT` statements will fail.

Example

We illustrate the use of transactions through a simple example.

```
CREATE TABLE person (name VARCHAR, age BIGINT);

BEGIN TRANSACTION;
INSERT INTO person VALUES ('Ada', 52);
COMMIT;

BEGIN TRANSACTION;
DELETE FROM person WHERE name = 'Ada';
INSERT INTO person VALUES ('Bruce', 39);
ROLLBACK;

SELECT * FROM person;
```

The first transaction (inserting "Ada") was committed but the second (deleting "Ada" and inserting "Bruce") was aborted. Therefore, the resulting table will only contain `<'Ada', 52>`.

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.

> 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 INTEGER, dept TEXT, Jan INTEGER, Feb INTEGER, Mar INTEGER, Apr INTEGER, May INTEGER, Jun
INTEGER);
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

empid	dept	month	sales
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` 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
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

empid	dept	month	sales
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 `UNPIVOT`s 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;
```

Expressions within `UNPIVOT` Statements

DuckDB allows expressions within the `UNPIVOT` statements, provided that they only involve a single column. These can be used to perform computations as well as explicit casts. For example:

```
UNPIVOT
    (SELECT 42 as col1, 'woot' as col2)
    ON
        (col1 * 2)::VARCHAR,
        col2;
```

name	value
col1	84
col2	woot

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

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

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
1	electronics	Feb	2

empid	dept	month	sales
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 the `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
    );
```

Update Using Joins

To select the rows to update, UPDATE statements can use the FROM clause and express joins via the WHERE clause. For example:

```
CREATE TABLE city (name VARCHAR, revenue BIGINT, country_code VARCHAR);
CREATE TABLE country (code VARCHAR, name VARCHAR);
INSERT INTO city VALUES ('Paris', 700, 'FR'), ('Lyon', 200, 'FR'), ('Brussels', 400, 'BE');
INSERT INTO country VALUES ('FR', 'France'), ('BE', 'Belgium');
```

To increase the revenue of all cities in France, join the `city` and the `country` tables, and filter on the latter:

```
UPDATE city
SET revenue = revenue + 100
FROM country
WHERE city.country_code = country.code
  AND country.name = 'France';

SELECT *
FROM city;
```

name	revenue	country_code
Paris	800	FR
Lyon	300	FR
Brussels	400	BE

Upsert (Insert or Update)

See the Insert documentation for details.

USE Statement

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 Statement

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 and column:

```
VACUUM ANALYZE memory.main.my_table(my_column);
```

Not supported:

```
VACUUM FULL; -- error
```

Reclaiming Space

To reclaim space after deleting rows, use the CHECKPOINT statement.

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 the 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 regular expression from the table:

```
SELECT COLUMNS('number\d+') FROM addresses;
```

Compute a function on all given columns of a table:

```
SELECT min(COLUMNS(*)) FROM addresses;
```

To select columns with spaces or special characters, use double quotes (`"`):

```
SELECT "Some Column Name" FROM tbl;
```

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.

> 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 regular expression 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.

> 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.

> 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` Function

Unnest an array by one level:

```
SELECT unnest([1, 2, 3]);
```

Unnest a struct by one level:

```
SELECT unnest({'a': 42, 'b': 84});
```

The `unnest` function is a special function that can be used together with arrays, lists, or structs. The unnest function strips one level of nesting from the type. For example, `INTEGER[]` is transformed into `INTEGER`. `STRUCT(a INTEGER, b INTEGER)` is transformed into a `INTEGER, b INTEGER`. 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 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` through an alias `tn`:

```
SELECT tn.* FROM table_name tn;
```

Select all columns from the table `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 (Cartesian Product)

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;
```

This is equivalent to omitting the `JOIN` clause:

```
SELECT a.*, b.* FROM a, 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 do 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
0	1
2	3
1	2

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
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) ORDER BY ALL;
```

i	j	k
0	1	1
1	2	3
2	3	5

> 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 such as 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:*

```
CREATE TABLE t1 (x INTEGER);
CREATE TABLE t2 (s VARCHAR);

INSERT INTO t1 VALUES (1), (2), (3);
INSERT INTO t2 VALUES ('a'), ('b');

SELECT *
FROM t1
POSITIONAL JOIN t2;
```

x	s
1	a
2	b
3	NULL

Positional joins are always FULL OUTER joins, i.e., missing values (the last values in the shorter column) are set to NULL.

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");
```

FROM-First Syntax

DuckDB's SQL supports the FROM-first syntax, i.e., it allows putting the `FROM` clause before the `SELECT` clause or completely omitting the `SELECT` clause. We use the following example to demonstrate it:

```
CREATE TABLE tbl AS
    SELECT * FROM (VALUES ('a'), ('b')) t1(s), range(1, 3) t2(i);
```

FROM-First Syntax with a SELECT Clause

The following statement demonstrates the use of the FROM-first syntax:

```
FROM tbl
SELECT i, s;
```

This is equivalent to:

```
SELECT i, s
FROM tbl;
```

i	s
1	a
2	a
1	b
2	b

FROM-First Syntax without a SELECT Clause

The following statement demonstrates the use of the optional SELECT clause:

```
FROM tbl;
```

This is equivalent to:

```
SELECT *
FROM tbl;
```

s	i
a	1
a	2
b	1
b	2

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 where the `id` is 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:

```
-- the syntax () denotes 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()
Math	NULL	1
NULL	NULL	7
CS	PhD	1
CS	Bachelor	2
Math	Masters	1
CS	NULL	2
Math	NULL	2
CS	NULL	5
NULL	NULL	3
NULL	Masters	1
NULL	Bachelor	2
NULL	PhD	1

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:

```sql
SELECT course, type, count(*)
FROM students
GROUP BY course, type
UNION ALL
```

Group by type:

```sql
SELECT NULL AS course, type, count(*)
FROM students
GROUP BY type
UNION ALL
```

Group by course:

```sql
SELECT course, NULL AS type, count(*)
FROM students
GROUP BY course
UNION ALL
```

Group by nothing:

```sql
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.

Identifying Grouping Sets with `GROUPING_ID()`

The super-aggregate rows generated by `GROUPING SETS`, ROLLUP and CUBE can often be identified by NULL-values returned for the respective column in the grouping. But if the columns used in the grouping can themselves contain actual NULL-values, then it can be challenging to distinguish whether the value in the resultset is a "real" NULL-value coming out of the data itself, or a NULL-value generated by the grouping construct. The `GROUPING_ID()` or `GROUPING()` function is designed to identify which groups generated the super-aggregate rows in the result.

`GROUPING_ID()` is an aggregate function that takes the column expressions that make up the grouping(s). It returns a BIGINT value. The return value is `0` for the rows that are not super-aggregate rows. But for the super-aggregate rows, it returns an integer value that identifies the combination of expressions that make up the group for which the super-aggregate is generated. At this point, an example might help. Consider the following query:

```sql
WITH days AS (
    SELECT
        year("generate_series")    AS y,
        quarter("generate_series") AS q,
        month("generate_series")   AS m
    FROM generate_series(DATE '2023-01-01', DATE '2023-12-31', INTERVAL 1 DAY)
)
SELECT y, q, m, GROUPING_ID(y, q, m) AS "grouping_id()"
FROM days
GROUP BY GROUPING SETS (
    (y, q, m),
    (y, q),
    (y),
    ()
)
ORDER BY y, q, m;
```

These are the results:

y	q	m	grouping_id()
2023	1	1	0
2023	1	2	0
2023	1	3	0
2023	1	NULL	1
2023	2	4	0
2023	2	5	0
2023	2	6	0
2023	2	NULL	1
2023	3	7	0
2023	3	8	0
2023	3	9	0
2023	3	NULL	1
2023	4	10	0
2023	4	11	0
2023	4	12	0
2023	4	NULL	1
2023	NULL	NULL	3
NULL	NULL	NULL	7

In this example, the lowest level of grouping is at the month level, defined by the grouping set `(y, q, m)`. Result rows corresponding to that level are simply aggregate rows and the `GROUPING_ID(y, q, m)` function returns `0` for those. The grouping set `(y, q)` results in super-aggregate rows over the month level, leaving a `NULL`-value for the `m` column, and for which `GROUPING_ID(y, q, m)` returns `1`. The grouping set `(y)` results in super-aggregate rows over the quarter level, leaving `NULL`-values for the `m` and `q` column, for which `GROUPING_ID(y, q, m)` returns 3. Finally, the `()` grouping set results in one super-aggregate row for the entire resultset, leaving `NULL`-values for `y`, `q` and `m` and for which `GROUPING_ID(y, q, m)` returns 7.

To understand the relationship between the return value and the grouping set, you can think of `GROUPING_ID(y, q, m)` writing to a bitfield, where the first bit corresponds to the last expression passed to `GROUPING_ID()`, the second bit to the one-but-last expression passed to `GROUPING_ID()`, and so on. This may become clearer by casting `GROUPING_ID()` to `BIT`:

```
WITH days AS (
    SELECT
        year("generate_series")    AS y,
        quarter("generate_series") AS q,
        month("generate_series")   AS m
    FROM generate_series(DATE '2023-01-01', DATE '2023-12-31', INTERVAL 1 DAY)
)
SELECT
    y, q, m,
    GROUPING_ID(y, q, m) AS "grouping_id(y, q, m)",
    right(GROUPING_ID(y, q, m)::BIT::VARCHAR, 3) AS "y_q_m_bits"
FROM days
GROUP BY GROUPING SETS (
    (y, q, m),
    (y, q),
    (y),
    ()
```

```
)
ORDER BY y, q, m;
```

Which returns these results:

y	q	m	grouping_id(y, q, m)	y_q_m_bits
2023	1	1	0	000
2023	1	2	0	000
2023	1	3	0	000
2023	1	NULL	1	001
2023	2	4	0	000
2023	2	5	0	000
2023	2	6	0	000
2023	2	NULL	1	001
2023	3	7	0	000
2023	3	8	0	000
2023	3	9	0	000
2023	3	NULL	1	001
2023	4	10	0	000
2023	4	11	0	000
2023	4	12	0	000
2023	4	NULL	1	001
2023	NULL	NULL	3	011
NULL	NULL	NULL	7	111

Note that the number of expressions passed to `GROUPING_ID()`, or the order in which they are passed is independent from the actual group definitions appearing in the `GROUPING SETS`-clause (or the groups implied by `ROLLUP` and `CUBE`). As long as the expressions passed to `GROUPING_ID()` are expressions that appear some where in the `GROUPING SETS`-clause, `GROUPING_ID()` will set a bit corresponding to the position of the expression whenever that expression is rolled up to a super-aggregate.

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 with the following configuration options.

> Using `ASC NULLS LAST` as 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`:

```
SET default_null_order = 'NULLS FIRST';
```

Change the default sorting order to either `DESC` or `ASC`:

```
SET default_order = 'DESC';
```

Collations

Text is sorted using the binary comparison collation by default, which means values are sorted on their binary UTF-8 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 and OFFSET Clauses

`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:

```
SELECT *
FROM (SELECT * FROM addresses)
USING SAMPLE 10 ROWS;
```

Syntax

Unnesting

Examples

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);
```

Limit depth of recursive unnest using `max_depth`:

```
SELECT unnest([[[1, 2], [3, 4]], [[5, 6], [7, 8, 9], []], [[10, 11]]], max_depth := 2);
```

The `unnest` special 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. Invoking `unnest` with the `recursive` parameter will unnest lists and structs of multiple levels. The depth of unnesting can be limited using the `max_depth` parameter (which assumes `recursive` unnesting by default).

Unnesting Lists

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);
```

Using `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.

Seeting the Maximum Depth of Unnesting

The `max_depth` parameter allows limiting the maximum depth of recursive unnesting (which is assumed by default and does not have to be specified separately). For example, unnestig to `max_depth` of 2 yields the following:

```
SELECT unnest([[[1, 2], [3, 4]], [[5, 6], [7, 8, 9], []], [[10, 11]]], max_depth := 2) AS x;
```

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

x
[]
[10, 11]

Meanwhile, unnesting to `max_depth` of 3 results in:

```
SELECT unnest([[[1, 2], [3, 4]], [[5, 6], [7, 8, 9], []], [[10, 11]]], max_depth := 3) AS x;
```

x
1
2
3
4
5
6
7
8
9
10
11

Keeping Track of List Entry Positions

To keep track of each entry's position within the original list, `unnest` may be combined with `generate_subscripts`:

```
SELECT unnest(l) as x, generate_subscripts(l, 1) AS index
FROM (VALUES ([1, 2, 3]), ([4, 5])) tbl(l);
```

x	index
1	1
2	2
3	3
4	1
5	2

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.

Example: Fibonacci Sequence

`WITH RECURSIVE` can be used to make recursive calculations. For example, here is how `WITH RECURSIVE` could be used to calculate the first ten Fibonacci numbers:

```
WITH RECURSIVE FibonacciNumbers (RecursionDepth, FibonacciNumber, NextNumber) AS (
    -- Base case
    SELECT
        0  AS RecursionDepth,
        0  AS FibonacciNumber,
        1  AS NextNumber
    UNION ALL
    -- Recursive step
    SELECT
        fib.RecursionDepth + 1 AS RecursionDepth,
        fib.NextNumber AS FibonacciNumber,
        fib.FibonacciNumber + fib.NextNumber AS NextNumber
    FROM
        FibonacciNumbers fib
    WHERE
        fib.RecursionDepth + 1 < 10
)
```

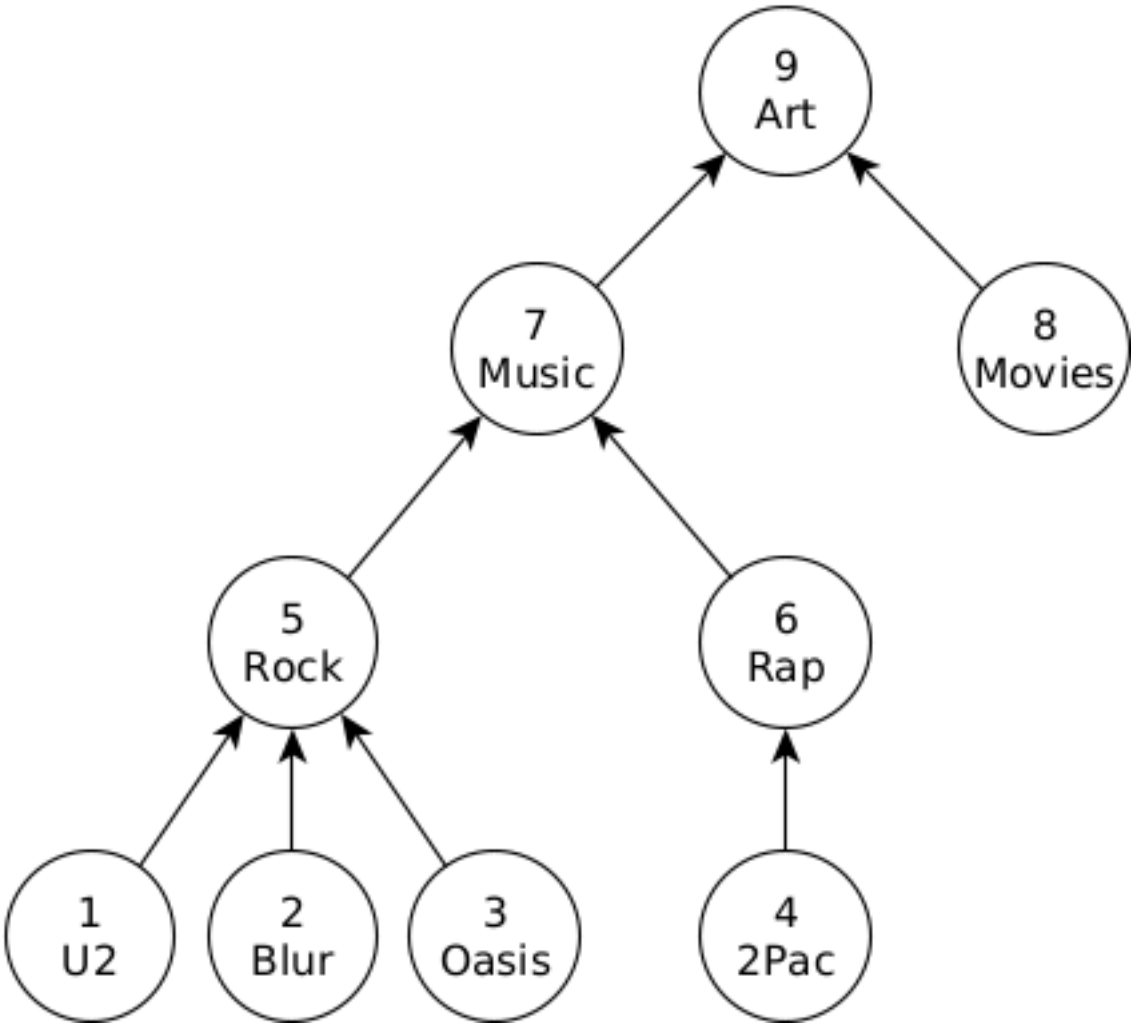
Query the CTE:

```
SELECT
    fn.RecursionDepth AS FibonacciNumberIndex,
    fn.FibonacciNumber
FROM
    FibonacciNumbers fn;
```

FibonacciNumberIndex	FibonacciNumber
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34

Example: Tree Traversal

`WITH RECURSIVE` can be used to traverse trees. For example, take a hierarchy of tags:




```
CREATE TABLE tag (id INTEGER, name VARCHAR, subclassof INTEGER);
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`).

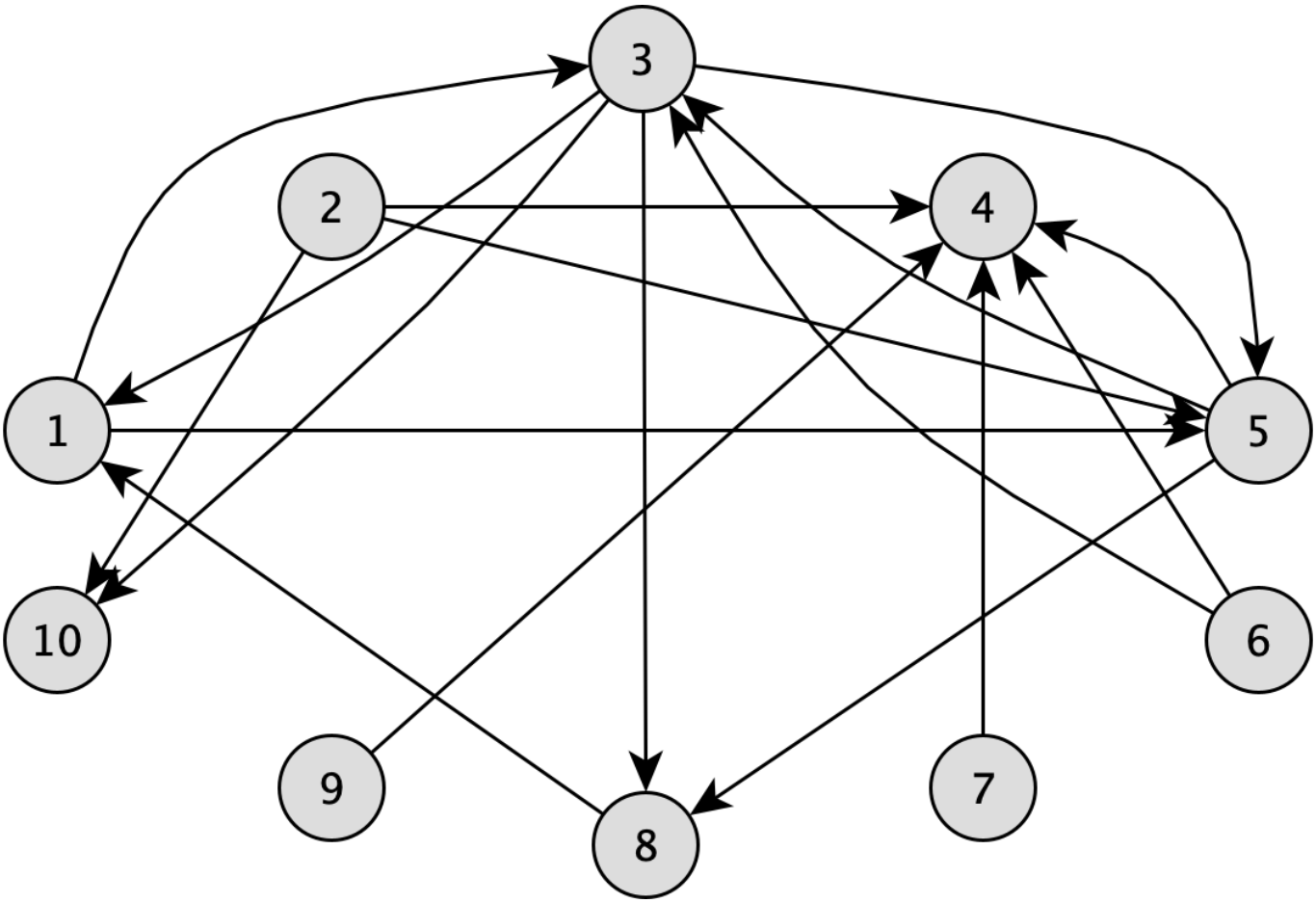
```
WITH RECURSIVE tag_hierarchy(id, source, path) AS (
    SELECT id, name, [name] AS path
    FROM tag
    WHERE subclassof IS NULL
UNION ALL
    SELECT tag.id, tag.name, list_prepend(tag.name, tag_hierarchy.path)
    FROM tag, tag_hierarchy
    WHERE tag.subclassof = tag_hierarchy.id
)
SELECT path
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 INTEGER, node2id INTEGER);
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:

```sql
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:

```sql
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 a `QUALIFY` clause):

```sql
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

Return the following:

- The total number of rows.
- The number of rows where `i <= 5`
- The number of rows where `i` is odd

```
SELECT
    count(*) AS total_rows,
    count(*) FILTER (i <= 5) AS lte_five,
    count(*) FILTER (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:

```
SELECT
    sum(i) FILTER (i <= 5) AS lte_five_sum,
    median(i) FILTER (i % 2 = 1) AS odds_median,
    median(i) FILTER (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, 100_000_000) tbl(i)
    ) tbl;
```

”Pivot” the data out by year (move each year out to a separate column):

```
SELECT
    count(i) FILTER (year = 2022) AS "2022",
    count(i) FILTER (year = 2023) AS "2023",
    count(i) FILTER (year = 2024) AS "2024",
    count(i) FILTER (year = 2025) AS "2025",
    count(i) FILTER (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 (year = 2022) AS "2022",
    first(i) FILTER (year = 2023) AS "2023",
    first(i) FILTER (year = 2024) AS "2024",
    first(i) FILTER (year = 2025) AS "2025",
    first(i) FILTER (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 [ALL] and EXCEPT [ALL] clauses. The vanilla variants use set semantics, i.e., they eliminate duplicates, while the variants with ALL use bag semantics.

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 [ALL] 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.

UNION

The UNION clause can be used to combine rows from multiple queries. The queries are required to return the same number of columns. Implicit casting to one of the returned types is performed to combine columns of different types where necessary. If this is not possible, the UNION clause throws an error.

Vanilla UNION (Set Semantics)

The vanilla UNION clause follows set semantics, therefore it performs duplicate elimination, i.e., only unique rows will be included in the result.

```
SELECT * FROM range(2) t1(x)
UNION
SELECT * FROM range(3) t2(x);
```

x
2
1

x
0

`UNION ALL` (Bag Semantics)

`UNION ALL` returns all rows of both queries following bag semantics, i.e., *without* duplicate elimination.

```
SELECT * FROM range(2) t1(x)
UNION ALL
SELECT * FROM range(3) t2(x);
```

x
0
1
0
1
2

`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.

Take the following tables for example:

```
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');

SELECT * FROM capitals
UNION BY NAME
SELECT * FROM weather;
```

city	country	degrees	date
Seattle	NULL	8	2022-10-12
Amsterdam	NL	NULL	NULL
Berlin	Germany	NULL	NULL
Amsterdam	NULL	10	2022-10-14

`UNION BY NAME` follows set semantics (therefore it performs duplicate elimination), whereas `UNION ALL BY NAME` follows bag semantics.

INTERSECT

The INTERSECT clause can be used to select all rows that occur in the result of **both** queries.

Vanilla INTERSECT (Set Semantics)

Vanilla INTERSECT performs duplicate elimination, so only unique rows are returned.

```
SELECT * FROM range(2) t1(x)
INTERSECT
SELECT * FROM range(6) t2(x);
```

x
0
1

INTERSECT ALL (Bag Semantics)

INTERSECT ALL follows bag semantics, so duplicates are returned.

```
SELECT unnest([5, 5, 6, 6, 6, 6, 7, 8]) AS x
INTERSECT ALL
SELECT unnest([5, 6, 6, 7, 7, 9]);
```

x
5
6
6
7

EXCEPT

The EXCEPT clause can be used to select all rows that **only** occur in the left query.

Vanilla EXCEPT (Set Semantics)

Vanilla EXCEPT follows set semantics, therefore, it performs duplicate elimination, so only unique rows are returned.

```
SELECT * FROM range(5) t1(x)
EXCEPT
SELECT * FROM range(2) t2(x);
```

x
2
3
4

x

EXCEPT ALL (Bag Semantics)

EXCEPT ALL uses bag semantics:

```
SELECT unnest([5, 5, 6, 6, 6, 6, 7, 8]) AS x
EXCEPT ALL
SELECT unnest([5, 6, 6, 7, 7, 9]);
```

x
5
8
6
6

Syntax

Prepared Statements

DuckDB supports prepared statements where parameters are substituted when the query is executed. This can improve readability and is useful for preventing SQL injections.

Syntax

There are three syntaxes for denoting parameters in prepared statements: auto-incremented (`?`), positional (`$1`), and named (`$param`). Note that not all clients support all of these syntaxes, e.g., the JDBC client only supports auto-incremented parameters in prepared statements.

Example Data Set

In the following, we introduce the three different syntaxes and illustrate them with examples using the following table.

```
CREATE TABLE person (name VARCHAR, age BIGINT);
INSERT INTO person VALUES ('Alice', 37), ('Ana', 35), ('Bob', 41), ('Bea', 25);
```

In our example query, we'll look for people whose name starts with a "B" and are at least 40 years old. This will return a single row `<'Bob', 41>`.

Auto-Incremented Parameters: ?

DuckDB support using prepared statements with auto-incremented indexing, i.e., the position of the parameters in the query corresponds to their position in the execution statement. For example:

```
PREPARE query_person AS
    SELECT *
    FROM person
    WHERE starts_with(name, ?)
      AND age >= ?;
```

Using the CLI client, the statement is executed as follows.

```
EXECUTE query_person('B', 40);
```

Positional Parameters: `$1`

Prepared statements can use positional parameters, where parameters are denoted with an integer (`$1`, `$2`). For example:

```
PREPARE query_person AS
    SELECT *
    FROM person
    WHERE starts_with(name, $2)
      AND age >= $1;
```

Using the CLI client, the statement is executed as follows. Note that the first parameter corresponds to `$1`, the second to `$2`, and so on.

```
EXECUTE query_person(40, 'B');
```

Named Parameters: `$parameter`

DuckDB also supports names parameters where parameters are denoted with `$parameter_name`. For example:

```
PREPARE query_person AS
    SELECT *
    FROM person
    WHERE starts_with(name, $name_start_letter)
      AND age >= $minimum_age;
```

Using the CLI client, the statement is executed as follows.

```
EXECUTE query_person(name_start_letter := 'B', minimum_age := 40);
```

Dropping Prepared Statements: DEALLOCATE

To drop a prepared statement, use the `DEALLOCATE` statement:

```
DEALLOCATE query_person;
```

Alternatively, use:

```
DEALLOCATE PREPARE query_person;
```

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 1s and 0s
`BLOB`	`BYTEA`, `BINARY`, `VARBINARY`	variable-length binary data
`BOOLEAN`	`BOOL`, `LOGICAL`	logical boolean (true/false)
`DATE`		calendar date (year, month day)
`DECIMAL(prec, scale)`	`NUMERIC(prec, scale)`	fixed-precision number with the given width (precision) and scale, defaults to `prec = 18` and `scale = 3`
`DOUBLE`	`FLOAT8`,	double precision floating-point number (8 bytes)
`HUGEINT`		signed sixteen-byte integer
`INTEGER`	`INT4`, `INT`, `SIGNED`	signed four-byte integer
`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 WITH TIME ZONE`	`TIMESTAMPTZ`	combination of time and date that uses the current time zone
`TIMESTAMP`	`DATETIME`	combination of time and date
`TINYINT`	`INT1`	signed one-byte integer
`UBIGINT`		unsigned eight-byte integer
`UHUGEINT`		unsigned sixteen-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

Implicit and explicit typecasting is possible between numerous types, see the Typecasting page for details.

Nested / Composite Types

DuckDB supports five nested data types: `ARRAY`, `LIST`, `MAP`, `STRUCT`, 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
`ARRAY`	An ordered, fixed-length sequence of data values of the same type.	Each row must have the same data type within each instance of the `ARRAY` and the same number of elements.	`[1, 2, 3]`	`INTEGER[3]`
`LIST`	An ordered sequence of data values of the same type.	Each row must have the same data type within each instance of the `LIST`, but can have any number of elements.	`[1, 2, 3]`	`INTEGER[]`
`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(INTEGER, VARCHAR)`
`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 INTEGER, j 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 INTEGER, text VARCHAR)`

Updating Values of Nested Types

When performing *updates* on values of nested types, DuckDB performs a *delete* operation followed by an *insert* operation. When used in a table with ART indexes (either via explicit indexes or primary keys/unique constraints), this can lead to unexpected constraint violations. For example:

```
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO students VALUES (1, 'Student 1');

UPDATE tbl
  SET j = [2]
  WHERE i = 1;
```

```
Constraint Error: Duplicate key "i: 1" violates primary key constraint.
If this is an unexpected constraint violation please double check with the known index limitations section
in our documentation (https://duckdb.org/docs/sql/indexes).
```

Nesting

ARRAY, LIST, MAP, STRUCT, and UNION types 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)];
```

Performance Implications

The choice of data types can have a strong effect on performance. Please consult the Performance Guide for details.

Array Type

An ARRAY column stores fixed-sized arrays. All fields in the column must have the same length and the same underlying type. Arrays are typically used to store arrays of numbers, but can contain any uniform data type, including ARRAY, LIST and STRUCT types.

Arrays can be used to store vectors such as word embeddings or image embeddings.

To store variable-length lists, use the LIST type. See the data types overview for a comparison between nested data types.

> The ARRAY type in PostgreSQL allows variable-length fields. DuckDB's ARRAY type is fixed-length.

Creating Arrays

Arrays can be created using the `array_value(expr, ...)` function.

Construct with the `array_value` function:

```
SELECT array_value(1, 2, 3);
```

You can always implicitly cast an array to a list (and use list functions, like `list_extract`, `[i]`):

```
SELECT array_value(1, 2, 3)[2];
```

You can cast from a list to an array, but the dimensions have to match up!:

```
SELECT [3, 2, 1]::INTEGER[3];
```

Arrays can be nested:

```
SELECT array_value(array_value(1, 2), array_value(3, 4), array_value(5, 6));
```

Arrays can store structs:

```
SELECT array_value({'a': 1, 'b': 2}, {'a': 3, 'b': 4});
```

Defining an Array Field

Arrays can be created using the `<TYPE_NAME>[<LENGTH>]` syntax. For example, to create an array field for 3 integers, run:

```
CREATE TABLE array_table (id INTEGER, arr INTEGER[3]);
INSERT INTO array_table VALUES (10, [1, 2, 3]), (20, [4, 5, 6]);
```

Retrieving Values from Arrays

Retrieving one or more values from an array can be accomplished using brackets and slicing notation, or through list functions like `list_extract` and `array_extract`. Using the example in Defining an Array Field.

The following queries for extracting the second element of an array are equivalent:

```
SELECT id, arr[1] AS element FROM array_table;
SELECT id, list_extract(arr, 1) AS element FROM array_table;
SELECT id, array_extract(arr, 1) AS element FROM array_table;
```

id	element
10	1
20	4

Using the slicing notation returns a `LIST`:

```
SELECT id, arr[1:2] AS elements FROM array_table;
```

id	elements
10	[1, 2]
20	[4, 5]

Functions

All `LIST` functions work with the ARRAY type. Additionally, several ARRAY-native functions are also supported. See the ARRAY functions.

Examples

Create sample data:

```
CREATE TABLE x (i INTEGER, v FLOAT[3]);
CREATE TABLE y (i INTEGER, v FLOAT[3]);
INSERT INTO x VALUES (1, array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT));
INSERT INTO y VALUES (1, array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT));
```

Compute cross product:

```
SELECT array_cross_product(x.v, y.v)
FROM x, y
WHERE x.i = y.i;
```

Compute cosine similarity:

```
SELECT array_cosine_similarity(x.v, y.v)
FROM x, y
WHERE x.i = y.i;
```

Ordering

The ordering of ARRAY instances is defined using a lexicographical order. `NULL` values compare greater than all other values and are considered equal to each other.

See Also

For more functions, see List Functions.

Bitstring Type

Name	Aliases	Description
BIT	BITSTRING	variable-length strings of 1s and 0s

Bitstrings are strings of 1s and 0s. 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 values 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` is `false` and `NULL > 10` is NULL):

i
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`).

```
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 AS negative,
    'epoch'::DATE AS epoch,
    'infinity'::DATE AS positive;
```

negative	epoch	positive
-infinity	1970-01-01	infinity

Functions

See Date Functions.

Enum Data Type

Name	Description
enum	Dictionary Encoding representing all possible string values of a column.

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');
```

The following query will fail since the mood type does not have `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';
```

name	current_mood
Pagliacci	sad

If you are importing data from a file, you can create an enum for a VARCHAR column before importing. Given this, the following subquery selects 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';
```

Enums 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.*') AS contains_a
FROM person;
```

contains_a
true
NULL
true
false

Create a new mood and table:

```
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 the `current_mood` and `future_mood` columns are constructed on different enum types, DuckDB will cast both enums to strings and perform a string comparison:

```
SELECT *
FROM person_2
WHERE current_mood = future_mood;
```

When comparing the `past_mood` column (string), 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.

> **Warning.** This behavior of the enum removal feature is subject to change. In future releases, it is expected that any dependent columns must be removed before dropping the enum, or the enum must be dropped with the additional CASCADE parameter.

Comparison of Enums

Enum values are compared according to their order in the enum's definition. For example:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

```
SELECT 'sad'::mood < 'ok'::mood AS comp;
```

comp
true

```
SELECT unnest(['ok'::mood, 'happy'::mood, 'sad'::mood]) AS m
ORDER BY m;
```

m
sad
ok
happy

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.

Examples

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';
```

Intervals greater than 24 hours/12 months/etc. are supported:

```
SELECT '540:58:47.210'::INTERVAL;
SELECT INTERVAL '16 MONTHS';
```

> **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))`
>
> ```
> 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
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
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 Type

A `LIST` column encodes lists of values. Fields in the 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.

> For storing fixed-length lists, DuckDB uses the ARRAY type.

Creating Lists

Lists can be created using the `list_value(expr, ...)` function or the equivalent bracket notation `[expr, ...]`. The expressions can be constants or arbitrary expressions. To create a list from a table column, use the `list` aggregate function.

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 INTEGER[], 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`.

> 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.

Updating Lists

Updates on lists are internally represented as an insert and a delete operation. Therefore, updating list values may lead to a duplicate key error on primary/unique keys. See the following example:

```
CREATE TABLE tbl (id INTEGER PRIMARY KEY, lst INTEGER[], comment VARCHAR);
INSERT INTO tbl VALUES (1, [12, 34], 'asd');
UPDATE tbl SET lst = [56, 78] WHERE id = 1;
```

```
Constraint Error: Duplicate key "id: 1" violates primary key constraint.
If this is an unexpected constraint violation please double check with the known index limitations section
in our documentation (https://duckdb.org/docs/sql/indexes).
```

Functions

See Nested Functions.

Literal Types

DuckDB has special literal types for representing NULL, integer and string literals in queries. These have their own binding and conversion rules.

> Prior to version 0.10.0, integer and string literals behaved identically to the INTEGER and VARCHAR types.

Null Literals

The NULL literal is denoted with the keyword NULL. The NULL literal can be implicitly converted to any other type.

Integer Literals

Integer literals are denoted as a sequence of one or more digits. At runtime, these result in values of the INTEGER_LITERAL type. INTEGER_LITERAL types can be implicitly converted to any integer type in which the value fits. For example, the integer literal 42 can be implicitly converted to a TINYINT, but the integer literal 1000 cannot be.

Other Numeric Literals

Non-integer numeric literals can be denoted with decimal notation, using the period character (.) to separate the integer part and the decimal part of the number. Either the integer part or the decimal part may be omitted:

```
SELECT 1.5;          -- 1.5
SELECT .50;          -- 0.5
SELECT 2.;           -- 2.0
```

Non-integer numeric literals can also be denoted using *E notation*. In E notation, an integer or decimal literal is followed by and exponential part, which is denoted by e or E, followed by a literal integer indicating the exponent. The exponential part indicates that the preceding value should be multiplied by 10 raised to the power of the exponent:

```
SELECT 1e2;           -- 100
SELECT 6.02214e23;    -- Avogadro's constant
SELECT 1e-10;         -- 1 ångström
```

Underscores in Numeric Literals

DuckDB's SQL dialect allows using the underscore character _ in numeric literals as an optional separator. The rules for using underscores are as follows:

- Underscores are allowed in integer, decimal, hexadecimal and binary notation.
- Underscores can not be the first or last character in a literal.
- Underscores have to have an integer/numeric part on either side of them, i.e., there can not be multiple underscores in a row and not immediately before/after a decimal or exponent.

Examples:

```
SELECT 100_000_000;          -- 100000000
SELECT '0xFF_FF'::INTEGER;   -- 65535
SELECT 1_2.1_2E0_1;          -- 121.2
SELECT '0b0_1_0_1'::INTEGER; -- 5
```

String Literals

String literals are delimited using single quotes (', apostrophe) and result in `STRING_LITERAL` values. Note that double quotes (") cannot be used as string delimiter character: instead, double quotes are used to delimit quoted identifiers.

Implicit String Literal Concatenation

Consecutive single-quoted string literals sepearated only by whitespace that contains at least one newline are implicitly concatenated:

```
SELECT 'Hello'
    ' '
    'World' AS greeting;
```

is equivalent to:

```
SELECT 'Hello'
    || ' '
    || 'World' AS greeting;
```

They both return the following result:

greeting
Hello World

Note that implicit concatenation only works if there is at least one newline between the literals. Using adjacent string literals separated by whitspace without a newline results in a syntax error:

```
SELECT 'Hello' ' ' 'World' AS greeting;
```

```
Parser Error: syntax error at or near "' '"
LINE 1: SELECT 'Hello' ' ' 'World' as greeting;
```

Also note that implicit concatenation only works with single-quoted string literals, and does not work with other kinds of string values.

Implicit string conversion

`STRING_LITERAL` instances can be implicitly converted to *any* other type.

For example, we can compare string literals with dates:

```
SELECT d > '1992-01-01' AS result FROM (VALUES (DATE '1992-01-01')) t(d);
```

result
false

However, we cannot compare VARCHAR values with dates.

```
SELECT d > '1992-01-01'::VARCHAR FROM (VALUES (DATE '1992-01-01')) t(d);
```

```
Binder Error: Cannot compare values of type DATE and type VARCHAR - an explicit cast is required
```

Escape String Literals

To escape a single quote (apostrophe) character in a string literal, use `''`. For example, `SELECT '''' AS s` returns `'`.

To include special characters such as newline, use `E` escape the string. Both the uppercase (`E'...'`) and lowercase variants (`e'...'`) work.

```
SELECT E'Hello\nworld' AS msg;
```

Or:

```
SELECT e'Hello\nworld' AS msg;
```

```
┌──────────────┐
│     msg      │
│   varchar    │
├──────────────┤
│ Hello\nworld │
└──────────────┘
```

The following backslash escape sequences are supported:

Escape sequence	Name	ASCII code
`\b`	backspace	8
`\f`	form feed	12
`\n`	newline	10
`\r`	carriage return	13
`\t`	tab	9

Dollar-Quoted String Literals

DuckDB supports dollar-quoted string literals, which are surrounded by double-dollar symbols (`$$`):

```
SELECT $$Hello
world$$ AS msg;
```

msg
varchar
Hello\nworld

```
SELECT $$The price is $9.95$$ AS msg;
```

msg
The price is $9.95

Implicit concatenation only works for single-quoted string literals, not with dollar-quoted ones.

Map 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=10, key2=20, key3=30}`:

```
SELECT MAP {'key1': 10, 'key2': 20, 'key3': 30};
```

Alternatively use the map_from_entries function. This returns `{key1=10, key2=20, key3=30}`:

```
SELECT map_from_entries([('key1', 10), ('key2', 20), ('key3', 30)]);
```

A map can be also created using two lists: keys and values. This returns `{key1=10, key2=20, key3=30}`:

```
SELECT MAP(['key1', 'key2', 'key3'], [10, 20, 30]);
```

A map can also use 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 tbl (col MAP(INTEGER, 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. This 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".

A NULL value can be inserted to any field that does not have the NOT NULL qualifier:

```
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:

> 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;
```

```
NULL
```

```
SELECT NULL IS NOT DISTINCT FROM NULL;
```

```
true
```

```
SELECT NULL IS NULL;
```

```
true
```

NULL and Functions

A function that has input argument as `NULL` **usually** returns `NULL`.

```
SELECT cos(NULL);
```

```
NULL
```

The `coalesce` function is an exception to this: it 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
```

The `ifnull` function 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` and `UHUGEINT` 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
`HUGEINT`	-	-170141183460469231731687303715884105728	$2^{127} - 1$
`UTINYINT`	-	0	255
`USMALLINT`	-	0	65535
`UINTEGER`	-	0	4294967295
`UBIGINT`	-	0	18446744073709551615
`UHUGEINT`	-	0	340282366920938463463374607431768211455

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)` (also available under the alias `NUMERIC(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 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 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)

Like for fixed-point data types, conversion from literals or casts from other datatypes to floating-point types stores inputs that cannot be represented exactly as approximations. However, it can be harder to predict what inputs are affected by this. For example, it is not surprising that `1.3::DECIMAL(1, 0) - 0.7::DECIMAL(1, 0) != 0.6::DECIMAL(1, 0)` but it may he surprising that `1.3::REAL - 0.7::REAL != 0.6::REAL`.

Additionally, whereas multiplication, addition, and subtraction of fixed-point decimal data types is exact, these operations are only approximate on floating-point binary data types.

For more complex mathematical operations, however, floating-point arithmetic is used internally and more precise results can be obtained if intermediate steps are *not* cast to fixed point formats of the same width as in- and outputs. For example, `(10::REAL / 3::REAL)::REAL * 3 = 10` whereas `(10::DECIMAL(18, 3) / 3::DECIMAL(18, 3))::DECIMAL(18, 3) * 3 = 9.999`.

In general, we advise that:

- If you require exact storage of numbers with a known number of decimal digits and require exact additions, subtractions, and multiplications (such as for monetary amounts), use the `DECIMAL` data type or its `NUMERIC` alias instead.
- If you want to do fast or complicated calculations, the floating-point data types may be more appropriate. However, if you use the results for anything important, you should evaluate your implementation carefully for corner cases (ranges, infinities, underflows, invalid operations) that may be handled differently from what you expect and you should familiarize yourself with common floating-point pitfalls. The article "What Every Computer Scientist Should Know About Floating-Point Arithmetic" by David Goldberg and the floating point series on Bruce Dawson's blog provide excellent starting points.

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. Positive numbers outside of these ranges (and negative numbers ourside the mirrored ranges) may cause errors on some platforms but will usually be converted to zero or infinity, respectively.

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.

Universally Unique Identifiers (`UUID`s)

DuckDB supports universally unique identifiers (UUIDs) through the `UUID` type. These use 128 bits and are represented internally as `HUGEINT` values. When printed, they are shown with hexadecimal characters, separated by dashes as follows: `<8 characters>-<4`

`characters>-<4 characters>-<4 characters>-<12 characters>` (using 36 characters in total). For example, `4ac7a9e9-607c-4c8a-84f3-843f0191e3fd` is a valid UUID.

To generate a new UUID, use the `uuid()` utility function.

Functions

See Numeric Functions and Operators.

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.

Creating Structs

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.

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 `{'': 1, '': 2, '': 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 {'': 1, '': 2, '': 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. In the following query, 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 a key contains a space, simply wrap it in double quotes (").

```
SELECT a."x space" FROM (SELECT {'x space': 1, 'y': 2, 'z': 3} AS a);
```

Bracket notation may also be used. Note that this uses single quotes (') since the goal is to specify a certain string key and only constant expressions may be used inside the brackets (no expressions):

```
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. When using `row` the keys will be empty strings allowing for easy insertion into a table with a struct column. Columns, however, cannot be initialized with the `row` function, and must be explicitly named. For example:

Inserting values into a struct column using the row function:

```
CREATE TABLE t1 (s STRUCT(v VARCHAR, i INTEGER));
INSERT INTO t1 VALUES (ROW('a', 42));
```

The table will contain a single entry:

```
{'v': a, 'i': 42}
```

The following produces the same result as above:

```
CREATE TABLE t1 AS (
    SELECT ROW('a', 42)::STRUCT(v VARCHAR, i INTEGER)
);
```

Initializing a struct column with the row function will fail:

```
CREATE TABLE t2 AS SELECT ROW('a');
```

```
Invalid Input Error: A table cannot be created from an unnamed struct
```

When casting structs, the names of fields have to match. Therefore, the following query will fail:

```
SELECT a::STRUCT(y INTEGER) AS b
FROM
    (SELECT {'x': 42} AS a);
```

```
Mismatch Type Error: Type STRUCT(x INTEGER) does not match with STRUCT(y INTEGER). Cannot cast STRUCTs -
element "x" in source struct was not found in target struct
```

A workaround for this would be to use `struct_pack` instead:

```
SELECT struct_pack(y := a.x) AS b
FROM
    (SELECT {'x': 42} AS a);
```

> This behavior was introduced in DuckDB v0.9.0. Previously, this query ran successfully and returned struct `{'y': 42}` as column b.

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.

> Up to DuckDB 0.10.1, nested `NULL` values were compared as follows. At the top level, nested `NULL` values obey standard SQL `NULL` comparison rules: comparing a nested `NULL` value to a nested non-`NULL` value produces a `NULL` result. Comparing nested value *members*, however, uses the internal nested value rules for `NULL`s, and a nested `NULL` value member will compare above a nested non-`NULL` value member. DuckDB 0.10.2 introduced a breaking change in semantics, described below.

Nested `NULL` values are compared following Postgres' semantics, i.e., lower nested levels are used for tie-breaking.

Functions

See Nested Functions.

Text Types

In DuckDB, strings can be stored in the `VARCHAR` field. The field allows storage of Unicode characters. Internally, the data is encoded as UTF-8.

Name	Aliases	Description
`VARCHAR`	`CHAR`, `BPCHAR`, `STRING`, `TEXT`	Variable-length character string
`VARCHAR(n)`	`STRING(n)`, `TEXT(n)`	Variable-length character string. The maximum length *n* has no effect and is only provided for compatibility.

Specifying a Length Limit

Specifying the length for the `VARCHAR`, `STRING`, and `TEXT` types is not required and has no effect on the system. Specifying the length will not improve performance or reduce storage space of the strings in the database. These variants 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
);
```

The `VARCHAR` field allows storage of Unicode characters. Internally, the data is encoded as UTF-8.

Text Type Values

Values of the text type are character strings, also known as string values or simply strings. At runtime, string values are constructed in one of the following ways:

- referencing columns whose declared or implied type is the text data type
- string literals
- casting expressions to a text type
- applying a string operator, or invoking a function that returns a text type value

Strings with Special Characters

To use special characters in string, use escape string literals or dollar-quoted string literals. Alternatively, you can use concatenation and the `chr` character function:

```
SELECT 'Hello' || chr(10) || 'world' AS msg;
```

```
┌─────────────┐
│     msg     │
│   varchar   │
├─────────────┤
│ Hello\nworld │
└─────────────┘
```

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
```

> **Warning.** 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 with nanosecond precision (ignores time zone)
TIMESTAMP	DATETIME	timestamp with microsecond 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]]`). Decimal places beyond the targeted sub-second precision are ignored.

> **Warning.** When defining timestamps using a TIMESTAMP_NS literal, the decimal places beyond *microseconds* are ignored. Note that the TIMESTAMP_NS type is able to hold nanoseconds when created e.g., via the ingestion of Parquet files.

```
SELECT TIMESTAMP_NS '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123456
```

```
SELECT TIMESTAMP '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123456
```

```
SELECT DATETIME '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123456
```

```
SELECT TIMESTAMP_MS '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123
```

```
SELECT TIMESTAMP_S '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00
```

```
SELECT TIMESTAMPTZ '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123456+00
```

```
SELECT TIMESTAMP WITH TIME ZONE '1992-09-20 11:30:00.123456789';
```

```
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.

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 queried from in the `duckdb_settings()` table function:

```
SELECT *
FROM duckdb_settings()
WHERE name = 'TimeZone';
```

name	value	description	input_type
TimeZone	Europe/Amsterdam	The current time zone	VARCHAR

```
SELECT *
FROM duckdb_settings()
WHERE name = 'Calendar';
```

name	value	description	input_type
Calendar	gregorian	The current calendar	VARCHAR

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
FROM pg_timezone_names()
ORDER BY name;
```

name	abbrev
ACT	ACT
AET	AET
AGT	AGT
ART	ART

name	abbrev
AST	AST
Africa/Abidjan	Iceland
Africa/Accra	Iceland
Africa/Addis_Ababa	EAT
Africa/Algiers	Africa/Algiers
Africa/Asmara	EAT
Africa/Asmera	EAT
Africa/Bamako	Iceland
Africa/Bangui	Africa/Bangui
Africa/Banjul	Iceland
Africa/Bissau	Africa/Bissau
Africa/Blantyre	CAT
Africa/Brazzaville	Africa/Brazzaville
Africa/Bujumbura	CAT
Africa/Cairo	ART
Africa/Casablanca	Africa/Casablanca
Africa/Ceuta	Africa/Ceuta
Africa/Conakry	Iceland
Africa/Dakar	Iceland
Africa/Dar_es_Salaam	EAT
Africa/Djibouti	EAT
Africa/Douala	Africa/Douala
Africa/El_Aaiun	Africa/El_Aaiun
Africa/Freetown	Iceland
Africa/Gaborone	CAT
Africa/Harare	CAT
Africa/Johannesburg	Africa/Johannesburg
Africa/Juba	Africa/Juba
Africa/Kampala	EAT
Africa/Khartoum	Africa/Khartoum
Africa/Kigali	CAT
Africa/Kinshasa	Africa/Kinshasa
Africa/Lagos	Africa/Lagos
Africa/Libreville	Africa/Libreville
Africa/Lome	Iceland
Africa/Luanda	Africa/Luanda
Africa/Lubumbashi	CAT
Africa/Lusaka	CAT
Africa/Malabo	Africa/Malabo

name	abbrev
Africa/Maputo	CAT
Africa/Maseru	Africa/Maseru
Africa/Mbabane	Africa/Mbabane
Africa/Mogadishu	EAT
Africa/Monrovia	Africa/Monrovia
Africa/Nairobi	EAT
Africa/Ndjamena	Africa/Ndjamena
Africa/Niamey	Africa/Niamey
Africa/Nouakchott	Iceland
Africa/Ouagadougou	Iceland
Africa/Porto-Novo	Africa/Porto-Novo
Africa/Sao_Tome	Africa/Sao_Tome
Africa/Timbuktu	Iceland
Africa/Tripoli	Libya
Africa/Tunis	Africa/Tunis
Africa/Windhoek	Africa/Windhoek
America/Adak	America/Adak
America/Anchorage	AST
America/Anguilla	PRT
America/Antigua	PRT
America/Araguaina	America/Araguaina
America/Argentina/Buenos_Aires	AGT
America/Argentina/Catamarca	America/Argentina/Catamarca
America/Argentina/ComodRivadavia	America/Argentina/ComodRivadavia
America/Argentina/Cordoba	America/Argentina/Cordoba
America/Argentina/Jujuy	America/Argentina/Jujuy
America/Argentina/La_Rioja	America/Argentina/La_Rioja
America/Argentina/Mendoza	America/Argentina/Mendoza
America/Argentina/Rio_Gallegos	America/Argentina/Rio_Gallegos
America/Argentina/Salta	America/Argentina/Salta
America/Argentina/San_Juan	America/Argentina/San_Juan
America/Argentina/San_Luis	America/Argentina/San_Luis
America/Argentina/Tucuman	America/Argentina/Tucuman
America/Argentina/Ushuaia	America/Argentina/Ushuaia
America/Aruba	PRT
America/Asuncion	America/Asuncion
America/Atikokan	America/Atikokan
America/Atka	America/Atka
America/Bahia	America/Bahia

name	abbrev
America/Bahia_Banderas	America/Bahia_Banderas
America/Barbados	America/Barbados
America/Belem	America/Belem
America/Belize	America/Belize
America/Blanc-Sablon	PRT
America/Boa_Vista	America/Boa_Vista
America/Bogota	America/Bogota
America/Boise	America/Boise
America/Buenos_Aires	AGT
America/Cambridge_Bay	America/Cambridge_Bay
America/Campo_Grande	America/Campo_Grande
America/Cancun	America/Cancun
America/Caracas	America/Caracas
America/Catamarca	America/Catamarca
America/Cayenne	America/Cayenne
America/Cayman	America/Cayman
America/Chicago	CST
America/Chihuahua	America/Chihuahua
America/Ciudad_Juarez	America/Ciudad_Juarez
America/Coral_Harbour	America/Coral_Harbour
America/Cordoba	America/Cordoba
America/Costa_Rica	America/Costa_Rica
America/Creston	PNT
America/Cuiaba	America/Cuiaba
America/Curacao	PRT
America/Danmarkshavn	America/Danmarkshavn
America/Dawson	America/Dawson
America/Dawson_Creek	America/Dawson_Creek
America/Denver	Navajo
America/Detroit	America/Detroit
America/Dominica	PRT
America/Edmonton	America/Edmonton
America/Eirunepe	America/Eirunepe
America/El_Salvador	America/El_Salvador
America/Ensenada	America/Ensenada
America/Fort_Nelson	America/Fort_Nelson
America/Fort_Wayne	IET
America/Fortaleza	America/Fortaleza
America/Glace_Bay	America/Glace_Bay

name	abbrev
America/Godthab	America/Godthab
America/Goose_Bay	America/Goose_Bay
America/Grand_Turk	America/Grand_Turk
America/Grenada	PRT
America/Guadeloupe	PRT
America/Guatemala	America/Guatemala
America/Guayaquil	America/Guayaquil
America/Guyana	America/Guyana
America/Halifax	America/Halifax
America/Havana	Cuba
America/Hermosillo	America/Hermosillo
America/Indiana/Indianapolis	IET
America/Indiana/Knox	America/Indiana/Knox
America/Indiana/Marengo	America/Indiana/Marengo
America/Indiana/Petersburg	America/Indiana/Petersburg
America/Indiana/Tell_City	America/Indiana/Tell_City
America/Indiana/Vevay	America/Indiana/Vevay
America/Indiana/Vincennes	America/Indiana/Vincennes
America/Indiana/Winamac	America/Indiana/Winamac
America/Indianapolis	IET
America/Inuvik	America/Inuvik
America/Iqaluit	America/Iqaluit
America/Jamaica	Jamaica
America/Jujuy	America/Jujuy
America/Juneau	America/Juneau
America/Kentucky/Louisville	America/Kentucky/Louisville
America/Kentucky/Monticello	America/Kentucky/Monticello
America/Knox_IN	America/Knox_IN
America/Kralendijk	PRT
America/La_Paz	America/La_Paz
America/Lima	America/Lima
America/Los_Angeles	PST
America/Louisville	America/Louisville
America/Lower_Princes	PRT
America/Maceio	America/Maceio
America/Managua	America/Managua
America/Manaus	America/Manaus
America/Marigot	PRT
America/Martinique	America/Martinique

name	abbrev
America/Matamoros	America/Matamoros
America/Mazatlan	America/Mazatlan
America/Mendoza	America/Mendoza
America/Menominee	America/Menominee
America/Merida	America/Merida
America/Metlakatla	America/Metlakatla
America/Mexico_City	America/Mexico_City
America/Miquelon	America/Miquelon
America/Moncton	America/Moncton
America/Monterrey	America/Monterrey
America/Montevideo	America/Montevideo
America/Montreal	America/Montreal
America/Montserrat	PRT
America/Nassau	America/Nassau
America/New_York	America/New_York
America/Nipigon	America/Nipigon
America/Nome	America/Nome
America/Noronha	America/Noronha
America/North_Dakota/Beulah	America/North_Dakota/Beulah
America/North_Dakota/Center	America/North_Dakota/Center
America/North_Dakota/New_Salem	America/North_Dakota/New_Salem
America/Nuuk	America/Nuuk
America/Ojinaga	America/Ojinaga
America/Panama	America/Panama
America/Pangnirtung	America/Pangnirtung
America/Paramaribo	America/Paramaribo
America/Phoenix	PNT
America/Port-au-Prince	America/Port-au-Prince
America/Port_of_Spain	PRT
America/Porto_Acre	America/Porto_Acre
America/Porto_Velho	America/Porto_Velho
America/Puerto_Rico	PRT
America/Punta_Arenas	America/Punta_Arenas
America/Rainy_River	America/Rainy_River
America/Rankin_Inlet	America/Rankin_Inlet
America/Recife	America/Recife
America/Regina	America/Regina
America/Resolute	America/Resolute
America/Rio_Branco	America/Rio_Branco

name	abbrev
America/Rosario	America/Rosario
America/Santa_Isabel	America/Santa_Isabel
America/Santarem	America/Santarem
America/Santiago	America/Santiago
America/Santo_Domingo	America/Santo_Domingo
America/Sao_Paulo	BET
America/Scoresbysund	America/Scoresbysund
America/Shiprock	Navajo
America/Sitka	America/Sitka
America/St_Barthelemy	PRT
America/St_Johns	CNT
America/St_Kitts	PRT
America/St_Lucia	PRT
America/St_Thomas	PRT
America/St_Vincent	PRT
America/Swift_Current	America/Swift_Current
America/Tegucigalpa	America/Tegucigalpa
America/Thule	America/Thule
America/Thunder_Bay	America/Thunder_Bay
America/Tijuana	America/Tijuana
America/Toronto	America/Toronto
America/Tortola	PRT
America/Vancouver	America/Vancouver
America/Virgin	PRT
America/Whitehorse	America/Whitehorse
America/Winnipeg	America/Winnipeg
America/Yakutat	America/Yakutat
America/Yellowknife	America/Yellowknife
Antarctica/Casey	Antarctica/Casey
Antarctica/Davis	Antarctica/Davis
Antarctica/DumontDUrville	Antarctica/DumontDUrville
Antarctica/Macquarie	Antarctica/Macquarie
Antarctica/Mawson	Antarctica/Mawson
Antarctica/McMurdo	NZ
Antarctica/Palmer	Antarctica/Palmer
Antarctica/Rothera	Antarctica/Rothera
Antarctica/South_Pole	NZ
Antarctica/Syowa	Antarctica/Syowa
Antarctica/Troll	Antarctica/Troll

name	abbrev
Antarctica/Vostok	Antarctica/Vostok
Arctic/Longyearbyen	Arctic/Longyearbyen
Asia/Aden	Asia/Aden
Asia/Almaty	Asia/Almaty
Asia/Amman	Asia/Amman
Asia/Anadyr	Asia/Anadyr
Asia/Aqtau	Asia/Aqtau
Asia/Aqtobe	Asia/Aqtobe
Asia/Ashgabat	Asia/Ashgabat
Asia/Ashkhabad	Asia/Ashkhabad
Asia/Atyrau	Asia/Atyrau
Asia/Baghdad	Asia/Baghdad
Asia/Bahrain	Asia/Bahrain
Asia/Baku	Asia/Baku
Asia/Bangkok	Asia/Bangkok
Asia/Barnaul	Asia/Barnaul
Asia/Beirut	Asia/Beirut
Asia/Bishkek	Asia/Bishkek
Asia/Brunei	Asia/Brunei
Asia/Calcutta	IST
Asia/Chita	Asia/Chita
Asia/Choibalsan	Asia/Choibalsan
Asia/Chongqing	CTT
Asia/Chungking	CTT
Asia/Colombo	Asia/Colombo
Asia/Dacca	BST
Asia/Damascus	Asia/Damascus
Asia/Dhaka	BST
Asia/Dili	Asia/Dili
Asia/Dubai	Asia/Dubai
Asia/Dushanbe	Asia/Dushanbe
Asia/Famagusta	Asia/Famagusta
Asia/Gaza	Asia/Gaza
Asia/Harbin	CTT
Asia/Hebron	Asia/Hebron
Asia/Ho_Chi_Minh	VST
Asia/Hong_Kong	Hongkong
Asia/Hovd	Asia/Hovd
Asia/Irkutsk	Asia/Irkutsk

name	abbrev
Asia/Istanbul	Turkey
Asia/Jakarta	Asia/Jakarta
Asia/Jayapura	Asia/Jayapura
Asia/Jerusalem	Israel
Asia/Kabul	Asia/Kabul
Asia/Kamchatka	Asia/Kamchatka
Asia/Karachi	PLT
Asia/Kashgar	Asia/Kashgar
Asia/Kathmandu	Asia/Kathmandu
Asia/Katmandu	Asia/Katmandu
Asia/Khandyga	Asia/Khandyga
Asia/Kolkata	IST
Asia/Krasnoyarsk	Asia/Krasnoyarsk
Asia/Kuala_Lumpur	Singapore
Asia/Kuching	Asia/Kuching
Asia/Kuwait	Asia/Kuwait
Asia/Macao	Asia/Macao
Asia/Macau	Asia/Macau
Asia/Magadan	Asia/Magadan
Asia/Makassar	Asia/Makassar
Asia/Manila	Asia/Manila
Asia/Muscat	Asia/Muscat
Asia/Nicosia	Asia/Nicosia
Asia/Novokuznetsk	Asia/Novokuznetsk
Asia/Novosibirsk	Asia/Novosibirsk
Asia/Omsk	Asia/Omsk
Asia/Oral	Asia/Oral
Asia/Phnom_Penh	Asia/Phnom_Penh
Asia/Pontianak	Asia/Pontianak
Asia/Pyongyang	Asia/Pyongyang
Asia/Qatar	Asia/Qatar
Asia/Qostanay	Asia/Qostanay
Asia/Qyzylorda	Asia/Qyzylorda
Asia/Rangoon	Asia/Rangoon
Asia/Riyadh	Asia/Riyadh
Asia/Saigon	VST
Asia/Sakhalin	Asia/Sakhalin
Asia/Samarkand	Asia/Samarkand
Asia/Seoul	ROK

name	abbrev
Asia/Shanghai	CTT
Asia/Singapore	Singapore
Asia/Srednekolymsk	Asia/Srednekolymsk
Asia/Taipei	ROC
Asia/Tashkent	Asia/Tashkent
Asia/Tbilisi	Asia/Tbilisi
Asia/Tehran	Iran
Asia/Tel_Aviv	Israel
Asia/Thimbu	Asia/Thimbu
Asia/Thimphu	Asia/Thimphu
Asia/Tokyo	JST
Asia/Tomsk	Asia/Tomsk
Asia/Ujung_Pandang	Asia/Ujung_Pandang
Asia/Ulaanbaatar	Asia/Ulaanbaatar
Asia/Ulan_Bator	Asia/Ulan_Bator
Asia/Urumqi	Asia/Urumqi
Asia/Ust-Nera	Asia/Ust-Nera
Asia/Vientiane	Asia/Vientiane
Asia/Vladivostok	Asia/Vladivostok
Asia/Yakutsk	Asia/Yakutsk
Asia/Yangon	Asia/Yangon
Asia/Yekaterinburg	Asia/Yekaterinburg
Asia/Yerevan	NET
Atlantic/Azores	Atlantic/Azores
Atlantic/Bermuda	Atlantic/Bermuda
Atlantic/Canary	Atlantic/Canary
Atlantic/Cape_Verde	Atlantic/Cape_Verde
Atlantic/Faeroe	Atlantic/Faeroe
Atlantic/Faroe	Atlantic/Faroe
Atlantic/Jan_Mayen	Atlantic/Jan_Mayen
Atlantic/Madeira	Atlantic/Madeira
Atlantic/Reykjavik	Iceland
Atlantic/South_Georgia	Atlantic/South_Georgia
Atlantic/St_Helena	Iceland
Atlantic/Stanley	Atlantic/Stanley
Australia/ACT	AET
Australia/Adelaide	Australia/Adelaide
Australia/Brisbane	Australia/Brisbane
Australia/Broken_Hill	Australia/Broken_Hill

name	abbrev
Australia/Canberra	AET
Australia/Currie	Australia/Currie
Australia/Darwin	ACT
Australia/Eucla	Australia/Eucla
Australia/Hobart	Australia/Hobart
Australia/LHI	Australia/LHI
Australia/Lindeman	Australia/Lindeman
Australia/Lord_Howe	Australia/Lord_Howe
Australia/Melbourne	Australia/Melbourne
Australia/NSW	AET
Australia/North	ACT
Australia/Perth	Australia/Perth
Australia/Queensland	Australia/Queensland
Australia/South	Australia/South
Australia/Sydney	AET
Australia/Tasmania	Australia/Tasmania
Australia/Victoria	Australia/Victoria
Australia/West	Australia/West
Australia/Yancowinna	Australia/Yancowinna
BET	BET
BST	BST
Brazil/Acre	Brazil/Acre
Brazil/DeNoronha	Brazil/DeNoronha
Brazil/East	BET
Brazil/West	Brazil/West
CAT	CAT
CET	CET
CNT	CNT
CST	CST
CST6CDT	CST6CDT
CTT	CTT
Canada/Atlantic	Canada/Atlantic
Canada/Central	Canada/Central
Canada/East-Saskatchewan	Canada/East-Saskatchewan
Canada/Eastern	Canada/Eastern
Canada/Mountain	Canada/Mountain
Canada/Newfoundland	CNT
Canada/Pacific	Canada/Pacific
Canada/Saskatchewan	Canada/Saskatchewan

name	abbrev
Canada/Yukon	Canada/Yukon
Chile/Continental	Chile/Continental
Chile/EasterIsland	Chile/EasterIsland
Cuba	Cuba
EAT	EAT
ECT	ECT
EET	EET
EST	EST
EST5EDT	EST5EDT
Egypt	ART
Eire	Eire
Etc/GMT	GMT
Etc/GMT+0	GMT
Etc/GMT+1	Etc/GMT+1
Etc/GMT+10	Etc/GMT+10
Etc/GMT+11	Etc/GMT+11
Etc/GMT+12	Etc/GMT+12
Etc/GMT+2	Etc/GMT+2
Etc/GMT+3	Etc/GMT+3
Etc/GMT+4	Etc/GMT+4
Etc/GMT+5	Etc/GMT+5
Etc/GMT+6	Etc/GMT+6
Etc/GMT+7	Etc/GMT+7
Etc/GMT+8	Etc/GMT+8
Etc/GMT+9	Etc/GMT+9
Etc/GMT-0	GMT
Etc/GMT-1	Etc/GMT-1
Etc/GMT-10	Etc/GMT-10
Etc/GMT-11	Etc/GMT-11
Etc/GMT-12	Etc/GMT-12
Etc/GMT-13	Etc/GMT-13
Etc/GMT-14	Etc/GMT-14
Etc/GMT-2	Etc/GMT-2
Etc/GMT-3	Etc/GMT-3
Etc/GMT-4	Etc/GMT-4
Etc/GMT-5	Etc/GMT-5
Etc/GMT-6	Etc/GMT-6
Etc/GMT-7	Etc/GMT-7
Etc/GMT-8	Etc/GMT-8

name	abbrev
Etc/GMT-9	Etc/GMT-9
Etc/GMT0	GMT
Etc/Greenwich	GMT
Etc/UCT	UCT
Etc/UTC	UCT
Etc/Universal	UCT
Etc/Zulu	UCT
Europe/Amsterdam	Europe/Amsterdam
Europe/Andorra	Europe/Andorra
Europe/Astrakhan	Europe/Astrakhan
Europe/Athens	Europe/Athens
Europe/Belfast	GB
Europe/Belgrade	Europe/Belgrade
Europe/Berlin	Europe/Berlin
Europe/Bratislava	Europe/Bratislava
Europe/Brussels	Europe/Brussels
Europe/Bucharest	Europe/Bucharest
Europe/Budapest	Europe/Budapest
Europe/Busingen	Europe/Busingen
Europe/Chisinau	Europe/Chisinau
Europe/Copenhagen	Europe/Copenhagen
Europe/Dublin	Eire
Europe/Gibraltar	Europe/Gibraltar
Europe/Guernsey	GB
Europe/Helsinki	Europe/Helsinki
Europe/Isle_of_Man	GB
Europe/Istanbul	Turkey
Europe/Jersey	GB
Europe/Kaliningrad	Europe/Kaliningrad
Europe/Kiev	Europe/Kiev
Europe/Kirov	Europe/Kirov
Europe/Kyiv	Europe/Kyiv
Europe/Lisbon	Portugal
Europe/Ljubljana	Europe/Ljubljana
Europe/London	GB
Europe/Luxembourg	Europe/Luxembourg
Europe/Madrid	Europe/Madrid
Europe/Malta	Europe/Malta
Europe/Mariehamn	Europe/Mariehamn

name	abbrev
Europe/Minsk	Europe/Minsk
Europe/Monaco	ECT
Europe/Moscow	W-SU
Europe/Nicosia	Europe/Nicosia
Europe/Oslo	Europe/Oslo
Europe/Paris	ECT
Europe/Podgorica	Europe/Podgorica
Europe/Prague	Europe/Prague
Europe/Riga	Europe/Riga
Europe/Rome	Europe/Rome
Europe/Samara	Europe/Samara
Europe/San_Marino	Europe/San_Marino
Europe/Sarajevo	Europe/Sarajevo
Europe/Saratov	Europe/Saratov
Europe/Simferopol	Europe/Simferopol
Europe/Skopje	Europe/Skopje
Europe/Sofia	Europe/Sofia
Europe/Stockholm	Europe/Stockholm
Europe/Tallinn	Europe/Tallinn
Europe/Tirane	Europe/Tirane
Europe/Tiraspol	Europe/Tiraspol
Europe/Ulyanovsk	Europe/Ulyanovsk
Europe/Uzhgorod	Europe/Uzhgorod
Europe/Vaduz	Europe/Vaduz
Europe/Vatican	Europe/Vatican
Europe/Vienna	Europe/Vienna
Europe/Vilnius	Europe/Vilnius
Europe/Volgograd	Europe/Volgograd
Europe/Warsaw	Poland
Europe/Zagreb	Europe/Zagreb
Europe/Zaporozhye	Europe/Zaporozhye
Europe/Zurich	Europe/Zurich
Factory	Factory
GB	GB
GB-Eire	GB
GMT	GMT
GMT+0	GMT
GMT-0	GMT
GMT0	GMT

name	abbrev
Greenwich	GMT
HST	HST
Hongkong	Hongkong
IET	IET
IST	IST
Iceland	Iceland
Indian/Antananarivo	EAT
Indian/Chagos	Indian/Chagos
Indian/Christmas	Indian/Christmas
Indian/Cocos	Indian/Cocos
Indian/Comoro	EAT
Indian/Kerguelen	Indian/Kerguelen
Indian/Mahe	Indian/Mahe
Indian/Maldives	Indian/Maldives
Indian/Mauritius	Indian/Mauritius
Indian/Mayotte	EAT
Indian/Reunion	Indian/Reunion
Iran	Iran
Israel	Israel
JST	JST
Jamaica	Jamaica
Japan	JST
Kwajalein	Kwajalein
Libya	Libya
MET	MET
MIT	MIT
MST	MST
MST7MDT	MST7MDT
Mexico/BajaNorte	Mexico/BajaNorte
Mexico/BajaSur	Mexico/BajaSur
Mexico/General	Mexico/General
NET	NET
NST	NZ
NZ	NZ
NZ-CHAT	NZ-CHAT
Navajo	Navajo
PLT	PLT
PNT	PNT
PRC	CTT

name	abbrev
PRT	PRT
PST	PST
PST8PDT	PST8PDT
Pacific/Apia	MIT
Pacific/Auckland	NZ
Pacific/Bougainville	Pacific/Bougainville
Pacific/Chatham	NZ-CHAT
Pacific/Chuuk	Pacific/Chuuk
Pacific/Easter	Pacific/Easter
Pacific/Efate	Pacific/Efate
Pacific/Enderbury	Pacific/Enderbury
Pacific/Fakaofo	Pacific/Fakaofo
Pacific/Fiji	Pacific/Fiji
Pacific/Funafuti	Pacific/Funafuti
Pacific/Galapagos	Pacific/Galapagos
Pacific/Gambier	Pacific/Gambier
Pacific/Guadalcanal	SST
Pacific/Guam	Pacific/Guam
Pacific/Honolulu	Pacific/Honolulu
Pacific/Johnston	Pacific/Johnston
Pacific/Kanton	Pacific/Kanton
Pacific/Kiritimati	Pacific/Kiritimati
Pacific/Kosrae	Pacific/Kosrae
Pacific/Kwajalein	Kwajalein
Pacific/Majuro	Pacific/Majuro
Pacific/Marquesas	Pacific/Marquesas
Pacific/Midway	Pacific/Midway
Pacific/Nauru	Pacific/Nauru
Pacific/Niue	Pacific/Niue
Pacific/Norfolk	Pacific/Norfolk
Pacific/Noumea	Pacific/Noumea
Pacific/Pago_Pago	Pacific/Pago_Pago
Pacific/Palau	Pacific/Palau
Pacific/Pitcairn	Pacific/Pitcairn
Pacific/Pohnpei	SST
Pacific/Ponape	SST
Pacific/Port_Moresby	Pacific/Port_Moresby
Pacific/Rarotonga	Pacific/Rarotonga
Pacific/Saipan	Pacific/Saipan

name	abbrev
Pacific/Samoa	Pacific/Samoa
Pacific/Tahiti	Pacific/Tahiti
Pacific/Tarawa	Pacific/Tarawa
Pacific/Tongatapu	Pacific/Tongatapu
Pacific/Truk	Pacific/Truk
Pacific/Wake	Pacific/Wake
Pacific/Wallis	Pacific/Wallis
Pacific/Yap	Pacific/Yap
Poland	Poland
Portugal	Portugal
ROC	ROC
ROK	ROK
SST	SST
Singapore	Singapore
SystemV/AST4	SystemV/AST4
SystemV/AST4ADT	SystemV/AST4ADT
SystemV/CST6	SystemV/CST6
SystemV/CST6CDT	SystemV/CST6CDT
SystemV/EST5	SystemV/EST5
SystemV/EST5EDT	SystemV/EST5EDT
SystemV/HST10	SystemV/HST10
SystemV/MST7	SystemV/MST7
SystemV/MST7MDT	SystemV/MST7MDT
SystemV/PST8	SystemV/PST8
SystemV/PST8PDT	SystemV/PST8PDT
SystemV/YST9	SystemV/YST9
SystemV/YST9YDT	SystemV/YST9YDT
Turkey	Turkey
UCT	UCT
US/Alaska	AST
US/Aleutian	US/Aleutian
US/Arizona	PNT
US/Central	CST
US/East-Indiana	IET
US/Eastern	US/Eastern
US/Hawaii	US/Hawaii
US/Indiana-Starke	US/Indiana-Starke
US/Michigan	US/Michigan
US/Mountain	Navajo

name	abbrev
US/Pacific	PST
US/Pacific-New	PST
US/Samoa	US/Samoa
UTC	UCT
Universal	UCT
VST	VST
W-SU	W-SU
WET	WET
Zulu	UCT

Union 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 INTEGER, str VARCHAR));
INSERT INTO tbl1 values (1), ('two'), (union_value(str := 'three'));
```

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 target's (if the cast is unambiguous).

`UNION` uses the member types' VARCHAR cast functions when casting to VARCHAR:

```
SELECT u FROM tbl1;
```

u
1
two
three

Select all the `str` members:

```
SELECT union_extract(u, 'str') AS str
FROM tbl1;
```

str
NULL
two
three

Alternatively, you can use 'dot syntax' similarly to STRUCTs.

```
SELECT u.str
FROM tbl1;
```

str
NULL
two
three

Select the currently active tag from the UNION as an ENUM.

```
SELECT union_tag(u) AS t
FROM tbl1;
```

t
num
str
str

Union Casts

Compared to other nested types, UNIONs 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 UNIONs 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 INTEGER, v VARCHAR)` cast will always cast the FLOAT to the INTEGER 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 INTEGER, num INTEGER)` 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::INTEGER)` will create a UNION with a single member of type INTEGER 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 INTEGER, b INTEGER))`.

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.

Typecasting

Typecasting is an operation that converts a value in one particular data type to the closest corresponding value in another data type. Like other SQL engines, DuckDB supports both implicit and explicit typecasting.

Explicit Casting

Explicit typecasting is performed by using a CAST expression. For example, `CAST(col AS VARCHAR)` or `col::VARCHAR` explicitly cast the column `col` to VARCHAR. See the cast page for more information.

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.

Implicit casts can only be added for a number of type combinations, and is generally only possible when the cast cannot fail. For example, an implicit cast can be added from `INTEGER` to `DOUBLE` – but not from `DOUBLE` to `INTEGER`.

Casting Operations Matrix

Values of a particular data type cannot always be cast to any arbitrary target data type. The only exception is the `NULL` value – which can always be converted between types. The following matrix describes which conversions are supported. When implicit casting is allowed, it implies that explicit casting is also possible.

from \ to	BLOB	INTERVAL	DATE	TIME	TIMESTAMP	BOOLEAN	BIT	DOUBLE	FLOAT	HUGEINT	BIGINT	INTEGER	SMALLINT	TINYINT	UBIGINT	UINTEGER	USMALLINT	UTINYINT	DECIMAL	UUID	VARCHAR
BLOB		✗	✗	✗	✗	✗	■	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	■
INTERVAL	✗		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
DATE	✗	✗		✗	●	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
TIME	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
TIMESTAMP	✗	✗	■	■		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
BOOLEAN	✗	✗	✗	✗	✗		■	■	■	■	■	■	■	■	■	■	■	■	■	✗	●
BIT	■	✗	✗	✗	✗	✗		■	■	■	■	■	✗	✗	■	■	✗	✗	✗	✗	●
DOUBLE	✗	✗	✗	✗	✗	■	■		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
FLOAT	✗	✗	✗	✗	✗	■	■	●		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
HUGEINT	✗	✗	✗	✗	✗	■	■	●	●		✗	✗	✗	✗	✗	✗	✗	✗	●	✗	●
BIGINT	✗	✗	✗	✗	✗	■	■	●	●	●		✗	✗	✗	✗	✗	✗	✗	●	✗	●
INTEGER	✗	✗	✗	✗	✗	■	■	●	●	●	●		✗	✗	✗	✗	✗	✗	●	✗	●
SMALLINT	✗	✗	✗	✗	✗	■	■	●	●	●	●	●		✗	✗	✗	✗	✗	●	✗	●
TINYINT	✗	✗	✗	✗	✗	■	■	●	●	●	●	●	●		✗	✗	✗	✗	●	✗	●
UBIGINT	✗	✗	✗	✗	✗	■	■	●	●	●	■	■	■	■		■	■	■	●	✗	●
UINTEGER	✗	✗	✗	✗	✗	■	■	●	●	●	●	■	■	■	●		■	■	●	✗	●
USMALLINT	✗	✗	✗	✗	✗	■	■	●	●	●	●	●	■	■	●	●		■	●	✗	●
UTINYINT	✗	✗	✗	✗	✗	■	■	●	●	●	●	●	●	■	●	●	●		●	✗	●
DECIMAL	✗	✗	✗	✗	✗	■	✗	●	●	■	■	■	■	✗	✗	✗	✗	✗		✗	●
UUID	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗		●
VARCHAR	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	

Legend:
✗ casting is not allowed
● implicit casting
■ explicit casting

Even though a casting operation is supported based on the source and target data type, it does not necessarily mean the cast operation will succeed at runtime.

> **Deprecated.** Prior to version 0.10.0, DuckDB allowed any type to be implicitly cast to VARCHAR during function binding. Version 0.10.0 introduced a breaking change which no longer allows implicit casts to VARCHAR. The `old_implicit_casting` configuration option setting can be used to revert to the old behavior. However, please note that this flag will be deprecated in the future.

Lossy Casts

Casting operations that result in loss of precision are allowed. For example, it is possible to explicitly cast a numeric type with fractional digits like `DECIMAL`, `FLOAT` or `DOUBLE` to an integral type like `INTEGER`. The number will be rounded.

```
SELECT CAST(3.5 AS INTEGER);
```

Overflows

Casting operations that would result in a value overflow throw an error. For example, the value `999` is too large to be represented by the `TINYINT` data type. Therefore, an attempt to cast that value to that type results in a runtime error:

```
SELECT CAST(999 AS TINYINT);
```

```
Conversion Error: Type INT32 with value 999 can't be cast because the value is out of range for the
destination type INT8
```

So even though the cast operation from `INTEGER` to `TINYINT` is supported, it is not possible for this particular value. `TRY_CAST` can be used to convert the value into `NULL` instead of throwing an error.

Varchar

The `VARCHAR` type acts as a univeral target: any arbitrary value of any arbitrary type can always be cast to the VARCHAR type. This type is also used for displaying values in the shell.

```
SELECT CAST(42.5 AS VARCHAR);
```

Casting from VARCHAR to another data type is supported, but can raise an error at runtime if DuckDB cannot parse and convert the provided text to the target data type.

```
SELECT CAST('NotANumber' AS INTEGER);
```

In general, casting to VARCHAR is a lossless operation and any type can be cast back to the original type after being converted into text.

```
SELECT CAST(CAST([1, 2, 3] AS VARCHAR) AS INTEGER[]);
```

Literal Types

Integer literals (such as `42`) and string literals (such as `'string'`) have special implicit casting rules. See the literal types page for more information.

Lists / Arrays

Lists can be explicitly cast to other lists using the same casting rules. The cast is applied to the children of the list. For example, if we convert a `INTEGER[]` list to a `VARCHAR[]` list, the child `INTEGER` elements are individually cast to `VARCHAR` and a new list is constructed.

```
SELECT CAST([1, 2, 3] AS VARCHAR[]);
```

Arrays

Arrays follow the same casting rules as lists. In addition, arrays can be implicitly cast to lists of the same type. For example, an `INTEGER[3]` array can be implicitly cast to an `INTEGER[]` list.

Structs

Structs can be cast to other structs as long as the names of the child elements match.

```
SELECT CAST({'a': 42} AS STRUCT(a VARCHAR));
```

The names of the struct can also be in a different order. The fields of the struct will be reshuffled based on the names of the structs.

```
SELECT CAST({'a': 42, 'b': 84} AS STRUCT(b VARCHAR, a VARCHAR));
```

Unions

Union casting rules can be found on the `UNION type page`.

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;
```

i	test
1	0
2	0
3	1

This is equivalent to:

```
SELECT i, IF(i > 2, 1, 0) AS test
FROM integers;
```

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;
```

i	test
1	10
2	20
3	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;
```

i	test
1	10
2	NULL
3	NULL

It is also possible to provide an individual expression after the CASE but before the WHEN. When this is done, the CASE statement is effectively 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;
```

i	test
1	10
2	20
3	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 operation of converting a value in a particular data type to the corresponding value in another data type. Casting can occur either implicitly or explicitly. The syntax described here performs an explicit cast. More information on casting can be found on the typecasting page.

Explicit Casting

The standard SQL syntax for explicit casting is `CAST(expr AS TYPENAME)`, where TYPENAME is a name (or alias) of one of DuckDB's data types. DuckDB also supports the shorthand `expr::TYPENAME`, which is also present in PostgreSQL.

```
SELECT CAST(i AS VARCHAR) AS i FROM generate_series(1, 3) tbl(i);
```

i
1
2
3

```
SELECT i::DOUBLE AS i FROM generate_series(1, 3) tbl(i);
```

i
1.0
2.0
3.0

Casting Rules

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.

```
SELECT CAST('hello' AS INTEGER);
```

```
Conversion Error: Could not convert string 'hello' to INT32
```

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.

TRY_CAST

`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.

```
SELECT TRY_CAST('hello' AS INTEGER) AS i;
```

i
NULL

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 Unicode normalization 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.

```
SET 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 names.name AS name, other_names.name AS other_name
FROM names, other_names
WHERE names.name = other_names.name;
```

```
ERROR: Cannot combine types with different collation!
```

We need to manually overwrite the collation:

```
SELECT names.name AS name, other_names.name AS other_name
FROM names, other_names
WHERE names.name COLLATE NOACCENT.NOCASE = other_names.name COLLATE NOACCENT.NOCASE;
```

name	other_name
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`

Operator	Description	Example	Result
>=	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 x <= a AND a <= y
a NOT BETWEEN x AND y	equivalent to x > a 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)

> For the expression BETWEEN x AND y, x is used as the lower bound and y is used as the upper bound. Therefore, if x > y, the result will always be false.

IN Operator

IN

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`

`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)`.

Use with Subqueries

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.

Binary Operators: `AND` and `OR`

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

Unary Operator: `NOT`

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;
```

Count the number of rows in a table:

```
SELECT count(*) 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 Expression

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 INTEGER, number INTEGER);
INSERT INTO numbers VALUES (1, 10), (2, 20), (3, NULL);
SELECT min(COLUMNS(*)), count(COLUMNS(*)) FROM numbers;
```

id	number	id	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;
```

id	min(number := (number + id))	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;
```

id	number
2	20
4	40
6	NULL

COLUMNS expressions can also be used in WHERE clauses. The conditions are applied to all columns and are combined using the logical AND operator.

```
SELECT *
FROM (
    SELECT 0 AS x, 1 AS y, 2 AS z
    UNION ALL
    SELECT 1 AS x, 2 AS y, 3 AS z
    UNION ALL
    SELECT 2 AS x, 3 AS y, 4 AS z
)
WHERE COLUMNS(*) > 1; -- equivalent to: x > 1 AND y > 1 AND z > 1
```

x	y	z
2	3	4

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

The matches of capture groups can be used to rename columns selected by a regular expression:

```
SELECT COLUMNS('(\w{2}).*') AS '\1' FROM numbers;
```

id	nu
1	10
2	20
3	NULL

The capture groups are one-indexed; `\0` is the original column name.

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 data type and nested functions pages for more details on working with structs.

For example:

```
SELECT st.* FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS st);
```

x	y	z
1	2	3

Subqueries

Subqueries are parenthesized query expressions that appear as part of a larger, outer query. Subqueries are usually based on `SELECT ... FROM`, but in DuckDB other query constructs such as `PIVOT` can also appear as a subquery.

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;
```

min(grade)
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);
```

course
Math

Subquery Comparisons: ALL, ANY and SOME

In the section on scalar subqueries, a scalar expression was compared directly to a subquery using the equality comparison operator (=). Such direct comparisons only make sense with scalar subqueries.

Scalar expressions can still be compared to single-column subqueries returning multiple rows by specifying a quantifier. Available quantifiers are `ALL`, `ANY` and `SOME`. The quantifiers `ANY` and `SOME` are equivalent.

ALL

The `ALL` quantifier specifies that the comparison as a whole evaluates to `true` when the individual comparison results of *the expression at the left hand side of the comparison operator* with each of the values from *the subquery at the right hand side of the comparison operator* **all** evaluate to `true`:

```
SELECT 6 <= ALL (SELECT grade FROM grades) AS adequate;
```

returns:

adequate
true

because 6 is less than or equal to each of the subquery results 7, 8 and 9.

However, the following query

```
SELECT 8 >= ALL (SELECT grade FROM grades) AS excellent;
```

returns

excellent
false

because 8 is not greater than or equal to the subquery result 7. And thus, because not all comparisons evaluate `true`, `>= ALL` as a whole evaluates to `false`.

ANY

The `ANY` quantifier specifies that the comparison as a whole evaluates to `true` when at least one of the individual comparison results evaluates to `true`. For example:

```
SELECT 5 >= ANY (SELECT grade FROM grades) AS fail;
```

returns

fail
false

because no result of the subquery is less than or equal to 5.

The quantifier `SOME` maybe used instead of `ANY`: `ANY` and `SOME` are interchangeable.

> In DuckDB, and contrary to most SQL implementations, a comparison of a scalar with a single-column subquery returning multiple values still executes without error. However, the result is unstable, as the final comparison result is based on comparing just one (non-deterministically selected) value returned by the subquery.

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') AS math_grades_present;
```

math_grades_present
true

```
SELECT EXISTS (SELECT * FROM grades WHERE course = 'History') AS history_grades_present;
```

history_grades_present
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	name
1	Jane

> 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) AS math_grades_present;
```

math_grades_present
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);
```

grade	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
{'x': 41, 'y': hello}
{'x': 42, 'y': hello}
{'x': 43, 'y': hello}

Functions

Functions

Function Syntax

Function Chaining via the Dot Operator

DuckDB supports the dot syntax for function chaining. This allows the function call `fn(arg1, arg2, arg3, ...)` to be rewritten as `arg1.fn(arg2, arg3, ...)`. For example, take the following use of the `replace` function:

```
SELECT replace(goose_name, 'goose', 'duck') AS duck_name
FROM unnest(['African goose', 'Faroese goose', 'Hungarian goose', 'Pomeranian goose']) breed(goose_name);
```

This can be rewritten as follows:

```
SELECT goose_name.replace('goose', 'duck') AS duck_name
FROM unnest(['African goose', 'Faroese goose', 'Hungarian goose', 'Pomeranian goose']) breed(goose_name);
```

Query Functions

The `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
bar	scalar	VARCHAR	[x, min, max, width]	[DOUBLE, DOUBLE, DOUBLE, DOUBLE]	Draws a band whose width is proportional to (x - min) and equal to width characters when x = max. width defaults to 80
base64	scalar	VARCHAR	[blob]	[BLOB]	Convert a blob to a base64 encoded string
bin	scalar	VARCHAR	[value]	[VARCHAR]	Converts the value to binary representation
bit_count	scalar	TINYINT	[x]	[TINYINT]	Returns the number of bits that are set
bit_length	scalar	BIGINT	[col0]	[VARCHAR]	NULL

function_name	function_type	return_type	parameters	parameter_types	description
bit_position	scalar	INTEGER	[substring, bitstring]	[BIT, BIT]	Returns first starting index of the specified substring within bits, or zero if it is not present. The first (leftmost) bit is indexed 1
bitstring	scalar	BIT	[bitstring, length]	[VARCHAR, INTEGER]	Pads the bitstring until the specified length

> Currently, the description and parameter names of functions are not available in the `duckdb_functions()` function.

Array Functions

All `LIST` functions work with the ARRAY data type. Additionally, several ARRAY-native functions are also supported.

Array-Native Functions

Function	Description
`array_value(index)`	Create an ARRAY containing the argument values.
`array_cross_product(array1, array2)`	Compute the cross product of two arrays of size 3. The array elements can not be `NULL`.
`array_cosine_similarity(array1, array2)`	Compute the cosine similarity between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
`array_distance(array1, array2)`	Compute the distance between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
`array_inner_product(array1, array2)`	Compute the inner product between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
`array_dot_product(array1, array2)`	Alias for `array_inner_product(array1, array2)`.

`array_value(index)`

Description	Create an ARRAY containing the argument values.
Example	`array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT)`
Result	`[1.0, 2.0, 3.0]`

`array_cross_product(array1, array2)`

Description	Compute the cross product of two arrays of size 3. The array elements can not be `NULL`.
Example	`array_cross_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`[-1.0, 2.0, -1.0]`

`array_cosine_similarity(array1, array2)`

Description	Compute the cosine similarity between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
Example	`array_cosine_similarity(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`0.9925833`

`array_distance(array1, array2)`

Description	Compute the distance between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
Example	`array_distance(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`1.7320508`

`array_inner_product(array1, array2)`

Description	Compute the inner product between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
Example	`array_inner_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`20.0`

`array_dot_product(array1, array2)`

Description	Alias for `array_inner_product(array1, array2)`.

Example	`array_dot_product(l1, l2)`
Result	`20.0`

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.

Name	Description
`bit_count(bitstring)`	Returns the number of set bits in the bitstring.
`bit_length(bitstring)`	Returns the number of bits in the bitstring.
`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.
`bitstring(bitstring, length)`	Returns a bitstring of determined length.
`get_bit(bitstring, index)`	Extracts the nth bit from bitstring; the first (leftmost) bit is indexed 0.
`length(bitstring)`	Alias for `bit_length`.
`octet_length(bitstring)`	Returns the number of bytes in the bitstring.
`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.

`bit_count(bitstring)`

Description	Returns the number of set bits in the bitstring.
Example	`bit_count('1101011'::BIT)`
Result	5

`bit_length(bitstring)`

Description	Returns the number of bits in the bitstring.
Example	`bit_length('1101011'::BIT)`
Result	7

`bit_position(substring, bitstring)`

Description	Returns first starting index of the specified substring within bits, or zero if it's not present. The first (leftmost) bit is indexed 1
Example	`bit_position('010'::BIT, '1110101'::BIT)`
Result	4

`bitstring(bitstring, length)`

Description	Returns a bitstring of determined length.
Example	`bitstring('1010'::BIT, 7)`
Result	`0001010`

`get_bit(bitstring, index)`

Description	Extracts the nth bit from bitstring; the first (leftmost) bit is indexed 0.
Example	`get_bit('0110010'::BIT, 2)`
Result	1

`length(bitstring)`

Description	Alias for `bit_length`.
Example	`length('1101011'::BIT)`
Result	7

`octet_length(bitstring)`

Description	Returns the number of bytes in the bitstring.
Example	`octet_length('1101011'::BIT)`
Result	`1`

`set_bit(bitstring, index, new_value)`

Description	Sets the nth bit in bitstring to newvalue; the first (leftmost) bit is indexed 0. Returns a new bitstring.
Example	`set_bit('0110010'::BIT, 2, 0)`
Result	`0100010`

Bitstring Aggregate Functions

These aggregate functions are available for `BIT` type.

Name	Description
`bit_and(arg)`	Returns the bitwise AND operation performed on all bitstrings in a given expression.
`bit_or(arg)`	Returns the bitwise OR operation performed on all bitstrings in a given expression.
`bit_xor(arg)`	Returns the bitwise XOR operation performed on all bitstrings in a given expression.
`bitstring_agg(arg)`	Returns a bitstring with bits set for each distinct position defined in `arg`.
`bitstring_agg(arg, min, max)`	Returns a bitstring with bits set for each distinct position defined in `arg`. All positions must be within the range [`min`, `max`] or an ”Out of Range Error” will be thrown.

`bit_and(arg)`

Description	Returns the bitwise AND operation performed on all bitstrings in a given expression.
Example	`bit_and(A)`

`bit_or(arg)`

Description	Returns the bitwise OR operation performed on all bitstrings in a given expression.
Example	`bit_or(A)`

`bit_xor(arg)`

Description	Returns the bitwise XOR operation performed on all bitstrings in a given expression.
Example	`bit_xor(A)`

`bitstring_agg(arg)`

Description	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.
Example	`bitstring_agg(A)`

> **Tip.** The combination of `bit_count` and `bitstring_agg` can be used as an alternative to `count(DISTINCT ...)`, with possible performance improvements in cases of low cardinality and dense values.

`bitstring_agg(arg, min, max)`

Description	Returns a bitstring with bits set for each distinct position defined in `arg`. All positions must be within the range [`min`, `max`] or an "Out of Range Error" will be thrown.
Example	`bitstring_agg(A, 1, 42)`

Blob Functions

This section describes functions and operators for examining and manipulating blob values.

Name	Description
`blob || blob`	Blob concatenation.
`decode(blob)`	Converts BLOB to VARCHAR. Fails if blob is not valid UTF-8.
`encode(string)`	Converts VARCHAR to BLOB. Converts UTF-8 characters into literal encoding.
`octet_length(blob)`	Number of bytes in BLOB.
`read_blob(source)`	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a BLOB. See the `read_blob` guide for more details.

`blob || blob`

Description	BLOB concatenation.
Example	`'\xAA'::BLOB || '\xBB'::BLOB`
Result	`\xAA\xBB`

`decode(blob)`

Description	Convert BLOB to VARCHAR. Fails if blob is not valid UTF-8.
Example	`decode('\xC3\xBC'::BLOB)`
Result	ü

`encode(string)`

Description	Convert VARCHAR to BLOB. Converts UTF-8 characters into literal encoding.
Example	`encode('my_string_with_ü')`
Result	`my_string_with_\xC3\xBC`

`octet_length(blob)`

Description	Number of bytes in VARCHAR.
Example	`octet_length('\xAA\xBB'::BLOB)`
Result	2

`read_blob(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a BLOB. See the `read_blob` guide for more details.
Example	`read_blob('hello.bin')`
Result	`hello\x0A`

Date Format Functions

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 a COPY statement:

```
COPY dates FROM 'test.csv' (DATEFORMAT '%d/%m/%Y', TIMESTAMPFORMAT '%A, %-d %B %Y - %I:%M:%S %p');
```

In a `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, ...
%b	Abbreviated month name.	Jan, Feb, ..., Dec
%B	Full month name.	January, February, ...
%c	ISO date and time representation	1992-03-02 10:30:20
%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
%f	Microsecond as a decimal number, zero-padded on the left.	000000 - 999999
%g	Millisecond as a decimal number, zero-padded on the left.	000 - 999
%G	ISO 8601 year with century representing the year that contains the greater part of the ISO week (see %V).	0001, 0002, ..., 2013, 2014, ..., 9998, 9999
%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
%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
%m	Month as a zero-padded decimal number.	01, 02, ..., 12
%-m	Month as a decimal number.	1, 2, ..., 12
%M	Minute as a zero-padded decimal number.	00, 01, ..., 59
%-M	Minute as a decimal number.	0, 1, ..., 59

Specifier	Description	Example
%n	Nanosecond as a decimal number, zero-padded on the left.	000000000 - 999999999
%p	Locale's AM or PM.	AM, PM
%S	Second as a zero-padded decimal number.	00, 01, ..., 59
%-S	Second as a decimal number.	0, 1, ..., 59
%u	ISO 8601 weekday as a decimal number where 1 is Monday.	1, 2, ..., 7
%U	Week number of the year. Week 01 starts on the first Sunday of the year, so there can be week 00. Note that this is not compliant with the week date standard in ISO-8601.	00, 01, ..., 53
%V	ISO 8601 week as a decimal number with Monday as the first day of the week. Week 01 is the week containing Jan 4.	01, ..., 53
%w	Weekday as a decimal number.	0, 1, ..., 6
%W	Week number of the year. Week 01 starts on the first Monday of the year, so there can be week 00. Note that this is not compliant with the week date standard in ISO-8601.	00, 01, ..., 53
%x	ISO date representation	1992-03-02
%X	ISO time representation	10:30:20
%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.
%z	Time offset from UTC in the form ±HH:MM, ±HHMM, or ±HH.	-0700
%Z	Time zone name.	Europe/Amsterdam
%%	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`
+	addition of a variable INTERVAL	`SELECT DATE '1992-03-22' + INTERVAL (d.days) DAY FROM (VALUES (5), (11)) AS d(days)`	`1992-03-27` and `1992-04-02`
-	subtraction of DATEs	`DATE '1992-03-27' - DATE '1992-03-22'`	`5`

Operator	Description	Example	Result
-	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 (d.days) DAY FROM (VALUES (5), (11)) AS d(days)	1992-03-22 and 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.

Name	Description
current_date	Current date (at start of current transaction).
date_add(date, interval)	Add the interval to the date.
date_diff(part, startdate, enddate)	The number of partition boundaries between the dates.
date_part(part, date)	Get the subfield (equivalent to extract).
date_sub(part, startdate, enddate)	The number of complete partitions between the dates.
date_trunc(part, date)	Truncate to specified precision.
datediff(part, startdate, enddate)	The number of partition boundaries between the dates. Alias of date_diff.
datepart(part, date)	Get the subfield (equivalent to extract). Alias of date_part.
datesub(part, startdate, enddate)	The number of complete partitions between the dates. Alias of date_sub.
datetrunc(part, date)	Truncate to specified precision. Alias of date_trunc.
dayname(date)	The (English) name of the weekday.
extract(part from date)	Get subfield from a date.
greatest(date, date)	The later of two dates.
isfinite(date)	Returns true if the date is finite, false otherwise.
isinf(date)	Returns true if the date is infinite, false otherwise.
last_day(date)	The last day of the corresponding month in the date.
least(date, date)	The earlier of two dates.
make_date(year, month, day)	The date for the given parts.
monthname(date)	The (English) name of the month.
strftime(date, format)	Converts a date to a string according to the format string.
time_bucket(bucket_width, date[, offset])	Truncate date by the specified interval bucket_width. Buckets are offset by offset interval.

Name	Description
`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.
today()	Current date (start of current transaction).

current_date

Description	Current date (at start of current transaction).
Example	`current_date`
Result	`2022-10-08`

date_add(date, interval)

Description	Add the interval to the date.
Example	`date_add(DATE '1992-09-15', INTERVAL 2 MONTH)`
Result	`1992-11-15`

date_diff(part, startdate, enddate)

Description	The number of partition boundaries between the dates.
Example	`date_diff('month', DATE '1992-09-15', DATE '1992-11-14')`
Result	`2`

date_part(part, date)

Description	Get the subfield (equivalent to `extract`).
Example	`date_part('year', DATE '1992-09-20')`
Result	`1992`

date_sub(part, startdate, enddate)

Description	The number of complete partitions between the dates.
Example	`date_sub('month', DATE '1992-09-15', DATE '1992-11-14')`
Result	`1`

`date_trunc(part, date)`

Description	Truncate to specified precision.
Example	`date_trunc('month', DATE '1992-03-07')`
Result	`1992-03-01`

`datediff(part, startdate, enddate)`

Description	The number of partition boundaries between the dates.
Example	`datediff('month', DATE '1992-09-15', DATE '1992-11-14')`
Result	`2`
Alias	`date_diff`.

`datepart(part, date)`

Description	Get the subfield (equivalent to `extract`).
Example	`datepart('year', DATE '1992-09-20')`
Result	`1992`
Alias	`date_part`.

`datesub(part, startdate, enddate)`

Description	The number of complete partitions between the dates.
Example	`datesub('month', DATE '1992-09-15', DATE '1992-11-14')`
Result	`1`
Alias	`date_sub`.

`datetrunc(part, date)`

Description	Truncate to specified precision.
Example	`datetrunc('month', DATE '1992-03-07')`
Result	`1992-03-01`
Alias	`date_trunc`.

`dayname(date)`

Description	The (English) name of the weekday.

Example	`dayname(DATE '1992-09-20')`
Result	Sunday

`extract(part from date)`

Description	Get subfield from a date.
Example	`extract('year' FROM DATE '1992-09-20')`
Result	`1992`

`greatest(date, date)`

Description	The later of two dates.
Example	`greatest(DATE '1992-09-20', DATE '1992-03-07')`
Result	`1992-09-20`

`isfinite(date)`

Description	Returns `true` if the date is finite, false otherwise.
Example	`isfinite(DATE '1992-03-07')`
Result	`true`

`isinf(date)`

Description	Returns `true` if the date is infinite, false otherwise.
Example	`isinf(DATE '-infinity')`
Result	`true`

`last_day(date)`

Description	The last day of the corresponding month in the date.
Example	`last_day(DATE '1992-09-20')`
Result	`1992-09-30`

least(date, date)

Description	The earlier of two dates.
Example	`least(DATE '1992-09-20', DATE '1992-03-07')`
Result	`1992-03-07`

make_date(year, month, day)

Description	The date for the given parts.
Example	`make_date(1992, 9, 20)`
Result	`1992-09-20`

monthname(date)

Description	The (English) name of the month.
Example	`monthname(DATE '1992-09-20')`
Result	`September`

strftime(date, format)

Description	Converts a date to a string according to the format string.
Example	`strftime(date '1992-01-01', '%a, %-d %B %Y')`
Result	`Wed, 1 January 1992`

time_bucket(bucket_width, date[, offset])

Description	Truncate `date` by the specified interval `bucket_width`. Buckets are offset by `offset` interval.
Example	`time_bucket(INTERVAL '2 months', DATE '1992-04-20', INTERVAL '1 month')`
Result	`1992-04-01`

time_bucket(bucket_width, date[, origin])

Description	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.

Example	`time_bucket(INTERVAL '2 weeks', DATE '1992-04-20', DATE '1992-04-01')`
Result	`1992-04-15`

`today()`

Description	Current date (start of current transaction).
Example	`today()`
Result	`2022-10-08`

Date Part Extraction Functions

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 Part Functions

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.

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`.

Part Specifiers 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`
`'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`

Specifier	Description	Synonyms	Example
`'second'`	Seconds	`'sec'`, `'seconds'`, `'secs'`, `'s'`	`44`
`'year'`	Gregorian year	`'yr'`, `'y'`, `'years'`, `'yrs'`	`2021`

Part Specifiers Only Usable as Date Part Specifiers

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`
`'timezone_hour'`	Time zone offset hour portion		`0`
`'timezone_minute'`	Time zone offset minute portion		`0`
`'timezone'`	Time zone offset in seconds		`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:

Name	Description
`century(date)`	Century.
`day(date)`	Day.
`dayofmonth(date)`	Day (synonym).
`dayofweek(date)`	Numeric weekday (Sunday = 0, Saturday = 6).
`dayofyear(date)`	Day of the year (starts from 1, i.e., January 1 = 1).
`decade(date)`	Decade (year / 10).
`epoch(date)`	Seconds since 1970-01-01.

Name	Description
`era(date)`	Calendar era.
`hour(date)`	Hours.
`isodow(date)`	Numeric ISO weekday (Monday = 1, Sunday = 7).
`isoyear(date)`	ISO Year number (Starts on Monday of week containing Jan 4th).
`microsecond(date)`	Sub-minute microseconds.
`millennium(date)`	Millennium.
`millisecond(date)`	Sub-minute milliseconds.
`minute(date)`	Minutes.
`month(date)`	Month.
`quarter(date)`	Quarter.
`second(date)`	Seconds.
`timezone_hour(date)`	Time zone offset hour portion.
`timezone_minute(date)`	Time zone offset minutes portion.
`timezone(date)`	Time Zone offset in minutes.
`week(date)`	ISO Week.
`weekday(date)`	Numeric weekday synonym (Sunday = 0, Saturday = 6).
`weekofyear(date)`	ISO Week (synonym).
`year(date)`	Year.
`yearweek(date)`	`BIGINT` of combined ISO Year number and 2-digit version of ISO Week number.

`century(date)`

Description	Century.
Example	`century(date '1992-02-15')`
Result	`20`

`day(date)`

Description	Day.
Example	`day(date '1992-02-15')`
Result	`15`

`dayofmonth(date)`

Description	Day (synonym).
Example	`dayofmonth(date '1992-02-15')`
Result	`15`

dayofweek(date)

Description	Numeric weekday (Sunday = 0, Saturday = 6).
Example	`dayofweek(date '1992-02-15')`
Result	`6`

dayofyear(date)

Description	Day of the year (starts from 1, i.e., January 1 = 1).
Example	`dayofyear(date '1992-02-15')`
Result	`46`

decade(date)

Description	Decade (year / 10).
Example	`decade(date '1992-02-15')`
Result	`199`

epoch(date)

Description	Seconds since 1970-01-01.
Example	`epoch(date '1992-02-15')`
Result	`698112000`

era(date)

Description	Calendar era.
Example	`era(date '0044-03-15 (BC)')`
Result	`0`

hour(date)

Description	Hours.
Example	`hour(timestamp '2021-08-03 11:59:44.123456')`
Result	`11`

`isodow(date)`

Description	Numeric ISO weekday (Monday = 1, Sunday = 7).
Example	`isodow(date '1992-02-15')`
Result	6

`isoyear(date)`

Description	ISO Year number (Starts on Monday of week containing Jan 4th).
Example	`isoyear(date '2022-01-01')`
Result	2021

`microsecond(date)`

Description	Sub-minute microseconds.
Example	`microsecond(timestamp '2021-08-03 11:59:44.123456')`
Result	44123456

`millennium(date)`

Description	Millennium.
Example	`millennium(date '1992-02-15')`
Result	2

`millisecond(date)`

Description	Sub-minute milliseconds.
Example	`millisecond(timestamp '2021-08-03 11:59:44.123456')`
Result	44123

`minute(date)`

Description	Minutes.
Example	`minute(timestamp '2021-08-03 11:59:44.123456')`
Result	59

`month(date)`

Description	Month.
Example	`month(date '1992-02-15')`
Result	2

`quarter(date)`

Description	Quarter.
Example	`quarter(date '1992-02-15')`
Result	1

`second(date)`

Description	Seconds.
Example	`second(timestamp '2021-08-03 11:59:44.123456')`
Result	44

`timezone_hour(date)`

Description	Time zone offset hour portion.
Example	`timezone_hour(date '1992-02-15')`
Result	0

`timezone_minute(date)`

Description	Time zone offset minutes portion.
Example	`timezone_minute(date '1992-02-15')`
Result	0

`timezone(date)`

Description	Time Zone offset in minutes.
Example	`timezone(date '1992-02-15')`
Result	0

`week(date)`

Description	ISO Week.
Example	`week(date '1992-02-15')`
Result	7

`weekday(date)`

Description	Numeric weekday synonym (Sunday = 0, Saturday = 6).
Example	`weekday(date '1992-02-15')`
Result	6

`weekofyear(date)`

Description	ISO Week (synonym).
Example	`weekofyear(date '1992-02-15')`
Result	7

`year(date)`

Description	Year.
Example	`year(date '1992-02-15')`
Result	1992

`yearweek(date)`

Description	`BIGINT` of combined ISO Year number and 2-digit version of ISO Week number.
Example	`yearweek(date '1992-02-15')`
Result	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.

Name	Description
enum_code(enum_value)	Returns the numeric value backing the given enum value.
enum_first(enum)	Returns the first value of the input enum type.
enum_last(enum)	Returns the last value of the input enum type.
enum_range(enum)	Returns all values of the input enum type as an array.
enum_range_boundary(enum, enum)	Returns the range between the two given enum values as an array.

enum_code(enum_value)

Description	Returns the numeric value backing the given enum value.
Example	`enum_code('happy'::mood)`
Result	2

enum_first(enum)

Description	Returns the first value of the input enum type.
Example	`enum_first(NULL::mood)`
Result	sad

enum_last(enum)

Description	Returns the last value of the input enum type.
Example	`enum_last(NULL::mood)`
Result	anxious

enum_range(enum)

Description	Returns all values of the input enum type as an array.
Example	`enum_range(NULL::mood)`
Result	`[sad, ok, happy, anxious]`

enum_range_boundary(enum, enum)

Description	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.

Example	`enum_range_boundary(NULL, 'happy'::mood)`
Result	`[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.

Name	Description
`date_part(part, interval)`	Get subfield (equivalent to `extract`).
`datepart(part, interval)`	Alias of date_part. Get subfield (equivalent to `extract`).
`extract(part FROM interval)`	Get subfield from an interval.
`epoch(interval)`	Get total number of seconds in interval.
`to_centuries(integer)`	Construct a century interval.
`to_days(integer)`	Construct a day interval.
`to_decades(integer)`	Construct a decade interval.
`to_hours(integer)`	Construct a hour interval.
`to_microseconds(integer)`	Construct a microsecond interval.
`to_millennia(integer)`	Construct a millenium interval.
`to_milliseconds(integer)`	Construct a millisecond interval.

Name	Description
to_minutes(integer)	Construct a minute interval.
to_months(integer)	Construct a month interval.
to_seconds(integer)	Construct a second interval.
to_weeks(integer)	Construct a week interval.
to_years(integer)	Construct a year interval.

> Only the documented date parts are defined for intervals.

date_part(part, interval)

Description	Get subfield (equivalent to extract).
Example	date_part('year', INTERVAL '14 months')
Result	1

datepart(part, interval)

Description	Alias of date_part. Get subfield (equivalent to extract).
Example	datepart('year', INTERVAL '14 months')
Result	1

extract(part FROM interval)

Description	Get subfield from an interval.
Example	extract('month' FROM INTERVAL '14 months')
Result	2

epoch(interval)

Description	Get total number of seconds in interval.
Example	epoch(INTERVAL 5 HOUR)
Result	18000.0

to_centuries(integer)

Description	Construct a century interval.
Example	to_centuries(5)

Result	`INTERVAL 500 YEAR`

to_days(integer)

Description	Construct a day interval.
Example	`to_days(5)`
Result	`INTERVAL 5 DAY`

to_decades(integer)

Description	Construct a decade interval.
Example	`to_decades(5)`
Result	`INTERVAL 50 YEAR`

to_hours(integer)

Description	Construct a hour interval.
Example	`to_hours(5)`
Result	`INTERVAL 5 HOUR`

to_microseconds(integer)

Description	Construct a microsecond interval.
Example	`to_microseconds(5)`
Result	`INTERVAL 5 MICROSECOND`

to_millennia(integer)

Description	Construct a millenium interval.
Example	`to_millennia(5)`
Result	`INTERVAL 5000 YEAR`

to_milliseconds(integer)

Description	Construct a millisecond interval.
Example	`to_milliseconds(5)`

Result	`INTERVAL 5 MILLISECOND`

`to_minutes(integer)`

Description	Construct a minute interval.
Example	`to_minutes(5)`
Result	`INTERVAL 5 MINUTE`

`to_months(integer)`

Description	Construct a month interval.
Example	`to_months(5)`
Result	`INTERVAL 5 MONTH`

`to_seconds(integer)`

Description	Construct a second interval.
Example	`to_seconds(5)`
Result	`INTERVAL 5 SECOND`

`to_weeks(integer)`

Description	Construct a week interval.
Example	`to_weeks(5)`
Result	`INTERVAL 35 DAY`

`to_years(integer)`

Description	Construct a year interval.
Example	`to_years(5)`
Result	`INTERVAL 5 YEAR`

Lambda Functions

Lambda functions enable the use of more complex and flexible expressions in queries. DuckDB supports several scalar functions that accept lambda functions as parameters in the form `(parameter1, parameter2, ...) -> expression`. If the lambda function has only one parameter, then the parentheses can be omitted. The parameters can have any names. For example, the following are all valid lambda functions:

- `param -> param > 1`
- `s -> contains(concat(s, 'DB'), 'duck')`
- `(x, y) -> x + y`

Scalar Functions That Accept Lambda Functions

Name	Description
`list_transform(list, lambda)`	Returns a list that is the result of applying the lambda function to each element of the input list.
`list_filter(list, lambda)`	Constructs a list from those elements of the input list for which the lambda function returns `true`.
`list_reduce(list, lambda)`	Reduces all elements of the input list into a single value by executing the lambda function on a running result and the next list element.

`list_transform(list, lambda)`

Description	Returns a list that is the result of applying the lambda function to each element of the input list. For more information, see Transform.
Example	`list_transform([4, 5, 6], x -> x + 1)`
Result	`[5, 6, 7]`
Aliases	`array_transform`, `apply`, `list_apply`, `array_apply`

`list_filter(list, lambda)`

Description	Constructs a list from those elements of the input list for which the lambda function returns `true`. For more information, see Filter.
Example	`list_filter([4, 5, 6], x -> x > 4)`
Result	`[5, 6]`
Aliases	`array_filter`, `filter`

`list_reduce(list, lambda)`

Description	Reduces all elements of the input list into a single value by executing the lambda function on a running result and the next list element. For more information, see Reduce.
Example	`list_reduce([4, 5, 6], (x, y) -> x + y)`
Result	`15`
Aliases	`array_reduce`, `reduce`

Nesting

All scalar 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]
```

Nested lambda function to add each element of the first list to the sum of the second list:

```
SELECT list_transform(
        [1, 2, 3],
        x -> list_reduce([4, 5, 6], (a, b) -> a + b) + x
    );
```

```
[16, 17, 18]
```

Scoping

Lambda functions confirm to scoping rules in the following order:

- inner lambda parameters
- outer lambda parameters
- column names
- macro parameters

```
CREATE TABLE tbl (x INTEGER);
INSERT INTO tbl VALUES (10);
SELECT apply([1, 2], x -> apply([4], x -> x + tbl.x)[1] + x) FROM tbl;
```

```
[15, 16]
```

Indexes as Parameters

All lambda functions accept an optional extra parameter that represents the index of the current element. This is always the last parameter of the lambda function, and is 1-based (i.e., the first element has index 1).

Get all elements that are larger than their index:

```
SELECT list_filter([1, 3, 1, 5], (x, i) -> x > i);
```

```
[3, 5]
```

Transform

Signature: `list_transform(list, lambda)`

Description:
`list_transform` returns a list that is the result of applying the lambda function to each element of the input list.

Aliases:

- `array_transform`
- `apply`
- `list_apply`
- `array_apply`

Number of parameters excluding indexes: 1

Return type: Defined by the return type of the lambda function

Examples:
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'], s -> concat(s, '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

Signature: `list_filter(list, lambda)`

Description:
Constructs a list from those elements of the input list for which the lambda function returns `true`. DuckDB must be able to cast the lambda function's return type to `BOOL`.

Aliases:

- `array_filter`
- `filter`

Number of parameters excluding indexes: 1

Return type: The same type as the input list

Examples:
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]
[]
```

Reduce

Signature: `list_reduce(list, lambda)`

Description:
The scalar function returns a single value that is the result of applying the lambda function to each element of the input list. Starting with the first element and then repeatedly applying the lambda function to the result of the previous application and the next element of the list. The list must have at least one element.

Aliases:

- `array_reduce`
- `reduce`

Number of parameters excluding indexes: 2

Return type: The type of the input list's elements

Examples:
Sum of all list elements:

```
SELECT list_reduce([1, 2, 3, 4], (x, y) -> x + y);
```

```
10
```

Only add up list elements if they are greater than 2:

```
SELECT list_reduce(list_filter([1, 2, 3, 4], x -> x > 2), (x, y) -> x + y);
```

```
7
```

Concat all list elements:

```
SELECT list_reduce(['DuckDB', 'is', 'awesome'], (x, y) -> concat(x, ' ', y));
```

```
DuckDB is awesome
```

Nested Functions

This section describes functions and operators for examining and manipulating nested values. There are five nested data types: `ARRAY`, `LIST`, `MAP`, `STRUCT`, and `UNION`.

List Functions

Name	Description
`list[index]`	Bracket notation serves as an alias for `list_extract`.
`list[begin:end]`	Bracket notation with colon is an alias for `list_slice`.
`list[begin:end:step]`	`list_slice` in bracket notation with an added `step` feature.
`array_pop_back(list)`	Returns the list without the last element.
`array_pop_front(list)`	Returns the list without the first element.
`flatten(list_of_lists)`	Concatenate a list of lists into a single list. This only flattens one level of the list (see examples).
`len(list)`	Return the length of the list.
`list_aggregate(list, name)`	Executes the aggregate function `name` on the elements of `list`. See the List Aggregates section for more details.
`list_any_value(list)`	Returns the first non-null value in the list.
`list_append(list, element)`	Appends `element` to `list`.
`list_concat(list1, list2)`	Concatenates two lists.
`list_contains(list, element)`	Returns true if the list contains the element.
`list_cosine_similarity(list1, list2)`	Compute the cosine similarity between two lists.

Name	Description
`list_distance(list1, list2)`	Calculates the Euclidean distance between two points with coordinates given in two inputs lists of equal length.
`list_distinct(list)`	Removes all duplicates and `NULL` values from a list. Does not preserve the original order.
`list_dot_product(list1, list2)`	Computes the dot product of two same-sized lists of numbers.
`list_extract(list, index)`	Extract the `index`th (1-based) value from the list.
`list_filter(list, lambda)`	Constructs a list from those elements of the input list for which the lambda function returns true. See the Lambda Functions page for more details.
`list_grade_up(list)`	Works like sort, but the results are the indexes that correspond to the position in the original `list` instead of the actual values.
`list_has_all(list, sub-list)`	Returns true if all elements of sub-list exist in list.
`list_has_any(list1, list2)`	Returns true if any elements exist is both lists.
`list_intersect(list1, list2)`	Returns a list of all the elements that exist in both `l1` and `l2`, without duplicates.
`list_position(list, element)`	Returns the index of the element if the list contains the element.
`list_prepend(element, list)`	Prepends `element` to `list`.
`list_reduce(list, lambda)`	Returns a single value that is the result of applying the lambda function to each element of the input list. See the Lambda Functions page for more details.
`list_resize(list, size[, value])`	Resizes the list to contain `size` elements. Initializes new elements with `value` or `NULL` if `value` is not set.
`list_reverse_sort(list)`	Sorts the elements of the list in reverse order. See the Sorting Lists section for more details about the `NULL` sorting order.
`list_reverse(list)`	Reverses the list.
`list_select(value_list, index_list)`	Returns a list based on the elements selected by the `index_list`.
`list_slice(list, begin, end, step)`	`list_slice` with added `step` feature.
`list_slice(list, begin, end)`	Extract a sublist using slice conventions. Negative values are accepted. See slicing.
`list_sort(list)`	Sorts the elements of the list. See the Sorting Lists section for more details about the sorting order and the `NULL` sorting order.
`list_transform(list, lambda)`	Returns a list that is the result of applying the lambda function to each element of the input list. See the Lambda Functions page for more details.
`list_unique(list)`	Counts the unique elements of a list.
`list_value(any, ...)`	Create a `LIST` containing the argument values.
`list_where(value_list, mask_list)`	Returns a list with the `BOOLEAN`s in `mask_list` applied as a mask to the `value_list`.

Name	Description
`list_zip(list_1, list_2, ...[, truncate])`	Zips *k* `LIST`s to a new `LIST` whose length will be that of the longest list. Its elements are structs of *k* elements from each list `list_1`, ..., `list_k`, missing elements are replaced with `NULL`. If `truncate` is set, all lists are truncated to the smallest list length.
`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.

`list[index]`

Description	Bracket notation serves as an alias for `list_extract`.
Example	`[4, 5, 6][3]`
Result	6
Alias	`list_extract`

`list[begin:end]`

Description	Bracket notation with colon is an alias for `list_slice`.
Example	`[4, 5, 6][2:3]`
Result	`[5, 6]`
Alias	`list_slice`

`list[begin:end:step]`

Description	`list_slice` in bracket notation with an added `step` feature.
Example	`[4, 5, 6][:-:2]`
Result	`[4, 6]`
Alias	`list_slice`

`array_pop_back(list)`

Description	Returns the list without the last element.
Example	`array_pop_back(l)`
Result	`[4, 5]`

`array_pop_front(list)`

Description	Returns the list without the first element.

Example	`array_pop_front(l)`
Result	`[5, 6]`

`flatten(list_of_lists)`

Description	Concatenate a list of lists into a single list. This only flattens one level of the list (see examples).
Example	`flatten([[1, 2], [3, 4]])`
Result	`[1, 2, 3, 4]`

`len(list)`

Description	Return the length of the list.
Example	`len([1, 2, 3])`
Result	`3`
Alias	`array_length`

`list_aggregate(list, name)`

Description	Executes the aggregate function `name` on the elements of `list`. See the List Aggregates section for more details.
Example	`list_aggregate([1, 2, NULL], 'min')`
Result	`1`
Aliases	`list_aggr`, `aggregate`, `array_aggregate`, `array_aggr`

`list_any_value(list)`

Description	Returns the first non-null value in the list.
Example	`list_any_value([NULL, -3])`
Result	`-3`

`list_append(list, element)`

Description	Appends `element` to `list`.
Example	`list_append([2, 3], 4)`
Result	`[2, 3, 4]`
Aliases	`array_append`, `array_push_back`

`list_concat(list1, list2)`

Description	Concatenates two lists.
Example	`list_concat([2, 3], [4, 5, 6])`
Result	`[2, 3, 4, 5, 6]`
Aliases	`list_cat`, `array_concat`, `array_cat`

`list_contains(list, element)`

Description	Returns true if the list contains the element.
Example	`list_contains([1, 2, NULL], 1)`
Result	`true`
Aliases	`list_has`, `array_contains`, `array_has`

`list_cosine_similarity(list1, list2)`

Description	Compute the cosine similarity between two lists.
Example	`list_cosine_similarity([1, 2, 3], [1, 2, 5])`
Result	`0.9759000729485332`

`list_distance(list1, list2)`

Description	Calculates the Euclidean distance between two points with coordinates given in two inputs lists of equal length.
Example	`list_distance([1, 2, 3], [1, 2, 5])`
Result	`2.0`

`list_distinct(list)`

Description	Removes all duplicates and `NULL` values from a list. Does not preserve the original order.
Example	`list_distinct([1, 1, NULL, -3, 1, 5])`
Result	`[1, 5, -3]`
Alias	`array_distinct`

`list_dot_product(list1, list2)`

Description	Computes the dot product of two same-sized lists of numbers.

Example	`list_dot_product([1, 2, 3], [1, 2, 5])`
Result	`20.0`
Alias	`list_inner_product`

`list_extract(list, index)`

Description	Extract the `index`th (1-based) value from the list.
Example	`list_extract([4, 5, 6], 3)`
Result	`6`
Aliases	`list_element`, `array_extract`

`list_filter(list, lambda)`

Description	Constructs a list from those elements of the input list for which the lambda function returns true. See the Lambda Functions page for more details.
Example	`list_filter([4, 5, 6], x -> x > 4)`
Result	`[5, 6]`
Aliases	`array_filter`, `filter`

`list_grade_up(list)`

Description	Works like sort, but the results are the indexes that correspond to the position in the original `list` instead of the actual values.
Example	`list_grade_up([30, 10, 40, 20])`
Result	`[2, 4, 1, 3]`
Alias	`array_grade_up`

`list_has_all(list, sub-list)`

Description	Returns true if all elements of sub-list exist in list.
Example	`list_has_all([4, 5, 6], [4, 6])`
Result	`true`
Alias	`array_has_all`

`list_has_any(list1, list2)`

Description	Returns true if any elements exist is both lists.

Example	`list_has_any([1, 2, 3], [2, 3, 4])`
Result	`true`
Alias	`array_has_any`

`list_intersect(list1, list2)`

Description	Returns a list of all the elements that exist in both `l1` and `l2`, without duplicates.
Example	`list_intersect([1, 2, 3], [2, 3, 4])`
Result	`[2, 3]`
Alias	`array_intersect`

`list_position(list, element)`

Description	Returns the index of the element if the list contains the element.
Example	`list_contains([1, 2, NULL], 2)`
Result	2
Aliases	`list_indexof`, `array_position`, `array_indexof`

`list_prepend(element, list)`

Description	Prepends `element` to `list`.
Example	`list_prepend(3, [4, 5, 6])`
Result	`[3, 4, 5, 6]`
Aliases	`array_prepend`, `array_push_front`

`list_reduce(list, lambda)`

Description	Returns a single value that is the result of applying the lambda function to each element of the input list. See the Lambda Functions page for more details.
Example	`list_reduce([4, 5, 6], (x, y) -> x + y)`
Result	15
Aliases	`array_reduce`, `reduce`

`list_resize(list, size[, value])`

Description	Resizes the list to contain `size` elements. Initializes new elements with `value` or `NULL` if `value` is not set.

Example	`list_resize([1, 2, 3], 5, 0)`
Result	`[1, 2, 3, 0, 0]`
Alias	`array_resize`

`list_reverse_sort(list)`

Description	Sorts the elements of the list in reverse order. See the Sorting Lists section for more details about the `NULL` sorting order.
Example	`list_reverse_sort([3, 6, 1, 2])`
Result	`[6, 3, 2, 1]`
Alias	`array_reverse_sort`

`list_reverse(list)`

Description	Reverses the list.
Example	`list_reverse([3, 6, 1, 2])`
Result	`[2, 1, 6, 3]`
Alias	`array_reverse`

`list_select(value_list, index_list)`

Description	Returns a list based on the elements selected by the `index_list`.
Example	`list_select([10, 20, 30, 40], [1, 4])`
Result	`[10, 40]`
Alias	`array_select`

`list_slice(list, begin, end, step)`

Description	`list_slice` with added `step` feature.
Example	`list_slice([4, 5, 6], 1, 3, 2)`
Result	`[4, 6]`
Alias	`array_slice`

`list_slice(list, begin, end)`

Description	Extract a sublist using slice conventions. Negative values are accepted. See slicing.
Example	`list_slice([4, 5, 6], 2, 3)`

Result	`[5, 6]`
Alias	`array_slice`

`list_sort(list)`

Description	Sorts the elements of the list. See the Sorting Lists section for more details about the sorting order and the `NULL` sorting order.
Example	`list_sort([3, 6, 1, 2])`
Result	`[1, 2, 3, 6]`
Alias	`array_sort`

`list_transform(list, lambda)`

Description	Returns a list that is the result of applying the lambda function to each element of the input list. See the Lambda Functions page for more details.
Example	`list_transform([4, 5, 6], x -> x + 1)`
Result	`[5, 6, 7]`
Aliases	`array_transform`, `apply`, `list_apply`, `array_apply`

`list_unique(list)`

Description	Counts the unique elements of a list.
Example	`list_unique([1, 1, NULL, -3, 1, 5])`
Result	`3`
Alias	`array_unique`

`list_value(any, ...)`

Description	Create a `LIST` containing the argument values.
Example	`list_value(4, 5, 6)`
Result	`[4, 5, 6]`
Alias	`list_pack`

`list_where(value_list, mask_list)`

Description	Returns a list with the `BOOLEAN`s in `mask_list` applied as a mask to the `value_list`.
Example	`list_where([10, 20, 30, 40], [true, false, false, true])`

Result	`[10, 40]`
Alias	`array_where`

`list_zip(list1, list2, ...)`

Description	Zips *k* `LIST`s to a new `LIST` whose length will be that of the longest list. Its elements are structs of *k* elements from each list `list_1`, ..., `list_k`, missing elements are replaced with `NULL`. If `truncate` is set, all lists are truncated to the smallest list length.
Example	`list_zip([1, 2], [3, 4], [5, 6])`
Result	`[(1, 3, 5), (2, 4, 6)]`
Alias	`array_zip`

`unnest(list)`

Description	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.
Example	`unnest([1, 2, 3])`
Result	1, 2, 3

List Operators

The following operators are supported for lists:

Operator	Description	Example	Result
`&&`	Alias for `list_has_any`.	`[1, 2, 3, 4, 5] && [2, 5, 5, 6]`	`true`
`@>`	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

Name	Description
`struct.entry`	Dot notation that serves as an alias for `struct_extract` from named STRUCTs.
`struct[entry]`	Bracket notation that serves as an alias for `struct_extract` from named STRUCTs.
`struct[idx]`	Bracket notation that serves as an alias for `struct_extract` from unnamed STRUCTs (tuples), using an index (1-based).
`row(any, ...)`	Create an unnamed STRUCT (tuple) containing the argument values.
`struct_extract(struct, 'entry')`	Extract the named entry from the STRUCT.
`struct_extract(struct, idx)`	Extract the entry from an unnamed STRUCT (tuple) using an index (1-based).
`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_pack(name := any, ...)`	Create a STRUCT containing the argument values. The entry name will be the bound variable name.

`struct.entry`

Description	Dot notation that serves as an alias for `struct_extract` from named STRUCTs.
Example	`({'i': 3, 's': 'string'}).i`
Result	3

`struct[entry]`

Description	Bracket notation that serves as an alias for `struct_extract` from named STRUCTs.
Example	`({'i': 3, 's': 'string'})['i']`
Result	3

`struct[idx]`

Description	Bracket notation that serves as an alias for `struct_extract` from unnamed STRUCTs (tuples), using an index (1-based).
Example	`(row(42, 84))[1]`
Result	`42`

`row(any, ...)`

Description	Create an unnamed STRUCT (tuple) containing the argument values.
Example	`row(i, i % 4, i / 4)`
Result	`(10, 2, 2.5)`

`struct_extract(struct, 'entry')`

Description	Extract the named entry from the STRUCT.
Example	`struct_extract({'i': 3, 'v2': 3, 'v3': 0}, 'i')`
Result	`3`

`struct_extract(struct, idx)`

Description	Extract the entry from an unnamed STRUCT (tuple) using an index (1-based).
Example	`struct_extract(row(42, 84), 1)`
Result	`42`

`struct_insert(struct, name := any, ...)`

Description	Add field(s)/value(s) to an existing STRUCT with the argument values. The entry name(s) will be the bound variable name(s).
Example	`struct_insert({'a': 1}, b := 2)`
Result	`{'a': 1, 'b': 2}`

`struct_pack(name := any, ...)`

Description	Create a STRUCT containing the argument values. The entry name will be the bound variable name.
Example	`struct_pack(i := 4, s := 'string')`
Result	`{'i': 4, 's': string}`

Map Functions

Name	Description
`cardinality(map)`	Return the size of the map (or the number of entries in the map).
`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.
`map_entries(map)`	Return a list of struct(k, v) for each key-value pair in the map.
`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_from_entries(STRUCT(k, v)[])`	Returns a map created from the entries of the array.
`map_keys(map)`	Return a list of all keys in the map.
`map_values(map)`	Return a list of all values in the map.
`map()`	Returns an empty map.
`map[entry]`	Alias for `element_at`.

`cardinality(map)`

Description	Return the size of the map (or the number of entries in the map).
Example	`cardinality(map([4, 2], ['a', 'b']))`
Result	2

`element_at(map, key)`

Description	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.
Example	`element_at(map([100, 5], [42, 43]), 100)`
Result	`[42]`

`map_entries(map)`

Description	Return a list of struct(k, v) for each key-value pair in the map.
Example	`map_entries(map([100, 5], [42, 43]))`
Result	`[{'key': 100, 'value': 42}, {'key': 5, 'value': 43}]`

map_extract(map, key)

Description	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.
Example	`map_extract(map([100, 5], [42, 43]), 100)`
Result	`[42]`

map_from_entries(STRUCT(k, v)[])

Description	Returns a map created from the entries of the array.
Example	`map_from_entries([{k: 5, v: 'val1'}, {k: 3, v: 'val2'}])`
Result	`{5=val1, 3=val2}`

map_keys(map)

Description	Return a list of all keys in the map.
Example	`map_keys(map([100, 5], [42,43]))`
Result	`[100, 5]`

map_values(map)

Description	Return a list of all values in the map.
Example	`map_values(map([100, 5], [42, 43]))`
Result	`[42, 43]`

map()

Description	Returns an empty map.
Example	`map()`
Result	`{}`

map[entry]

Description	Alias for `element_at`.
Example	`map([100, 5], ['a', 'b'])[100]`
Result	`[a]`

Union Functions

Name	Description
`union.tag`	Dot notation serves as an alias for `union_extract`.
`union_extract(union, 'tag')`	Extract the value with the named tags from the union. `NULL` if the tag is not currently selected.
`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_tag(union)`	Retrieve the currently selected tag of the union as an Enum.

`union.tag`

Description	Dot notation serves as an alias for `union_extract`.
Example	`(union_value(k := 'hello')).k`
Result	`string`

`union_extract(union, 'tag')`

Description	Extract the value with the named tags from the union. `NULL` if the tag is not currently selected.
Example	`union_extract(s, 'k')`
Result	`hello`

`union_value(tag := any)`

Description	Create a single member `UNION` containing the argument value. The tag of the value will be the bound variable name.
Example	`union_value(k := 'hello')`
Result	`'hello'::UNION(k VARCHAR)`

`union_tag(union)`

Description	Retrieve the currently selected tag of the union as an Enum.
Example	`union_tag(union_value(k := 'foo'))`
Result	`'k'`

Range Functions

DuckDB offers two range functions, `range(start, stop, step)` and `generate_series(start, stop, step)`, and their variants with default arguments for `stop` and `step`. The two functions' behavior differens regarding their `stop` argument. This is documented below.

range

The `range` function creates a list of values in the range between `start` and `stop`. The `start` parameter is inclusive, while the `stop` parameter is exclusive. The default value of `start` is 0 and the default value of `step` is 1.

Based on the number of arguments, the following variants of `range` exist.

range(stop)

```
SELECT range(5);
```

```
[0, 1, 2, 3, 4]
```

range(start, stop)

```
SELECT range(2, 5);
```

```
[2, 3, 4]
```

range(start, stop, step)

```
SELECT range(2, 5, 3);
```

```
[2]
```

generate_series

The `generate_series` function creates a list of values in the range between `start` and `stop`. Both the `start` and the `stop` parameters are inclusive. The default value of `start` is 0 and the default value of `step` is 1. Based on the number of arguments, the following variants of `generate_series` exist.

generate_series(stop)

```
SELECT generate_series(5);
```

```
[0, 1, 2, 3, 4, 5]
```

generate_series(start, stop)

```
SELECT generate_series(2, 5);
```

```
[2, 3, 4, 5]
```

generate_series(start, stop, step)

```
SELECT generate_series(2, 5, 3);
```

```
[2, 5]
```

Date Ranges

Date ranges are also supported for `TIMESTAMP` and `TIMESTAMP WITH TIME ZONE` values. Note that for these types, the `stop` and `step` arguments have to be specified explicitly (a default value is not provided).

`range` for Date Ranges

```
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

`generate_series` for Date Ranges

```
SELECT *
FROM generate_series(DATE '1992-01-01', DATE '1992-03-01', INTERVAL '1' MONTH);
```

generate_series
1992-01-01 00:00:00
1992-02-01 00:00:00
1992-03-01 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, step)`
- `array_slice(list, begin, end)`
- `array_slice(list, begin, end, step)`
- `list[begin:end]`
- `list[begin:end:step]`

The arguments are as follows:

- `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

Examples:

```
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_kurtosis_pop`, `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 LAST, 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 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

DuckDB supports lambda functions in the form `(parameter1, parameter2, ...) -> expression`. For details, see the lambda functions page.

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_subscripts(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. The `unnest` function 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	`4!`	`24`

Division and Modulo Operators

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`).

Supported Types

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.

Name	Description
`@(x)`	Absolute value. Parentheses are optional if `x` is a column name.
`abs(x)`	Absolute value.
`acos(x)`	Computes the arccosine of `x`.
`add(x, y)`	Alias for `x + y`.
`asin(x)`	Computes the arcsine of `x`.
`atan(x)`	Computes the arctangent of `x`.
`atan2(y, x)`	Computes the arctangent `(y, x)`.
`bit_count(x)`	Returns the number of bits that are set.
`cbrt(x)`	Returns the cube root of the number.
`ceil(x)`	Rounds the number up.
`ceiling(x)`	Rounds the number up. Alias of `ceil`.
`cos(x)`	Computes the cosine of `x`.
`cot(x)`	Computes the cotangent of `x`.
`degrees(x)`	Converts radians to degrees.
`divide(x, y)`	Alias for `x // y`.
`even(x)`	Round to next even number by rounding away from zero.
`exp(x)`	Computes `e ** x`.
`factorial(x)`	See `!` operator. Computes the product of the current integer and all integers below it.
`fdiv(x, y)`	Performs integer division (`x // y`) but returns a `DOUBLE` value.
`floor(x)`	Rounds the number down.
`fmod(x, y)`	Calculates the modulo value. Always returns a `DOUBLE` value.
`gamma(x)`	Interpolation of the factorial of `x - 1`. Fractional inputs are allowed.
`gcd(x, y)`	Computes the greatest common divisor of `x` and `y`.
`greatest_common_divisor(x, y)`	Computes the greatest common divisor of `x` and `y`.
`greatest(x1, x2, ...)`	Selects the largest value.
`isfinite(x)`	Returns true if the floating point value is finite, false otherwise.
`isinf(x)`	Returns true if the floating point value is infinite, false otherwise.
`isnan(x)`	Returns true if the floating point value is not a number, false otherwise.
`lcm(x, y)`	Computes the least common multiple of `x` and `y`.
`least_common_multiple(x, y)`	Computes the least common multiple of `x` and `y`.

Name	Description
`least(x1, x2, ...)`	Selects the smallest value.
`lgamma(x)`	Computes the log of the gamma function.
`ln(x)`	Computes the natural logarithm of x.
`log(x)`	Computes the base-10 logarithm of x.
`log10(x)`	Alias of log. Computes the base-10 logarithm of x.
`log2(x)`	Computes the base-2 log of x.
`multiply(x, y)`	Alias for x * y.
`nextafter(x, y)`	Return the next floating point value after x in the direction of y.
`pi()`	Returns the value of pi.
`pow(x, y)`	Computes x to the power of y.
`power(x, y)`	Alias of pow. computes x to the power of y.
`radians(x)`	Converts degrees to radians.
`random()`	Returns a random number between 0 and 1.
`round_even(v NUMERIC, s INTEGER)`	Alias of `roundbankers(v, s)`. Round to s decimal places using the *rounding half to even* rule. Values `s < 0` are allowed.
`round(v NUMERIC, s INTEGER)`	Round to s decimal places. Values `s < 0` are allowed.
`setseed(x)`	Sets the seed to be used for the random function.
`sign(x)`	Returns the sign of x as -1, 0 or 1.
`signbit(x)`	Returns whether the signbit is set or not.
`sin(x)`	Computes the sin of x.
`sqrt(x)`	Returns the square root of the number.
`subtract(x, y)`	Alias for x - y.
`tan(x)`	Computes the tangent of x.
`trunc(x)`	Truncates the number.
`xor(x)`	Bitwise XOR.

@(x)

Description	Absolute value. Parentheses are optional if x is a column name.
Example	`@(-17.4)`
Result	`17.4`
Alias	`abs`

abs(x)

Description	Absolute value.
Example	`abs(-17.4)`

Result	`17.4`
Alias	`@`

`acos(x)`

Description	Computes the arccosine of `x`.
Example	`acos(0.5)`
Result	`1.0471975511965976`

`add(x, y)`

Description	Alias for `x + y`.
Example	`add(2, 3)`
Result	`5`

`asin(x)`

Description	Computes the arcsine of `x`.
Example	`asin(0.5)`
Result	`0.5235987755982989`

`atan(x)`

Description	Computes the arctangent of `x`.
Example	`atan(0.5)`
Result	`0.4636476090008061`

`atan2(y, x)`

Description	Computes the arctangent (y, x).
Example	`atan2(0.5, 0.5)`
Result	`0.7853981633974483`

`bit_count(x)`

Description	Returns the number of bits that are set.
Example	`bit_count(31)`
Result	`5`

`cbrt(x)`

Description	Returns the cube root of the number.
Example	`cbrt(8)`
Result	`2`

`ceil(x)`

Description	Rounds the number up.
Example	`ceil(17.4)`
Result	`18`

`ceiling(x)`

Description	Rounds the number up. Alias of `ceil`.
Example	`ceiling(17.4)`
Result	`18`

`cos(x)`

Description	Computes the cosine of `x`.
Example	`cos(90)`
Result	`-0.4480736161291701`

`cot(x)`

Description	Computes the cotangent of `x`.
Example	`cot(0.5)`
Result	`1.830487721712452`

`degrees(x)`

Description	Converts radians to degrees.
Example	`degrees(pi())`
Result	`180`

`divide(x, y)`

Description	Alias for `x // y`.
Example	`divide(5, 2)`
Result	`2`

`even(x)`

Description	Round to next even number by rounding away from zero.
Example	`even(2.9)`
Result	`4`

`exp(x)`

Description	Computes `e ** x`.
Example	`exp(0.693)`
Result	`2`

`factorial(x)`

Description	See `!` operator. Computes the product of the current integer and all integers below it.
Example	`factorial(4)`
Result	`24`

`fdiv(x, y)`

Description	Performs integer division (`x // y`) but returns a `DOUBLE` value.
Example	`fdiv(5, 2)`
Result	`2.0`

`floor(x)`

Description	Rounds the number down.
Example	`floor(17.4)`
Result	`17`

`fmod(x, y)`

Description	Calculates the modulo value. Always returns a DOUBLE value.
Example	`fmod(5, 2)`
Result	`1.0`

`gamma(x)`

Description	Interpolation of the factorial of `x - 1`. Fractional inputs are allowed.
Example	`gamma(5.5)`
Result	`52.34277778455352`

`gcd(x, y)`

Description	Computes the greatest common divisor of x and y.
Example	`gcd(42, 57)`
Result	`3`

`greatest_common_divisor(x, y)`

Description	Computes the greatest common divisor of x and y.
Example	`greatest_common_divisor(42, 57)`
Result	`3`

`greatest(x1, x2, ...)`

Description	Selects the largest value.
Example	`greatest(3, 2, 4, 4)`
Result	`4`

`isfinite(x)`

Description	Returns true if the floating point value is finite, false otherwise.
Example	`isfinite(5.5)`
Result	`true`

`isinf(x)`

Description	Returns true if the floating point value is infinite, false otherwise.
Example	`isinf('Infinity'::float)`
Result	`true`

`isnan(x)`

Description	Returns true if the floating point value is not a number, false otherwise.
Example	`isnan('NaN'::float)`
Result	`true`

`lcm(x, y)`

Description	Computes the least common multiple of x and y.
Example	`lcm(42, 57)`
Result	`798`

`least_common_multiple(x, y)`

Description	Computes the least common multiple of x and y.
Example	`least_common_multiple(42, 57)`
Result	`798`

`least(x1, x2, ...)`

Description	Selects the smallest value.
Example	`least(3, 2, 4, 4)`
Result	`2`

`lgamma(x)`

Description	Computes the log of the gamma function.
Example	`lgamma(2)`
Result	`0`

`ln(x)`

Description	Computes the natural logarithm of `x`.
Example	`ln(2)`
Result	`0.693`

`log(x)`

Description	Computes the base-10 log of `x`.
Example	`log(100)`
Result	`2`

`log10(x)`

Description	Alias of `log`. Computes the base-10 log of `x`.
Example	`log10(1000)`
Result	`3`

`log2(x)`

Description	Computes the base-2 log of `x`.
Example	`log2(8)`
Result	`3`

`multiply(x, y)`

Description	Alias for `x * y`.
Example	`multiply(2, 3)`
Result	`6`

nextafter(x, y)

Description	Return the next floating point value after x in the direction of y.
Example	`nextafter(1::float, 2::float)`
Result	`1.0000001`

pi()

Description	Returns the value of pi.
Example	`pi()`
Result	`3.141592653589793`

pow(x, y)

Description	Computes x to the power of y.
Example	`pow(2, 3)`
Result	`8`

power(x, y)

Description	Alias of pow. computes x to the power of y.
Example	`power(2, 3)`
Result	`8`

radians(x)

Description	Converts degrees to radians.
Example	`radians(90)`
Result	`1.5707963267948966`

random()

Description	Returns a random number between 0 and 1.
Example	`random()`
Result	various

`round_even(v NUMERIC, s INTEGER)`

Description	Alias of `roundbankers(v, s)`. Round to `s` decimal places using the *rounding half to even* rule. Values `s < 0` are allowed.
Example	`round_even(24.5, 0)`
Result	`24.0`

`round(v NUMERIC, s INTEGER)`

Description	Round to `s` decimal places. Values `s < 0` are allowed.
Example	`round(42.4332, 2)`
Result	`42.43`

`setseed(x)`

Description	Sets the seed to be used for the random function.
Example	`setseed(0.42)`

`sign(x)`

Description	Returns the sign of `x` as -1, 0 or 1.
Example	`sign(-349)`
Result	`-1`

`signbit(x)`

Description	Returns whether the signbit is set or not.
Example	`signbit(-0.0)`
Result	`true`

`sin(x)`

Description	Computes the sin of `x`.
Example	`sin(90)`
Result	`0.8939966636005579`

`sqrt(x)`

Description	Returns the square root of the number.
Example	`sqrt(9)`
Result	`3`

`subtract(x, y)`

Description	Alias for `x - y`.
Example	`subtract(2, 3)`
Result	`-1`

`tan(x)`

Description	Computes the tangent of `x`.
Example	`tan(90)`
Result	`-1.995200412208242`

`trunc(x)`

Description	Truncates the number.
Example	`trunc(17.4)`
Result	`17`

`xor(x)`

Description	Bitwise XOR.
Example	`xor(17, 5)`
Result	`20`

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:

```
SELECT 'abc' LIKE 'abc'; -- true
SELECT 'abc' LIKE 'a%' ; -- true
SELECT 'abc' LIKE '_b_'; -- true
SELECT 'abc' LIKE 'c';   -- false
SELECT 'abc' LIKE 'c%' ; -- false
SELECT 'abc' LIKE '%c';  -- true
SELECT '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:

```
SELECT 'abc' ILIKE '%C'; -- true
```

```
SELECT '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':

```
SELECT 'a%c' LIKE 'a$%c' ESCAPE '$'; -- true
SELECT 'azc' LIKE 'a$%c' ESCAPE '$'; -- false
```

Case-insensitive ILIKE with ESCAPE:

```
SELECT '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:

```
SELECT 'abc' SIMILAR TO 'abc';       -- true
SELECT 'abc' SIMILAR TO 'a';         -- false
SELECT 'abc' SIMILAR TO '.*(b|d).*'; -- true
SELECT 'abc' SIMILAR TO '(b|c).*';   -- false
SELECT '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:

```
SELECT 'best.txt' GLOB '*.txt';           -- true
SELECT 'best.txt' GLOB '????.txt';        -- true
SELECT 'best.txt' GLOB '?.txt';           -- false
SELECT 'best.txt' GLOB '[abc]est.txt';    -- true
SELECT 'best.txt' GLOB '[a-z]est.txt';    -- true
```

The bracket syntax is case-sensitive:

```
SELECT 'Best.txt' GLOB '[a-z]est.txt';    -- false
SELECT 'Best.txt' GLOB '[a-zA-Z]est.txt'; -- true
```

The `!` applies to all characters within the brackets:

```
SELECT 'Best.txt' GLOB '[!a-zA-Z]est.txt'; -- false
```

To negate a GLOB operator, negate the entire expression:

```
-- (NOT GLOB is not valid syntax)
SELECT 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

DuckDB's regex support is documented on the Regular Expressions page.

Regular Expressions

DuckDB offers pattern matching operators (`LIKE`, `SIMILAR TO`, `GLOB`), as well as support for regular expressions via functions.

Regular Expression Syntax

DuckDB uses the RE2 library as its regular expression engine. For the regular expression syntax, see the RE2 docs.

Functions

All functions accept an optional set of options.

Name	Description
`regexp_extract_all(string, regex[, group = 0][, options])`	Split the *string* along the *regex* and extract all occurrences of *group*.
`regexp_extract(string, pattern, name_list[, options])`	If *string* contains the regexp *pattern*, returns the capturing groups as a struct with corresponding names from *name_list*.
`regexp_extract(string, pattern[, idx][, options])`	If *string* contains the regexp *pattern*, returns the capturing group specified by optional parameter *idx*. The *idx* must be a constant value.

Name	Description
`regexp_full_match(string, regex[, options])`	Returns `true` if the entire *string* matches the *regex*.
`regexp_matches(string, pattern[, options])`	Returns `true` if *string* contains the regexp *pattern*, `false` otherwise.
`regexp_replace(string, pattern, replacement[, options])`	If *string* contains the regexp *pattern*, replaces the matching part with *replacement*.
`regexp_split_to_array(string, regex[, options])`	Alias of `string_split_regex`. Splits the *string* along the *regex*.
`regexp_split_to_table(string, regex[, options])`	Splits the *string* along the *regex* and returns a row for each part.

`regexp_extract_all(string, regex[, group = 0][, options])`

Description	Split the *string* along the *regex* and extract all occurrences of *group*. A set of optional `*options*` can be set.
Example	`regexp_extract_all('hello_world', '([a-z ]+)_?', 1)`
Result	`[hello, world]`

`regexp_extract(string, pattern, name_list[, options])`

Description	If *string* contains the regexp *pattern*, returns the capturing groups as a struct with corresponding names from *name_list*. A set of optional `*options*` can be set.
Example	`regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd'])`
Result	`{'y':'2023', 'm':'04', 'd':'15'}`

`regexp_extract(string, pattern[, idx][, options])`

Description	If *string* contains the regexp *pattern*, returns the capturing group specified by optional parameter *idx*. The *idx* must be a constant value. A set of optional `*options*` can be set.
Result	`hello`

`regexp_full_match(string, regex[, options])`

Description	Returns `true` if the entire *string* matches the *regex*. A set of optional `*options*` can be set.
Example	`regexp_full_match('anabanana', '(an)*')`
Result	`false`

`regexp_matches(string, pattern[, options])`

Description	Returns `true` if *string* contains the regexp *pattern*, `false` otherwise. A set of optional `*options*` can be set.
Example	`regexp_matches('anabanana', '(an)*')`
Result	`true`

`regexp_replace(string, pattern, replacement[, options])`

Description	If *string* contains the regexp *pattern*, replaces the matching part with *replacement*. A set of optional `*options*` can be set.
Example	`regexp_replace('hello', '[lo]', '-')`
Result	`he-lo`

`regexp_split_to_array(string, regex[, options])`

Description	Alias of `string_split_regex`. Splits the *string* along the *regex*. A set of optional `*options*` can be set.
Example	`regexp_split_to_array('hello world; 42', ';? ')`
Result	`['hello', 'world', '42']`

`regexp_split_to_table(string, regex[, options])`

Description	Splits the *string* along the *regex* and returns a row for each part. A set of optional `*options*` can be set.
Example	`regexp_split_to_array('hello world; 42', ';? ')`
Result	Two rows: `'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:

```
SELECT regexp_matches('abc', 'abc');       -- true
SELECT regexp_matches('abc', '^abc$');     -- true
SELECT regexp_matches('abc', 'a');         -- true
SELECT regexp_matches('abc', '^a$');       -- false
SELECT regexp_matches('abc', '.*(b|d).*'); -- true
SELECT regexp_matches('abc', '(b|c).*');   -- true
SELECT regexp_matches('abc', '^(b|c).*');  -- false
SELECT regexp_matches('abc', '(?i)A');     -- true
SELECT regexp_matches('abc', 'A', 'i');    -- true
```

Options for Regular Expression Functions

The regex functions support 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
`'g'`	global replace, only available for `regexp_replace`
`'s'`	non-newline sensitive matching

For example:

```
SELECT regexp_matches('abcd', 'ABC', 'c'); -- false
SELECT regexp_matches('abcd', 'ABC', 'i'); -- true
SELECT regexp_matches('ab^/$cd', '^/$', 'l'); -- true
SELECT regexp_matches(E'hello\nworld', 'hello.world', 'p'); -- false
SELECT regexp_matches(E'hello\nworld', 'hello.world', 's'); -- true
```

Using `regexp_matches`

The `regexp_matches` operator will be optimized to the `LIKE` operator when possible. To achieve best performance, the `'c'` option (case-sensitive matching) should be passed if applicable. Note that by default the RE2 library doesn't match the `.` character to newline.

Original	Optimized equivalent
`regexp_matches('hello world', '^hello', 'c')`	`prefix('hello world', 'hello')`
`regexp_matches('hello world', 'world$', 'c')`	`suffix('hello world', 'world')`
`regexp_matches('hello world', 'hello.world', 'c')`	`LIKE 'hello_world'`
`regexp_matches('hello world', 'he.*rld', 'c')`	`LIKE '%he%rld'`

Using `regexp_replace`

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. Note that by default, `regexp_replace` only replaces the first occurrence of the regular expression. To replace all occurrences, use the global replace (g) flag.

Some examples for using `regexp_replace`:

```
SELECT regexp_replace('abc', '(b|c)', 'X');         -- aXc
SELECT regexp_replace('abc', '(b|c)', 'X', 'g');    -- aXX
SELECT regexp_replace('abc', '(b|c)', '\1\1\1\1'); -- abbbbc
SELECT regexp_replace('abc', '(.*)c', '\1e');       -- abe
SELECT regexp_replace('abc', '(a)(b)', '\2\1');     -- bac
```

Using `regexp_extract`

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.

```
SELECT regexp_extract('abc', '.b.');     -- abc
SELECT regexp_extract('abc', '.b.', 0);  -- abc
SELECT regexp_extract('abc', '.b.', 1);  -- (empty)
SELECT regexp_extract('abc', '([a-z])(b)', 1); -- a
SELECT 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`:

```
SELECT regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd']);
```

```
{'y': 2023, 'm': 04, 'd': 15}
```

```
SELECT regexp_extract('2023-04-15 07:59:56', '^(\d+)-(\d+)-(\d+) (\d+):(\d+):(\d+)', ['y', 'm', 'd']);
```

```
{'y': 2023, 'm': 04, 'd': 15}
```

```
SELECT regexp_extract('duckdb_0_7_1', '^(\w+)_(\d+)_(\d+)', ['tool', 'major', 'minor', 'fix']);
```

```
Binder Error: Not enough group names in regexp_extract
```

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.

Text Functions

Text Functions and Operators

This section describes functions and operators for examining and manipulating string values.

Name	Description
`string ^@ search_string`	Return true if `string` begins with `search_string`.
`string || string`	String concatenation.
`string[index]`	Extract a single character using a (1-based) index.
`string[begin:end]`	Extract a string using slice conventions. Missing `begin` or `end` arguments are interpreted as the beginning or end of the list respectively. Negative values are accepted.
`string LIKE target`	Returns true if the `string` matches the like specifier (see Pattern Matching).
`string SIMILAR TO regex`	Returns `true` if the `string` matches the `regex`; identical to `regexp_full_match` (see Pattern Matching).
`array_extract(list, index)`	Extract a single character using a (1-based) index.
`array_slice(list, begin, end)`	Extract a string using slice conventions. Negative values are accepted.
`ascii(string)`	Returns an integer that represents the Unicode code point of the first character of the `string`.
`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.
`bit_length(string)`	Number of bits in a string.

Name	Description
`chr(x)`	Returns a character which is corresponding the ASCII code value or Unicode code point.
`concat_ws(separator, string, ...)`	Concatenate strings together separated by the specified separator.
`concat(string, ...)`	Concatenate many strings together.
`contains(string, search_string)`	Return true if `search_string` is found within `string`.
`ends_with(string, search_string)`	Return true if `string` ends with `search_string`.
`format_bytes(bytes)`	Converts bytes to a human-readable representation using units based on powers of 2 (KiB, MiB, GiB, etc.).
`format(format, parameters, ...)`	Formats a string using the fmt syntax.
`from_base64(string)`	Convert a base64 encoded string to a character string.
`greatest(x1, x2, ...)`	Selects the largest value using lexicographical ordering. Note that lowercase characters are considered "larger" than uppercase characters and collations are not supported.
`hash(value)`	Returns a `UBIGINT` with the hash of the `value`.
`ilike_escape(string, like_specifier, escape_character)`	Returns true if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
`instr(string, search_string)`	Return location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
`least(x1, x2, ...)`	Selects the smallest value using lexicographical ordering. Note that uppercase characters are considered "smaller" than lowercase characters, and collations are not supported.
`left_grapheme(string, count)`	Extract the left-most grapheme clusters.
`left(string, count)`	Extract the left-most count characters.
`length_grapheme(string)`	Number of grapheme clusters in `string`.
`length(string)`	Number of characters in `string`.
`like_escape(string, like_specifier, escape_character)`	Returns true if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
`lower(string)`	Convert `string` to lower case.
`lpad(string, count, character)`	Pads the `string` with the character from the left until it has count characters.
`ltrim(string, characters)`	Removes any occurrences of any of the `characters` from the left side of the `string`.
`ltrim(string)`	Removes any spaces from the left side of the `string`.
`md5(value)`	Returns the MD5 hash of the `value`.
`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.
`not_ilike_escape(string, like_specifier, escape_character)`	Returns false if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.

Name	Description
not_like_escape(string, like_specifier, escape_character)	Returns false if the string matches the like_specifier (see Pattern Matching) using case-insensitive matching. escape_character is used to search for wildcard characters in the string.
ord(string)	Return ASCII character code of the leftmost character in a string.
parse_dirname(path, separator)	Returns the top-level directory name from the given path. separator options: system, both_slash (default), forward_slash, backslash.
parse_dirpath(path, separator)	Returns the head of the path (the pathname until the last slash) similarly to Python's os.path.dirname function. separator options: system, both_slash (default), forward_slash, backslash.
parse_filename(path, trim_extension, separator)	Returns the last component of the path similarly to Python's os.path.basename function. If trim_extension is true, the file extension will be removed (defaults to false). separator options: system, both_slash (default), forward_slash, backslash.
parse_path(path, separator)	Returns a list of the components (directories and filename) in the path similarly to Python's pathlib.parts function. separator options: system, both_slash (default), forward_slash, backslash.
position(search_string IN string)	Return location of first occurrence of search_string in string, counting from 1. Returns 0 if no match found.
printf(format, parameters...)	Formats a string using printf syntax.
read_text(source)	Returns the content from source (a filename, a list of filenames, or a glob pattern) as a VARCHAR. The file content is first validated to be valid UTF-8. If read_text attempts to read a file with invalid UTF-8 an error is thrown suggesting to use read_blob instead. See the read_text guide for more details.
regexp_escape(string)	Escapes special patterns to turn string into a regular expression similarly to Python's re.escape function.
regexp_extract_all(string, regex[, group = 0])	Split the string along the regex and extract all occurrences of group.
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 (see Pattern Matching).
regexp_extract(string, pattern[, idx])	If string contains the regexp pattern, returns the capturing group specified by optional parameter idx (see Pattern Matching).
regexp_full_match(string, regex)	Returns true if the entire string matches the regex (see Pattern Matching).
regexp_matches(string, pattern)	Returns true if string contains the regexp pattern, false otherwise (see Pattern Matching).
regexp_replace(string, pattern, replacement)	If string contains the regexp pattern, replaces the matching part with replacement (see Pattern Matching).
regexp_split_to_array(string, regex)	Splits the string along the regex.
regexp_split_to_table(string, regex)	Splits the string along the regex and returns a row for each part.
repeat(string, count)	Repeats the string count number of times.
replace(string, source, target)	Replaces any occurrences of the source with target in string.

Name	Description
`reverse(string)`	Reverses the `string`.
`right_grapheme(string, count)`	Extract the right-most `count` grapheme clusters.
`right(string, count)`	Extract the right-most `count` characters.
`rpad(string, count, character)`	Pads the `string` with the character from the right until it has `count` characters.
`rtrim(string, characters)`	Removes any occurrences of any of the `characters` from the right side of the `string`.
`rtrim(string)`	Removes any spaces from the right side of the `string`.
`sha256(value)`	Returns a VARCHAR with the SHA-256 hash of the `value`.
`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).
`starts_with(string, search_string)`	Return true if `string` begins with `search_string`.
`str_split_regex(string, regex)`	Splits the `string` along the `regex`.
`string_split_regex(string, regex)`	Splits the `string` along the `regex`.
`string_split(string, separator)`	Splits the `string` along the `separator`.
`strip_accents(string)`	Strips accents from `string`.
`strlen(string)`	Number of bytes in `string`.
`strpos(string, search_string)`	Return location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
`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_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.
`to_base64(blob)`	Convert a blob to a base64 encoded string.
`trim(string, characters)`	Removes any occurrences of any of the `characters` from either side of the `string`.
`trim(string)`	Removes any spaces from either side of the `string`.
`unicode(string)`	Returns the Unicode code of the first character of the `string`.
`upper(string)`	Convert `string` to upper case.

`string ^@ search_string`

Description	Return true if `string` begins with `search_string`.
Example	`'abc' ^@ 'a'`
Result	`true`
Alias	`starts_with`

`string || string`

Description	String concatenation.
Example	`'Duck' || 'DB'`
Result	DuckDB
Alias	`concat`

`string[index]`

Description	Extract a single character using a (1-based) index.
Example	`'DuckDB'[4]`
Result	k
Alias	`array_extract`

`string[begin:end]`

Description	Extract a string using slice conventions. Missing `begin` or `end` arguments are interpreted as the beginning or end of the list respectively. Negative values are accepted.
Example	`'DuckDB'[:4]`
Result	Duck
Alias	`array_slice`

`string LIKE target`

Description	Returns true if the `string` matches the like specifier (see Pattern Matching).
Example	`'hello' LIKE '%lo'`
Result	`true`

`string SIMILAR TO regex`

Description	Returns `true` if the `string` matches the `regex`; identical to `regexp_full_match` (see Pattern Matching)
Example	`'hello' SIMILAR TO 'l+'`
Result	`false`

`array_extract(list, index)`

Description	Extract a single character using a (1-based) index.
Example	`array_extract('DuckDB', 2)`
Result	u
Aliases	`list_element`, `list_extract`

`array_slice(list, begin, end)`

Description	Extract a string using slice conventions. Negative values are accepted.
Example 1	`array_slice('DuckDB', 3, 4)`
Result	ck
Example 2	`array_slice('DuckDB', 3, NULL)`
Result	NULL
Example 3	`array_slice('DuckDB', 0, -3)`
Result	Duck

`ascii(string)`

Description	Returns an integer that represents the Unicode code point of the first character of the `string`.
Example	`ascii('Ω')`
Result	937

`bar(x, min, max[, width])`

Description	Draw a band whose width is proportional to (`x - min`) and equal to `width` characters when x = `max`. `width` defaults to 80.
Example	`bar(5, 0, 20, 10)`
Result	██

`bit_length(string)`

Description	Number of bits in a string.
Example	`bit_length('abc')`
Result	24

`chr(x)`

Description	Returns a character which is corresponding the ASCII code value or Unicode code point.

Example	`chr(65)`
Result	A

`concat_ws(separator, string, ...)`

Description	Concatenate strings together separated by the specified separator.
Example	`concat_ws(', ', 'Banana', 'Apple', 'Melon')`
Result	Banana, Apple, Melon

`concat(string, ...)`

Description	Concatenate many strings together.
Example	`concat('Hello', ' ', 'World')`
Result	Hello World

`contains(string, search_string)`

Description	Return true if `search_string` is found within `string`.
Example	`contains('abc', 'a')`
Result	`true`

`ends_with(string, search_string)`

Description	Return true if `string` ends with `search_string`.
Example	`ends_with('abc', 'c')`
Result	`true`
Alias	`suffix`

`format_bytes(bytes)`

Description	Converts bytes to a human-readable representation using units based on powers of 2 (KiB, MiB, GiB, etc.).
Example	`format_bytes(16384)`
Result	`16.0 KiB`

`format(format, parameters, ...)`

Description	Formats a string using the fmt syntax.
Example	`format('Benchmark "{}" took {} seconds', 'CSV', 42)`
Result	`Benchmark "CSV" took 42 seconds`

`from_base64(string)`

Description	Convert a base64 encoded string to a character string.
Example	`from_base64('QQ==')`
Result	`'A'`

`greatest(x1, x2, ...)`

Description	Selects the largest value using lexicographical ordering. Note that lowercase characters are considered "larger" than uppercase characters and collations are not supported.
Example	`greatest('abc', 'bcd', 'cde', 'EFG')`
Result	`'cde'`

`hash(value)`

Description	Returns a `UBIGINT` with the hash of the `value`.
Example	`hash('🦆')`
Result	`2595805878642663834`

`ilike_escape(string, like_specifier, escape_character)`

Description	Returns true if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`ilike_escape('A%c', 'a$%C', '$')`
Result	`true`

`instr(string, search_string)`

Description	Return location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
Example	`instr('test test', 'es')`

Result	2

`least(x1, x2, ...)`

Description	Selects the smallest value using lexicographical ordering. Note that uppercase characters are considered "smaller" than lowercase characters, and collations are not supported.
Example	`least('abc', 'BCD', 'cde', 'EFG')`
Result	`'BCD'`

`left_grapheme(string, count)`

Description	Extract the left-most grapheme clusters.
Example	`left_grapheme('👩🏻‍🦰👨🏽‍🦰', 1)`
Result	👩🏻‍🦰

`left(string, count)`

Description	Extract the left-most count characters.
Example	`left('Hello🦆', 2)`
Result	He

`length_grapheme(string)`

Description	Number of grapheme clusters in `string`.
Example	`length_grapheme('👩🏻‍🦰👨🏽‍🦰')`
Result	2

`length(string)`

Description	Number of characters in `string`.
Example	`length('Hello🦆')`
Result	6

`like_escape(string, like_specifier, escape_character)`

Description	Returns true if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`like_escape('a%c', 'a$%c', '$')`
Result	`true`

`lower(string)`

Description	Convert `string` to lower case.
Example	`lower('Hello')`
Result	`hello`
Alias	`lcase`

`lpad(string, count, character)`

Description	Pads the `string` with the character from the left until it has count characters.
Example	`lpad('hello', 8, '>')`
Result	`>>>hello`

`ltrim(string, characters)`

Description	Removes any occurrences of any of the `characters` from the left side of the `string`.
Example	`ltrim('>>>>test<<', '><')`
Result	`test<<`

`ltrim(string)`

Description	Removes any spaces from the left side of the `string`. In the example, the ␣ symbol denotes a space character.
Example	`ltrim('␣␣␣␣test␣␣')`
Result	`test␣␣`

`md5(value)`

Description	Returns the MD5 hash of the `value`.
Example	`md5('123')`
Result	`202cb962ac59075b964b07152d234b70`

nfc_normalize(string)

Description	Convert string to Unicode NFC normalized string. Useful for comparisons and ordering if text data is mixed between NFC normalized and not.
Example	`nfc_normalize('ardèch')`
Result	`ardèch`

not_ilike_escape(string, like_specifier, escape_character)

Description	Returns false if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`not_ilike_escape('A%c', 'a$%C', '$')`
Result	`false`

not_like_escape(string, like_specifier, escape_character)

Description	Returns false if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`not_like_escape('a%c', 'a$%c', '$')`
Result	`false`

ord(string)

Description	Return ASCII character code of the leftmost character in a string.
Example	`ord('ü')`
Result	`252`

parse_dirname(path, separator)

Description	Returns the top-level directory name from the given path. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_dirname('path/to/file.csv', 'system')`
Result	`path`

parse_dirpath(path, separator)

Description	Returns the head of the path (the pathname until the last slash) similarly to Python's `os.path.dirname` function. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_dirpath('/path/to/file.csv', 'forward_slash')`
Result	`/path/to`

`parse_filename(path, trim_extension, separator)`

Description	Returns the last component of the path similarly to Python's `os.path.basename` function. If `trim_extension` is true, the file extension will be removed (defaults to `false`). `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_filename('path/to/file.csv', true, 'system')`
Result	`file`

`parse_path(path, separator)`

Description	Returns a list of the components (directories and filename) in the path similarly to Python's `pathlib.parts` function. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_path('/path/to/file.csv', 'system')`
Result	`[/, path, to, file.csv]`

`position(search_string IN string)`

Description	Return location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
Example	`position('b' IN 'abc')`
Result	2

`printf(format, parameters...)`

Description	Formats a `string` using printf syntax.
Example	`printf('Benchmark "%s" took %d seconds', 'CSV', 42)`
Result	`Benchmark "CSV" took 42 seconds`

`read_text(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a VARCHAR. The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead. See the `read_text` guide for more details.
Example	`read_text('hello.txt')`
Result	`hello\n`

`regexp_escape(string)`

Description	Escapes special patterns to turn `string` into a regular expression similarly to Python's `re.escape` function.
Example	`regexp_escape('http://d.org')`
Result	`http\:\/\/d\.org`

`regexp_extract_all(string, regex[, group = 0])`

Description	Split the `string` along the `regex` and extract all occurrences of `group`.
Example	`regexp_extract_all('hello_world', '([a-z ]+)_?', 1)`
Result	`[hello, world]`

`regexp_extract(string, pattern, name_list)`

Description	If `string` contains the regexp `pattern`, returns the capturing groups as a struct with corresponding names from `name_list` (see Pattern Matching).
Example	`regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd'])`
Result	`{'y':'2023', 'm':'04', 'd':'15'}`

`regexp_extract(string, pattern[, idx])`

Description	If `string` contains the regexp `pattern`, returns the capturing group specified by optional parameter `idx` (see Pattern Matching).
Example	`regexp_extract('hello_world', '([a-z ]+)_?', 1)`
Result	`hello`

`regexp_full_match(string, regex)`

Description	Returns `true` if the entire `string` matches the `regex` (see Pattern Matching).
Example	`regexp_full_match('anabanana', '(an)')`

Result	`false`

`regexp_matches(string, pattern)`

Description	Returns `true` if `string` contains the regexp `pattern`, `false` otherwise (see Pattern Matching).
Example	`regexp_matches('anabanana', '(an)')`
Result	`true`

`regexp_replace(string, pattern, replacement)`

Description	If `string` contains the regexp `pattern`, replaces the matching part with `replacement` (see Pattern Matching).
Example	`regexp_replace('hello', '[lo]', '-')`
Result	`he-lo`

`regexp_split_to_array(string, regex)`

Description	Splits the `string` along the `regex`.
Example	`regexp_split_to_array('hello world; 42', ';? ')`
Result	`['hello', 'world', '42']`
Aliases	`string_split_regex`, `str_split_regex`

`regexp_split_to_table(string, regex)`

Description	Splits the `string` along the `regex` and returns a row for each part.
Example	`regexp_split_to_table('hello world; 42', ';? ')`
Result	Two rows: `'hello'`, `'world'`

`repeat(string, count)`

Description	Repeats the `string` `count` number of times.
Example	`repeat('A', 5)`
Result	AAAAA

`replace(string, source, target)`

Description	Replaces any occurrences of the `source` with `target` in `string`.
Example	`replace('hello', 'l', '-')`
Result	`he--o`

`reverse(string)`

Description	Reverses the `string`.
Example	`reverse('hello')`
Result	`olleh`

`right_grapheme(string, count)`

Description	Extract the right-most `count` grapheme clusters.
Example	`right_grapheme('👩🏻‍🤝‍👨🏽♀', 1)`
Result	👩🏽♀

`right(string, count)`

Description	Extract the right-most `count` characters.
Example	`right('Hello🦆', 3)`
Result	`lo🦆`

`rpad(string, count, character)`

Description	Pads the `string` with the character from the right until it has `count` characters.
Example	`rpad('hello', 10, '<')`
Result	`hello<<<<<`

`rtrim(string, characters)`

Description	Removes any occurrences of any of the `characters` from the right side of the `string`.
Example	`rtrim('>>>>test<<', '><')`
Result	`>>>>test`

`rtrim(string)`

Description	Removes any spaces from the right side of the `string`. In the example, the ␣ symbol denotes a space character.
Example	`rtrim('␣␣␣␣test␣␣')`
Result	`␣␣␣␣test`

sha256(value)

Description	Returns a VARCHAR with the SHA-256 hash of the `value`.
Example	`sha256('🦆')`
Result	`d7a5c5e0d1d94c32218539e7e47d4ba9c3c7b77d61332fb60d633dde89e473fb`

split_part(string, separator, index)

Description	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).
Example	`split_part('a;b;c', ';', 2)`
Result	b

starts_with(string, search_string)

Description	Return true if `string` begins with `search_string`.
Example	`starts_with('abc', 'a')`
Result	`true`

str_split_regex(string, regex)

Description	Splits the `string` along the `regex`.
Example	`str_split_regex('hello world; 42', ';? ')`
Result	`['hello', 'world', '42']`
Aliases	`string_split_regex`, `regexp_split_to_array`

string_split_regex(string, regex)

Description	Splits the `string` along the `regex`.
Example	`string_split_regex('hello world; 42', ';? ')`
Result	`['hello', 'world', '42']`
Aliases	`str_split_regex`, `regexp_split_to_array`

string_split(string, separator)

Description	Splits the `string` along the `separator`.
Example	`string_split('hello world', ' ')`
Result	`['hello', 'world']`
Aliases	`str_split`, `string_to_array`

strip_accents(string)

Description	Strips accents from `string`.
Example	`strip_accents('mühleisen')`
Result	`muhleisen`

strlen(string)

Description	Number of bytes in `string`.
Example	`strlen('🦆')`
Result	4

strpos(string, search_string)

Description	Return location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
Example	`strpos('test test', 'es')`
Result	2
Alias	`instr`

substring(string, start, length)

Description	Extract substring of `length` characters starting from character `start`. Note that a `start` value of `1` refers to the `first` character of the string.
Example	`substring('Hello', 2, 2)`
Result	`el`
Alias	`substr`

substring_grapheme(string, start, length)

Description	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.
Example	`substring_grapheme('🦆🤦🏼‍♂️🤦🏽‍♀️🦆', 3, 2)`
Result	🤦🏽‍♀️🦆

`to_base64(blob)`

Description	Convert a blob to a base64 encoded string.
Example	`to_base64('A'::blob)`
Result	`QQ==`
Alias	`base64`

`trim(string, characters)`

Description	Removes any occurrences of any of the `characters` from either side of the `string`.
Example	`trim('>>>>test<<', '><')`
Result	`test`

`trim(string)`

Description	Removes any spaces from either side of the `string`.
Example	`trim('    test  ')`
Result	`test`

`unicode(string)`

Description	Returns the Unicode code of the first character of the `string`. Returns `-1` when `string` is empty, and `NULL` when `string` is `NULL`.
Example	`[unicode('âbcd'), unicode('â'), unicode(''), unicode(NULL)]`
Result	`[226, 226, -1, NULL]`

`upper(string)`

Description	Convert `string` to upper case.
Example	`upper('Hello')`
Result	`HELLO`
Alias	`ucase`

Text Similarity Functions

These functions are used to measure the similarity of two strings using various similarity measures.

Name	Description
`damerau_levenshtein(s1, s2)`	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. Characters of different cases (e.g., a and A) are considered different.
`editdist3(s1, s2)`	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. Characters of different cases (e.g., a and A) are considered different.
`hamming(s1, s2)`	The Hamming distance between to strings, i.e., the number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., a and A) are considered different.
`jaccard(s1, s2)`	The Jaccard similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1.
`jaro_similarity(s1, s2)`	The Jaro similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1.
`jaro_winkler_similarity(s1, s2)`	The Jaro-Winkler similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1.
`levenshtein(s1, s2)`	The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Characters of different cases (e.g., a and A) are considered different.
`mismatches(s1, s2)`	Alias for `hamming(s1, s2)`. The number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., a and A) are considered different.

`damerau_levenshtein(s1, s2)`

Description	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. Characters of different cases (e.g., a and A) are considered different.
Example	`damerau_levenshtein('duckdb', 'udckbd')`
Result	2

`editdist3(s1, s2)`

Description	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. Characters of different cases (e.g., a and A) are considered different.
Example	`editdist3('duck', 'db')`
Result	3

hamming(s1, s2)

Description	The Hamming distance between to strings, i.e., the number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., a and A) are considered different.
Example	`hamming('duck', 'luck')`
Result	`1`

jaccard(s1, s2)

Description	The Jaccard similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1.
Example	`jaccard('duck', 'luck')`
Result	`0.6`

jaro_similarity(s1, s2)

Description	The Jaro similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1.
Example	`jaro_similarity('duck', 'duckdb')`
Result	`0.88`

jaro_winkler_similarity(s1, s2)

Description	The Jaro-Winkler similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1.
Example	`jaro_winkler_similarity('duck', 'duckdb')`
Result	`0.93`

levenshtein(s1, s2)

Description	The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Characters of different cases (e.g., a and A) are considered different.
Example	`levenshtein('duck', 'db')`
Result	`3`

mismatches(s1, s2)

Description	Alias for `hamming(s1, s2)`. The number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., a and A) are considered different.
Example	`mismatches('duck', 'luck')`
Result	`1`

Formatters

`fmt` Syntax

The `format(format, parameters...)` function formats strings, loosely following the syntax of the {fmt} open-source formatting library.

Format without additional parameters:

```
SELECT format('Hello world'); -- Hello world
```

Format a string using {}:

```
SELECT format('The answer is {}', 42); -- The answer is 42
```

Format a string using positional arguments:

```
SELECT format('I''d rather be {1} than {0}.', 'right', 'happy'); -- I'd rather be happy than right.
```

Format Specifiers

Specifier	Description	Example
`{:d}`	integer	`123456`
`{:E}`	scientific notation	`3.141593E+00`
`{:f}`	float	`4.560000`
`{:o}`	octal	`361100`
`{:s}`	string	`asd`
`{:x}`	hexadecimal	`1e240`
`{:tX}`	integer, X is the thousand separator	`123 456`

Formatting Types

Integers:

```
SELECT format('{} + {} = {}', 3, 5, 3 + 5); -- 3 + 5 = 8
```

Booleans:

```
SELECT format('{} != {}', true, false); -- true != false
```

Format datetime values:

```
SELECT format('{}', DATE '1992-01-01'); -- 1992-01-01
SELECT format('{}', TIME '12:01:00'); -- 12:01:00
SELECT format('{}', TIMESTAMP '1992-01-01 12:01:00'); -- 1992-01-01 12:01:00
```

Format BLOB:

```
SELECT format('{}', BLOB '\x00hello'); -- \x00hello
```

Pad integers with 0s:

```
SELECT format('{:04d}', 33); -- 0033
```

Create timestamps from integers:

```
SELECT format('{:02d}:{:02d}:{:02d} {}', 12, 3, 16, 'AM'); -- 12:03:16 AM
```

Convert to hexadecimal:

```
SELECT format('{:x}', 123_456_789); -- 75bcd15
```

Convert to binary:

```
SELECT format('{:b}', 123_456_789); -- 111010110111100110100010101
```

Print Numbers with Thousand Separators

```
SELECT format('{:,}',  123_456_789); -- 123,456,789
SELECT format('{:t.}', 123_456_789); -- 123.456.789
SELECT format('{:''}', 123_456_789); -- 123'456'789
SELECT format('{:_}',  123_456_789); -- 123_456_789
SELECT format('{:t }', 123_456_789); -- 123 456 789
SELECT format('{:tX}', 123_456_789); -- 123X456X789
```

`printf` Syntax

The `printf(format, parameters...)` function formats strings using the `printf` syntax.

Format without additional parameters:

```
SELECT printf('Hello world');
```

```
Hello world
```

Format a string using arguments in a given order:

```
SELECT printf('The answer to %s is %d', 'life', 42);
```

```
The answer to life is 42
```

Format a string using positional arguments `%position$formatter`, e.g., the second parameter as a string is encoded as `%2$s`:

```
SELECT printf('I''d rather be %2$s than %1$s.', 'right', 'happy');
```

```
I'd rather be happy than right.
```

Format Specifiers

Specifier	Description	Example
%c	character code to character	a
%d	integer	123456
%Xd	integer with thousand seperarator X from `,`, `.`, `''`, `_`	123_456
%E	scientific notation	3.141593E+00

Specifier	Description	Example
%f	float	4.560000
%hd	integer	123456
%hhd	integer	123456
%lld	integer	123456
%o	octal	361100
%s	string	asd
%x	hexadecimal	1e240

Formatting Types

Integers:

```
SELECT printf('%d + %d = %d', 3, 5, 3 + 5); -- 3 + 5 = 8
```

Booleans:

```
SELECT printf('%s != %s', true, false); -- true != false
```

Format datetime values:

```
SELECT printf('%s', DATE '1992-01-01'); -- 1992-01-01
SELECT printf('%s', TIME '12:01:00'); -- 12:01:00
SELECT printf('%s', TIMESTAMP '1992-01-01 12:01:00'); -- 1992-01-01 12:01:00
```

Format BLOB:

```
SELECT printf('%s', BLOB '\x00hello'); -- \x00hello
```

Pad integers with 0s:

```
SELECT printf('%04d', 33); -- 0033
```

Create timestamps from integers:

```
SELECT printf('%02d:%02d:%02d %s', 12, 3, 16, 'AM'); -- 12:03:16 AM
```

Convert to hexadecimal:

```
SELECT printf('%x', 123_456_789); -- 75bcd15
```

Convert to binary:

```
SELECT printf('%b', 123_456_789); -- 111010110111100110100010101
```

Thousand Separators

```
SELECT printf('%,d',  123_456_789); -- 123,456,789
SELECT printf('%.d',  123_456_789); -- 123.456.789
SELECT printf('%''d', 123_456_789); -- 123'456'789
SELECT printf('%_d',  123_456_789); -- 123_456_789
```

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.

Name	Description
`current_time`	Current time (start of current transaction).
`date_diff(part, starttime, endtime)`	The number of partition boundaries between the times.
`date_part(part, time)`	Get subfield (equivalent to `extract`).
`date_sub(part, starttime, endtime)`	The number of complete partitions between the times.
`datediff(part, starttime, endtime)`	Alias of `date_diff`. The number of partition boundaries between the times.
`datepart(part, time)`	Alias of date_part. Get subfield (equivalent to `extract`).
`datesub(part, starttime, endtime)`	Alias of date_sub. The number of complete partitions between the times.
`extract(part FROM time)`	Get subfield from a time.
`get_current_time()`	Current time (start of current transaction).
`make_time(bigint, bigint, double)`	The time for the given parts.

The only date parts that are defined for times are `epoch`, `hours`, `minutes`, `seconds`, `milliseconds` and `microseconds`.

`current_time`

Description	Current time (start of current transaction). Note that parentheses should be omitted.
Example	`current_time`
Result	`10:31:58.578`
Alias	`get_current_time()`

`date_diff(part, starttime, endtime)`

Description	The number of partition boundaries between the times.
Example	`date_diff('hour', TIME '01:02:03', TIME '06:01:03')`
Result	5

`date_part(part, time)`

Description	Get subfield (equivalent to `extract`).
Example	`date_part('minute', TIME '14:21:13')`
Result	21

`date_sub(part, starttime, endtime)`

Description	The number of complete partitions between the times.
Example	`date_sub('hour', TIME '01:02:03', TIME '06:01:03')`
Result	4

`datediff(part, starttime, endtime)`

Description	Alias of `date_diff`. The number of partition boundaries between the times.
Example	`datediff('hour', TIME '01:02:03', TIME '06:01:03')`
Result	5

`datepart(part, time)`

Description	Alias of date_part. Get subfield (equivalent to `extract`).
Example	`datepart('minute', TIME '14:21:13')`
Result	21

`datesub(part, starttime, endtime)`

Description	Alias of date_sub. The number of complete partitions between the times.
Example	`datesub('hour', TIME '01:02:03', TIME '06:01:03')`
Result	4

`extract(part FROM time)`

Description	Get subfield from a time.
Example	`extract('hour' FROM TIME '14:21:13')`
Result	`14`

`get_current_time()`

Description	Current time (start of current transaction).
Example	`get_current_time()`
Result	`10:31:58.578`
Alias	`current_time`

`make_time(bigint, bigint, double)`

Description	The time for the given parts.
Example	`make_time(13, 34, 27.123456)`
Result	`13:34:27.123456`

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.

Scalar Timestamp Functions

The table below shows the available scalar functions for `TIMESTAMP` values.

Name	Description
`age(timestamp, timestamp)`	Subtract arguments, resulting in the time difference between the two timestamps.
`age(timestamp)`	Subtract from current_date.
`century(timestamp)`	Extracts the century of a timestamp.
`current_timestamp`	Returns the current timestamp (at the start of the transaction).
`date_diff(part, startdate, enddate)`	The number of partition boundaries between the timestamps.
`date_part([part, ...], timestamp)`	Get the listed subfields as a `struct`. The list must be constant.
`date_part(part, timestamp)`	Get subfield (equivalent to `extract`).
`date_sub(part, startdate, enddate)`	The number of complete partitions between the timestamps.
`date_trunc(part, timestamp)`	Truncate to specified precision.
`datediff(part, startdate, enddate)`	Alias of `date_diff`. The number of partition boundaries between the timestamps.
`datepart([part, ...], timestamp)`	Alias of `date_part`. Get the listed subfields as a `struct`. The list must be constant.
`datepart(part, timestamp)`	Alias of `date_part`. Get subfield (equivalent to `extract`).
`datesub(part, startdate, enddate)`	Alias of `date_sub`. The number of complete partitions between the timestamps.
`datetrunc(part, timestamp)`	Alias of `date_trunc`. Truncate to specified precision.
`dayname(timestamp)`	The (English) name of the weekday.
`epoch_ms(ms)`	Converts ms since epoch to a timestamp.
`epoch_ms(timestamp)`	Converts a timestamp to milliseconds since the epoch.
`epoch_ms(timestamp)`	Return the total number of milliseconds since the epoch.
`epoch_ns(timestamp)`	Return the total number of nanoseconds since the epoch.
`epoch_us(timestamp)`	Return the total number of microseconds since the epoch.
`epoch(timestamp)`	Converts a timestamp to seconds since the epoch.
`extract(field FROM timestamp)`	Get subfield from a timestamp.
`greatest(timestamp, timestamp)`	The later of two timestamps.
`isfinite(timestamp)`	Returns true if the timestamp is finite, false otherwise.
`isinf(timestamp)`	Returns true if the timestamp is infinite, false otherwise.
`last_day(timestamp)`	The last day of the month.
`least(timestamp, timestamp)`	The earlier of two timestamps.

Name	Description
make_timestamp(bigint, bigint, bigint, bigint, bigint, double)	The timestamp for the given parts.
make_timestamp(microseconds)	The timestamp for the given number of µs since the epoch.
monthname(timestamp)	The (English) name of the month.
strftime(timestamp, format)	Converts timestamp to string according to the format string.
strptime(text, format-list)	Converts string to timestamp applying the format strings in the list until one succeeds. Throws on failure.
strptime(text, format)	Converts string to timestamp according to the format string. Throws on failure.
time_bucket(bucket_width, timestamp[, offset])	Truncate timestamp by the specified interval bucket_width. Buckets are offset by offset interval.
time_bucket(bucket_width, timestamp[, 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.
to_timestamp(double)	Converts seconds since the epoch to a timestamp with time zone.
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(text, format)	Converts string to timestamp according to the format string. Returns NULL on failure.

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.

age(timestamp, timestamp)

Description	Subtract arguments, resulting in the time difference between the two timestamps.
Example	age(TIMESTAMP '2001-04-10', TIMESTAMP '1992-09-20')
Result	8 years 6 months 20 days

age(timestamp)

Description	Subtract from current_date.
Example	age(TIMESTAMP '1992-09-20')
Result	29 years 1 month 27 days 12:39:00.844

century(timestamp)

Description	Extracts the century of a timestamp.
Example	`century(TIMESTAMP '1992-03-22')`
Result	`20`

`current_timestamp`

Description	Returns the current timestamp with time zone (at the start of the transaction).
Example	`current_timestamp`
Result	`2024-04-16T09:14:36.098Z`

`date_diff(part, startdate, enddate)`

Description	The number of partition boundaries between the timestamps.
Example	`date_diff('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`
Result	`2`

`date_part([part, ...], timestamp)`

Description	Get the listed subfields as a `struct`. The list must be constant.
Example	`date_part(['year', 'month', 'day'], TIMESTAMP '1992-09-20 20:38:40')`
Result	`{year: 1992, month: 9, day: 20}`

`date_part(part, timestamp)`

Description	Get subfield (equivalent to `extract`).
Example	`date_part('minute', TIMESTAMP '1992-09-20 20:38:40')`
Result	`38`

`date_sub(part, startdate, enddate)`

Description	The number of complete partitions between the timestamps.
Example	`date_sub('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`
Result	`1`

`date_trunc(part, timestamp)`

Description	Truncate to specified precision.
Example	`date_trunc('hour', TIMESTAMP '1992-09-20 20:38:40')`
Result	`1992-09-20 20:00:00`

`datediff(part, startdate, enddate)`

Description	Alias of `date_diff`. The number of partition boundaries between the timestamps.
Example	`datediff('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`
Result	`2`

`datepart([part, ...], timestamp)`

Description	Alias of `date_part`. Get the listed subfields as a `struct`. The list must be constant.
Example	`datepart(['year', 'month', 'day'], TIMESTAMP '1992-09-20 20:38:40')`
Result	`{year: 1992, month: 9, day: 20}`

`datepart(part, timestamp)`

Description	Alias of `date_part`. Get subfield (equivalent to `extract`).
Example	`datepart('minute', TIMESTAMP '1992-09-20 20:38:40')`
Result	`38`

`datesub(part, startdate, enddate)`

Description	Alias of `date_sub`. The number of complete partitions between the timestamps.
Example	`datesub('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`
Result	`1`

`datetrunc(part, timestamp)`

Description	Alias of `date_trunc`. Truncate to specified precision.
Example	`datetrunc('hour', TIMESTAMP '1992-09-20 20:38:40')`
Result	`1992-09-20 20:00:00`

dayname(timestamp)

Description	The (English) name of the weekday.
Example	`dayname(TIMESTAMP '1992-03-22')`
Result	Sunday

epoch_ms(ms)

Description	Converts ms since epoch to a timestamp.
Example	`epoch_ms(701222400000)`
Result	`1992-03-22 00:00:00`

epoch_ms(timestamp)

Description	Converts a timestamp to milliseconds since the epoch.
Example	`epoch_ms('2022-11-07 08:43:04.123456'::TIMESTAMP);`
Result	`1667810584123`

epoch_ms(timestamp)

Description	Return the total number of milliseconds since the epoch.
Example	`epoch_ms(timestamp '2021-08-03 11:59:44.123456')`
Result	`1627991984123`

epoch_ns(timestamp)

Description	Return the total number of nanoseconds since the epoch.
Example	`epoch_ns(timestamp '2021-08-03 11:59:44.123456')`
Result	`1627991984123456000`

epoch_us(timestamp)

Description	Return the total number of microseconds since the epoch.
Example	`epoch_us(timestamp '2021-08-03 11:59:44.123456')`
Result	`1627991984123456`

epoch(timestamp)

Description	Converts a timestamp to seconds since the epoch.
Example	`epoch('2022-11-07 08:43:04'::TIMESTAMP);`
Result	`1667810584`

extract(field FROM timestamp)

Description	Get subfield from a timestamp.
Example	`extract('hour' FROM TIMESTAMP '1992-09-20 20:38:48')`
Result	`20`

greatest(timestamp, timestamp)

Description	The later of two timestamps.
Example	`greatest(TIMESTAMP '1992-09-20 20:38:48', TIMESTAMP '1992-03-22 01:02:03.1234')`
Result	`1992-09-20 20:38:48`

isfinite(timestamp)

Description	Returns true if the timestamp is finite, false otherwise.
Example	`isfinite(TIMESTAMP '1992-03-07')`
Result	`true`

isinf(timestamp)

Description	Returns true if the timestamp is infinite, false otherwise.
Example	`isinf(TIMESTAMP '-infinity')`
Result	`true`

last_day(timestamp)

Description	The last day of the month.
Example	`last_day(TIMESTAMP '1992-03-22 01:02:03.1234')`
Result	`1992-03-31`

least(timestamp, timestamp)

Description	The earlier of two timestamps.
Example	`least(TIMESTAMP '1992-09-20 20:38:48', TIMESTAMP '1992-03-22 01:02:03.1234')`
Result	`1992-03-22 01:02:03.1234`

make_timestamp(bigint, bigint, bigint, bigint, bigint, double)

Description	The timestamp for the given parts.
Example	`make_timestamp(1992, 9, 20, 13, 34, 27.123456)`
Result	`1992-09-20 13:34:27.123456`

make_timestamp(microseconds)

Description	The timestamp for the given number of µs since the epoch.
Example	`make_timestamp(1667810584123456)`
Result	`2022-11-07 08:43:04.123456`

monthname(timestamp)

Description	The (English) name of the month.
Example	`monthname(TIMESTAMP '1992-09-20')`
Result	`September`

strftime(timestamp, format)

Description	Converts timestamp to string according to the format string.
Example	`strftime(timestamp '1992-01-01 20:38:40', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`Wed, 1 January 1992 - 08:38:40 PM`

strptime(text, format-list)

Description	Converts string to timestamp applying the format strings in the list until one succeeds. Throws on failure.
Example	`strptime('4/15/2023 10:56:00', ['%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S'])`

Result	`2023-04-15 10:56:00`

`strptime(text, format)`

Description	Converts string to timestamp according to the format string. Throws on failure.
Example	`strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`1992-01-01 20:38:40`

`time_bucket(bucket_width, timestamp[, offset])`

Description	Truncate `timestamp` by the specified interval `bucket_width`. Buckets are offset by `offset` interval.
Example	`time_bucket(INTERVAL '10 minutes', TIMESTAMP '1992-04-20 15:26:00-07', INTERVAL '5 minutes')`
Result	`1992-04-20 15:25:00`

`time_bucket(bucket_width, timestamp[, origin])`

Description	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.
Example	`time_bucket(INTERVAL '2 weeks', TIMESTAMP '1992-04-20 15:26:00', TIMESTAMP '1992-04-01 00:00:00')`
Result	`1992-04-15 00:00:00`

`to_timestamp(double)`

Description	Converts seconds since the epoch to a timestamp with time zone.
Example	`to_timestamp(1284352323.5)`
Result	`2010-09-13 04:32:03.5+00`

`try_strptime(text, format-list)`

Description	Converts string to timestamp applying the format strings in the list until one succeeds. Returns `NULL` on failure.
Example	`try_strptime('4/15/2023 10:56:00', ['%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S'])`

Result	`2023-04-15 10:56:00`

`try_strptime(text, format)`

Description	Converts string to timestamp according to the format string. Returns `NULL` on failure.
Example	`try_strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`1992-01-01 20:38:40`

Timestamp Table Functions

The table below shows the available table functions for `TIMESTAMP` types.

Name	Description
`generate_series(timestamp, timestamp, interval)`	Generate a table of timestamps in the closed range, stepping by the interval.
`range(timestamp, timestamp, interval)`	Generate a table of timestamps in the half open range, stepping by the interval.

> Infinite values are not allowed as table function bounds.

`generate_series(timestamp, timestamp, interval)`

Description	Generate a table of timestamps in the closed range, stepping by the interval.
Example	`generate_series(TIMESTAMP '2001-04-10', TIMESTAMP '2001-04-11', INTERVAL 30 MINUTE)`

`range(timestamp, timestamp, interval)`

Description	Generate a table of timestamps in the half open range, stepping by the interval.
Example	`range(TIMESTAMP '2001-04-10', TIMESTAMP '2001-04-11', INTERVAL 30 MINUTE)`

Timestamp with Time Zone Functions

This section describes functions and operators for examining and manipulating `TIMESTAMP WITH TIME ZONE` (or `TIMESTAMPTZ`) 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 `TIMESTAMPTZ` values. Since these functions do not involve binning or display, they are always available.

Name	Description
`current_timestamp`	Current date and time (start of current transaction).
`get_current_timestamp()`	Current date and time (start of current transaction).
`greatest(timestamptz, timestamptz)`	The later of two timestamps.
`isfinite(timestamptz)`	Returns true if the timestamp with time zone is finite, false otherwise.
`isinf(timestamptz)`	Returns true if the timestamp with time zone is infinite, false otherwise.
`least(timestamptz, timestamptz)`	The earlier of two timestamps.
`now()`	Current date and time (start of current transaction).
`transaction_timestamp()`	Current date and time (start of current transaction).

`current_timestamp`

Description	Current date and time (start of current transaction).
Example	`current_timestamp`
Result	`2022-10-08 12:44:46.122-07`

`get_current_timestamp()`

Description	Current date and time (start of current transaction).
Example	`get_current_timestamp()`
Result	`2022-10-08 12:44:46.122-07`

`greatest(timestamptz, timestamptz)`

Description	The later of two timestamps.
Example	`greatest(TIMESTAMPTZ '1992-09-20 20:38:48', TIMESTAMPTZ '1992-03-22 01:02:03.1234')`
Result	`1992-09-20 20:38:48-07`

`isfinite(timestamptz)`

Description	Returns true if the timestamp with time zone is finite, false otherwise.
Example	`isfinite(TIMESTAMPTZ '1992-03-07')`
Result	`true`

`isinf(timestamptz)`

Description	Returns true if the timestamp with time zone is infinite, false otherwise.
Example	`isinf(TIMESTAMPTZ '-infinity')`
Result	`true`

`least(timestamptz, timestamptz)`

Description	The earlier of two timestamps.
Example	`least(TIMESTAMPTZ '1992-09-20 20:38:48', TIMESTAMPTZ '1992-03-22 01:02:03.1234')`
Result	`1992-03-22 01:02:03.1234-08`

`now()`

Description	Current date and time (start of current transaction).
Example	`now()`
Result	`2022-10-08 12:44:46.122-07`

`transaction_timestamp()`

Description	Current date and time (start of current transaction).
Example	`transaction_timestamp()`
Result	`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.

Name	Description
`age(timestamptz, timestamptz)`	Subtract arguments, resulting in the time difference between the two timestamps.
`age(timestamptz)`	Subtract from current_date.
`date_diff(part, startdate, enddate)`	The number of partition boundaries between the timestamps.
`date_part([part, ...], timestamptz)`	Get the listed subfields as a `struct`. The list must be constant.
`date_part(part, timestamptz)`	Get subfield (equivalent to *extract*).
`date_sub(part, startdate, enddate)`	The number of complete partitions between the timestamps.
`date_trunc(part, timestamptz)`	Truncate to specified precision.
`datediff(part, startdate, enddate)`	Alias of date_diff. The number of partition boundaries between the timestamps.
`datepart([part, ...], timestamptz)`	Alias of date_part. Get the listed subfields as a `struct`. The list must be constant.
`datepart(part, timestamptz)`	Alias of date_part. Get subfield (equivalent to *extract*).

Name	Description
`datesub(part, startdate, enddate)`	Alias of date_sub. The number of complete partitions between the timestamps.
`datetrunc(part, timestamptz)`	Alias of date_trunc. Truncate to specified precision.
`epoch_ms(timestamptz)`	Converts a timestamptz to milliseconds since the epoch.
`epoch_ns(timestamptz)`	Converts a timestamptz to nanoseconds since the epoch.
`epoch_us(timestamptz)`	Converts a timestamptz to microseconds since the epoch.
`extract(field FROM timestamptz)`	Get subfield from a `TIMESTAMP WITH TIME ZONE`.
`last_day(timestamptz)`	The last day of the month.
`make_timestamptz(bigint, bigint, bigint, bigint, bigint, double, string)`	The `TIMESTAMP WITH TIME ZONE` for the given parts and time zone.
`make_timestamptz(bigint, bigint, bigint, bigint, bigint, double)`	The `TIMESTAMP WITH TIME ZONE` for the given parts in the current time zone.
`make_timestamptz(microseconds)`	The `TIMESTAMP WITH TIME ZONE` for the given µs since the epoch.
`strftime(timestamptz, format)`	Converts a `TIMESTAMP WITH TIME ZONE` value to string according to the format string.
`strptime(text, format)`	Converts string to `TIMESTAMP WITH TIME ZONE` according to the format string if `%Z` is specified.
`time_bucket(bucket_width, timestamptz[, offset])`	Truncate `timestamptz` by the specified interval `bucket_width`. Buckets are offset by `offset` interval.
`time_bucket(bucket_width, timestamptz[, 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(bucket_width, timestamptz[, 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.

`age(timestamptz, timestamptz)`

Description	Subtract arguments, resulting in the time difference between the two timestamps.
Example	`age(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '1992-09-20')`
Result	`8 years 6 months 20 days`

`age(timestamptz)`

Description	Subtract from current_date.
Example	`age(TIMESTAMP '1992-09-20')`
Result	`29 years 1 month 27 days 12:39:00.844`

date_diff(part, startdate, enddate)

Description	The number of partition boundaries between the timestamps.
Example	`date_diff('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`
Result	2

date_part([part, ...], timestamptz)

Description	Get the listed subfields as a `struct`. The list must be constant.
Example	`date_part(['year', 'month', 'day'], TIMESTAMPTZ '1992-09-20 20:38:40-07')`
Result	`{year: 1992, month: 9, day: 20}`

date_part(part, timestamptz)

Description	Get subfield (equivalent to *extract*).
Example	`date_part('minute', TIMESTAMPTZ '1992-09-20 20:38:40')`
Result	38

date_sub(part, startdate, enddate)

Description	The number of complete partitions between the timestamps.
Example	`date_sub('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`
Result	1

date_trunc(part, timestamptz)

Description	Truncate to specified precision.
Example	`date_trunc('hour', TIMESTAMPTZ '1992-09-20 20:38:40')`
Result	1992-09-20 20:00:00

datediff(part, startdate, enddate)

Description	Alias of date_diff. The number of partition boundaries between the timestamps.
Example	`datediff('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`

Result	2

`datepart([part, ...], timestamptz)`

Description	Alias of date_part. Get the listed subfields as a `struct`. The list must be constant.
Example	`datepart(['year', 'month', 'day'], TIMESTAMPTZ '1992-09-20 20:38:40-07')`
Result	`{year: 1992, month: 9, day: 20}`

`datepart(part, timestamptz)`

Description	Alias of date_part. Get subfield (equivalent to *extract*).
Example	`datepart('minute', TIMESTAMPTZ '1992-09-20 20:38:40')`
Result	38

`datesub(part, startdate, enddate)`

Description	Alias of date_sub. The number of complete partitions between the timestamps.
Example	`datesub('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`
Result	1

`datetrunc(part, timestamptz)`

Description	Alias of date_trunc. Truncate to specified precision.
Example	`datetrunc('hour', TIMESTAMPTZ '1992-09-20 20:38:40')`
Result	`1992-09-20 20:00:00`

`epoch_ms(timestamptz)`

Description	Converts a timestamptz to milliseconds since the epoch.
Example	`epoch_ms('2022-11-07 08:43:04.123456+00'::TIMESTAMPTZ);`
Result	1667810584123

`epoch_ns(timestamptz)`

Description	Converts a timestamptz to nanoseconds since the epoch.
Example	`epoch_ns('2022-11-07 08:43:04.123456+00'::TIMESTAMPTZ);`
Result	`1667810584123456000`

`epoch_us(timestamptz)`

Description	Converts a timestamptz to microseconds since the epoch.
Example	`epoch_us('2022-11-07 08:43:04.123456+00'::TIMESTAMPTZ);`
Result	`1667810584123456`

`extract(field FROM timestamptz)`

Description	Get subfield from a `TIMESTAMP WITH TIME ZONE`.
Example	`extract('hour' FROM TIMESTAMPTZ '1992-09-20 20:38:48')`
Result	`20`

`last_day(timestamptz)`

Description	The last day of the month.
Example	`last_day(TIMESTAMPTZ '1992-03-22 01:02:03.1234')`
Result	`1992-03-31`

`make_timestamptz(bigint, bigint, bigint, bigint, bigint, double, string)`

Description	The `TIMESTAMP WITH TIME ZONE` for the given parts and time zone.
Example	`make_timestamptz(1992, 9, 20, 15, 34, 27.123456, 'CET')`
Result	`1992-09-20 06:34:27.123456-07`

`make_timestamptz(bigint, bigint, bigint, bigint, bigint, double)`

Description	The `TIMESTAMP WITH TIME ZONE` for the given parts in the current time zone.
Example	`make_timestamptz(1992, 9, 20, 13, 34, 27.123456)`
Result	`1992-09-20 13:34:27.123456-07`

`make_timestamptz(microseconds)`

Description	The `TIMESTAMP WITH TIME ZONE` for the given µs since the epoch.
Example	`make_timestamptz(1667810584123456)`
Result	`2022-11-07 16:43:04.123456-08`

`strftime(timestamptz, format)`

Description	Converts a `TIMESTAMP WITH TIME ZONE` value to string according to the format string.
Example	`strftime(timestamptz '1992-01-01 20:38:40', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`Wed, 1 January 1992 - 08:38:40 PM`

`strptime(text, format)`

Description	Converts string to `TIMESTAMP WITH TIME ZONE` according to the format string if `%Z` is specified.
Example	`strptime('Wed, 1 January 1992 - 08:38:40 PST', '%a, %-d %B %Y - %H:%M:%S %Z')`
Result	`1992-01-01 08:38:40-08`

`time_bucket(bucket_width, timestamptz[, offset])`

Description	Truncate `timestamptz` by the specified interval `bucket_width`. Buckets are offset by `offset` interval.
Example	`time_bucket(INTERVAL '10 minutes', TIMESTAMPTZ '1992-04-20 15:26:00-07', INTERVAL '5 minutes')`
Result	`1992-04-20 15:25:00-07`

`time_bucket(bucket_width, timestamptz[, origin])`

Description	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.
Example	`time_bucket(INTERVAL '2 weeks', TIMESTAMPTZ '1992-04-20 15:26:00-07', TIMESTAMPTZ '1992-04-01 00:00:00-07')`
Result	`1992-04-15 00:00:00-07`

`time_bucket(bucket_width, timestamptz[, timezone])`

Description	Truncate `timestamptz` by the specified interval `bucket_width`. Bucket starts and ends are calculated using `timezone`. `timezone` is a varchar and defaults to UTC.
Example	`time_bucket(INTERVAL '2 days', TIMESTAMPTZ '1992-04-20 15:26:00-07', 'Europe/Berlin')`
Result	`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.

Name	Description
`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.
`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.

> Infinite values are not allowed as table function bounds.

`generate_series(timestamptz, timestamptz, interval)`

Description	Generate a table of timestamps in the closed range (including both the starting timestamp and the ending timestamp), stepping by the interval.
Example	`generate_series(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '2001-04-11', INTERVAL 30 MINUTE)`

`range(timestamptz, timestamptz, interval)`

Description	Generate a table of timestamps in the half open range (including the starting timestamp, but stopping before the ending timestamp), stepping by the interval.
Example	`range(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '2001-04-11', INTERVAL 30 MINUTE)`

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.

Name	Description
current_localtime()	Returns a TIME whose GMT bin values correspond to local time in the current time zone.
current_localtimestamp()	Returns a TIMESTAMP whose GMT bin values correspond to local date and time in the current time zone.
localtime	Synonym for the current_localtime() function call.
localtimestamp	Synonym for the current_localtimestamp() function call.
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(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.

current_localtime()

Description	Returns a TIME whose GMT bin values correspond to local time in the current time zone.
Example	current_localtime()
Result	08:47:56.497

current_localtimestamp()

Description	Returns a TIMESTAMP whose GMT bin values correspond to local date and time in the current time zone.
Example	current_localtimestamp()
Result	2022-12-17 08:47:56.497

localtime

Description	Synonym for the current_localtime() function call.
Example	localtime
Result	08:47:56.497

localtimestamp

Description	Synonym for the current_localtimestamp() function call.
Example	localtimestamp
Result	2022-12-17 08:47:56.497

timezone(text, timestamp)

Description	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.
Example	`timezone('America/Denver', TIMESTAMP '2001-02-16 20:38:40')`
Result	`2001-02-16 19:38:40-08`

`timezone(text, timestamptz)`

Description	Use the date parts of the timestamp in the given time zone to construct a timestamp. Effectively, the result is a ”local” time.
Example	`timezone('America/Denver', TIMESTAMPTZ '2001-02-16 20:38:40-05')`
Result	`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

Scalar Utility Functions

The functions below are difficult to categorize into specific function types and are broadly useful.

Name	Description
`alias(column)`	Return the name of the column.
`checkpoint(database)`	Synchronize WAL with file for (optional) database without interrupting transactions.

Name	Description
`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.
`constant_or_null(arg1, arg2)`	If `arg2` is NULL, return NULL. Otherwise, return `arg1`.
`count_if(x)`	Returns 1 if `x` is `true` or a non-zero number.
`current_catalog()`	Return the name of the currently active catalog. Default is memory.
`current_schema()`	Return the name of the currently active schema. Default is main.
`current_schemas(boolean)`	Return list of schemas. Pass a parameter of `true` to include implicit schemas.
`current_setting('setting_name')`	Return the current value of the configuration setting.
`currval('sequence_name')`	Return the current value of the sequence. Note that `nextval` must be called at least once prior to calling `currval`.
`error(message)`	Throws the given error `message`.
`force_checkpoint(database)`	Synchronize WAL with file for (optional) database interrupting transactions.
`gen_random_uuid()`	Alias of `uuid`. Return a random UUID similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
`hash(value)`	Returns a UBIGINT with the hash of the `value`.
`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.
`ifnull(expr, other)`	A two-argument version of coalesce.
`md5(string)`	Return an MD5 hash of the `string`.
`nextval('sequence_name')`	Return the following value of the sequence.
`nullif(a, b)`	Return null if a = b, else return a. Equivalent to `CASE WHEN a = b THEN NULL ELSE a END`.
`pg_typeof(expression)`	Returns the lower case name of the data type of the result of the expression. For PostgreSQL compatibility.
`read_blob(source)`	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a BLOB. See the `read_blob` guide for more details.
`read_text(source)`	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a VARCHAR. The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead. See the `read_text` guide for more details.
`sha256(value)`	Returns a VARCHAR with the SHA-256 hash of the `value`.
`stats(expression)`	Returns a string with statistics about the expression. Expression can be a column, constant, or SQL expression.
`txid_current()`	Returns the current transaction's identifier, a BIGINT value. It will assign a new one if the current transaction does not have one already.
`typeof(expression)`	Returns the name of the data type of the result of the expression.
`uuid()`	Return a random UUID similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
`version()`	Return the currently active version of DuckDB in this format.

alias(column)

Description	Return the name of the column.
Example	`alias(column1)`
Result	`column1`

checkpoint(database)

Description	Synchronize WAL with file for (optional) database without interrupting transactions.
Example	`checkpoint(my_db)`
Result	success boolean

coalesce(expr, ...)

Description	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.
Example	`coalesce(NULL, NULL, 'default_string')`
Result	`default_string`

constant_or_null(arg1, arg2)

Description	If `arg2` is `NULL`, return `NULL`. Otherwise, return `arg1`.
Example	`constant_or_null(42, NULL)`
Result	`NULL`

count_if(x)

Description	Returns 1 if `x` is `true` or a non-zero number.
Example	`count_if(42)`
Result	1

current_catalog()

Description	Return the name of the currently active catalog. Default is memory.
Example	`current_catalog()`
Result	`memory`

`current_schema()`

Description	Return the name of the currently active schema. Default is main.
Example	`current_schema()`
Result	`main`

`current_schemas(boolean)`

Description	Return list of schemas. Pass a parameter of `true` to include implicit schemas.
Example	`current_schemas(true)`
Result	`['temp', 'main', 'pg_catalog']`

`current_setting('setting_name')`

Description	Return the current value of the configuration setting.
Example	`current_setting('access_mode')`
Result	`automatic`

`currval('sequence_name')`

Description	Return the current value of the sequence. Note that `nextval` must be called at least once prior to calling `currval`.
Example	`currval('my_sequence_name')`
Result	`1`

`error(message)`

Description	Throws the given error `message`.
Example	`error('access_mode')`

`force_checkpoint(database)`

Description	Synchronize WAL with file for (optional) database interrupting transactions.
Example	`force_checkpoint(my_db)`
Result	success boolean

`gen_random_uuid()`

Description	Alias of `uuid`. Return a random UUID similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
Example	`gen_random_uuid()`
Result	various

`hash(value)`

Description	Returns a `UBIGINT` with the hash of the `value`.
Example	`hash('🦆')`
Result	`2595805878642663834`

`icu_sort_key(string, collator)`

Description	Surrogate key used to sort special characters according to the specific locale. Collator parameter is optional. Valid only when ICU extension is installed.
Example	`icu_sort_key('ö', 'DE')`
Result	`460145960106`

`ifnull(expr, other)`

Description	A two-argument version of coalesce.
Example	`ifnull(NULL, 'default_string')`
Result	`default_string`

`md5(string)`

Description	Return an MD5 hash of the `string`.
Example	`md5('123')`
Result	`202cb962ac59075b964b07152d234b70`

`nextval('sequence_name')`

Description	Return the following value of the sequence.
Example	`nextval('my_sequence_name')`
Result	2

`nullif(a, b)`

Description	Return null if a = b, else return a. Equivalent to `CASE WHEN a = b THEN NULL ELSE a END`.
Example	`nullif(1+1, 2)`
Result	`NULL`

`pg_typeof(expression)`

Description	Returns the lower case name of the data type of the result of the expression. For PostgreSQL compatibility.
Example	`pg_typeof('abc')`
Result	`varchar`

`read_blob(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a BLOB. See the `read_blob` guide for more details.
Example	`read_blob('hello.bin')`
Result	`hello\x0A`

`read_text(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a VARCHAR. The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead. See the `read_text` guide for more details.
Example	`read_text('hello.txt')`
Result	`hello\n`

`sha256(value)`

Description	Returns a VARCHAR with the SHA-256 hash of the `value`.
Example	`sha256('🦆')`
Result	`d7a5c5e0d1d94c32218539e7e47d4ba9c3c7b77d61332fb60d633dde89e473fb`

`stats(expression)`

Description	Returns a string with statistics about the expression. Expression can be a column, constant, or SQL expression.
Example	`stats(5)`
Result	`'[Min: 5, Max: 5][Has Null: false]'`

`txid_current()`

Description	Returns the current transaction's identifier, a `BIGINT` value. It will assign a new one if the current transaction does not have one already.
Example	`txid_current()`
Result	various

`typeof(expression)`

Description	Returns the name of the data type of the result of the expression.
Example	`typeof('abc')`
Result	VARCHAR

`uuid()`

Description	Return a random UUID similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
Example	`uuid()`
Result	various

`version()`

Description	Return the currently active version of DuckDB in this format.
Example	`version()`
Result	various

Utility Table Functions

A table function is used in place of a table in a `FROM` clause.

Name	Description
`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.

Name	Description
`repeat_row(varargs, num_rows)`	Returns a table with `num_rows` rows, each containing the fields defined in `varargs`.

`glob(search_path)`

Description	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.
Example	`glob('*')`
Result	(table of filenames)

`repeat_row(varargs, num_rows)`

Description	Returns a table with `num_rows` rows, each containing the fields defined in `varargs`.
Example	`repeat_row(1, 2, 'foo', num_rows = 3)`
Result	3 rows of `1, 2, 'foo'`

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;
```

Returns the amount of the first sale using the `first()` aggregate function:

```
SELECT first(amount ORDER BY date ASC) 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.

`DISTINCT` Clause in Aggregate Functions

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.

`ORDER BY` Clause in Aggregate Functions

An `ORDER BY` clause can be provided after the last argument of the function call. Note the lack of the comma separator before the clause.

```
SELECT <aggregate_function>(<arg>, <sep> ORDER BY <ordering_criteria>);
```

This clause ensures that the values being aggregated are sorted before applying the function. Most aggregate functions are order-insensitive, and for them this clause is parsed and discarded. However, there are some order-sensitive aggregates that can have non-deterministic results without ordering, e.g., `first`, `last`, `list` and `string_agg` / `group_concat` / `listagg`. These can be made deterministic by ordering the arguments.

For example:

```
CREATE TABLE tbl AS SELECT s FROM range(1, 4) r(s);
SELECT string_agg(s, ', ' ORDER BY s DESC) AS countdown FROM tbl;
```

countdown
3, 2, 1

General Aggregate Functions

The table below shows the available general aggregate functions.

Function	Description
`any_value(arg)`	Returns the first non-null value from `arg`. This function is affected by ordering.
`arbitrary(arg)`	Returns the first value (null or non-null) from `arg`. This function is affected by ordering.
`arg_max(arg, val)`	Finds the row with the maximum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.
`arg_min(arg, val)`	Finds the row with the minimum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.
`array_agg(arg)`	Returns a `LIST` containing all the values of a column. This function is affected by ordering.
`avg(arg)`	Calculates the average value for all tuples in `arg`.
`bit_and(arg)`	Returns the bitwise AND of all bits in a given expression.
`bit_or(arg)`	Returns the bitwise OR of all bits in a given expression.
`bit_xor(arg)`	Returns the bitwise XOR of all bits in a given expression.
`bitstring_agg(arg)`	Returns a bitstring with bits set for each distinct value.
`bool_and(arg)`	Returns `true` if every input value is `true`, otherwise `false`.
`bool_or(arg)`	Returns `true` if any input value is `true`, otherwise `false`.
`count(arg)`	Calculates the number of tuples in `arg`.
`favg(arg)`	Calculates the average using a more accurate floating point summation (Kahan Sum).
`first(arg)`	Returns the first value (null or non-null) from `arg`. This function is affected by ordering.
`fsum(arg)`	Calculates the sum using a more accurate floating point summation (Kahan Sum).
`geomean(arg)`	Calculates the geometric mean for all tuples in `arg`.
`histogram(arg)`	Returns a `MAP` of key-value pairs representing buckets and counts.
`last(arg)`	Returns the last value of a column. This function is affected by ordering.
`list(arg)`	Returns a `LIST` containing all the values of a column. This function is affected by ordering.
`max(arg)`	Returns the maximum value present in `arg`.
`max_by(arg, val)`	Finds the row with the maximum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.

Function	Description
`min(arg)`	Returns the minimum value present in `arg`.
`min_by(arg, val)`	Finds the row with the minimum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.
`product(arg)`	Calculates the product of all tuples in `arg`.
`string_agg(arg, sep)`	Concatenates the column string values with a separator. This function is affected by ordering.
`sum(arg)`	Calculates the sum value for all tuples in `arg`.
`sum_no_overflow(arg)`	Calculates the sum value for all tuples in `arg` without overflow checks. Unlike `sum`, which works on floating-point values, `sum_no_overflow` only accepts `INTEGER` and `DECIMAL` values.

`any_value(arg)`

Description	Returns the first non-null value from `arg`. This function is affected by ordering.
Example	`any_value(A)`
Alias(es)	-

`arbitrary(arg)`

Description	Returns the first value (null or non-null) from `arg`. This function is affected by ordering.
Example	`arbitrary(A)`
Alias(es)	`first(A)`

`arg_max(arg, val)`

Description	Finds the row with the maximum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.
Example	`arg_max(A, B)`
Alias(es)	`argMax(arg, val)`,`max_by(arg, val)`

`arg_min(arg, val)`

Description	Finds the row with the minimum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.
Example	`arg_min(A, B)`
Alias(es)	`argMin(arg, val)`,`min_by(arg, val)`

`array_agg(arg)`

Description	Returns a `LIST` containing all the values of a column. This function is affected by ordering.
Example	`array_agg(A)`
Alias(es)	`list`

`avg(arg)`

Description	Calculates the average value for all tuples in `arg`.
Example	`avg(A)`
Alias(es)	`mean`

`bit_and(arg)`

Description	Returns the bitwise AND of all bits in a given expression.
Example	`bit_and(A)`
Alias(es)	-

`bit_or(arg)`

Description	Returns the bitwise OR of all bits in a given expression.
Example	`bit_or(A)`
Alias(es)	-

`bit_xor(arg)`

Description	Returns the bitwise XOR of all bits in a given expression.
Example	`bit_xor(A)`
Alias(es)	-

`bitstring_agg(arg)`

Description	Returns a bitstring with bits set for each distinct value.
Example	`bitstring_agg(A)`
Alias(es)	-

`bool_and(arg)`

Description	Returns `true` if every input value is `true`, otherwise `false`.
Example	`bool_and(A)`
Alias(es)	-

`bool_or(arg)`

Description	Returns `true` if any input value is `true`, otherwise `false`.
Example	`bool_or(A)`
Alias(es)	-

`count(arg)`

Description	Calculates the number of tuples in `arg`.
Example	`count(A)`
Alias(es)	-

`favg(arg)`

Description	Calculates the average using a more accurate floating point summation (Kahan Sum).
Example	`favg(A)`
Alias(es)	-

`first(arg)`

Description	Returns the first value (null or non-null) from `arg`. This function is affected by ordering.
Example	`first(A)`
Alias(es)	`arbitrary(A)`

`fsum(arg)`

Description	Calculates the sum using a more accurate floating point summation (Kahan Sum).
Example	`fsum(A)`
Alias(es)	`sumKahan`, `kahan_sum`

`geomean(arg)`

Description	Calculates the geometric mean for all tuples in `arg`.
Example	`geomean(A)`
Alias(es)	`geometric_mean(A)`

`histogram(arg)`

Description	Returns a MAP of key-value pairs representing buckets and counts.
Example	`histogram(A)`
Alias(es)	-

`last(arg)`

Description	Returns the last value of a column. This function is affected by ordering.
Example	`last(A)`
Alias(es)	-

`list(arg)`

Description	Returns a `LIST` containing all the values of a column. This function is affected by ordering.
Example	`list(A)`
Alias(es)	`array_agg`

`max(arg)`

Description	Returns the maximum value present in `arg`.
Example	`max(A)`
Alias(es)	-

`max_by(arg, val)`

Description	Finds the row with the maximum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.
Example	`max_by(A, B)`
Alias(es)	`argMax(arg, val)`, `arg_max(arg, val)`

`min(arg)`

Description	Returns the minimum value present in `arg`.
Example	`min(A)`
Alias(es)	-

`min_by(arg, val)`

Description	Finds the row with the minimum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.
Example	`min_by(A, B)`
Alias(es)	`argMin(arg, val)`,`arg_min(arg, val)`

`product(arg)`

Description	Calculates the product of all tuples in `arg`.
Example	`product(A)`
Alias(es)	-

`string_agg(arg, sep)`

Description	Concatenates the column string values with a separator. This function is affected by ordering.
Example	`string_agg(S, ',')`
Alias(es)	`group_concat(arg, sep)`,`listagg(arg, sep)`

`sum(arg)`

Description	Calculates the sum value for all tuples in `arg`.
Example	`sum(A)`
Alias(es)	-

`sum_no_overflow(arg)`

Description	Calculates the sum value for all tuples in `arg` without overflow checks. Unlike `sum`, which works on floating-point values, `sum_no_overflow` only accepts `INTEGER` and `DECIMAL` values.
Example	`sum_no_overflow(A)`
Alias(es)	-

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)`
`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
`corr(y, x)`	Returns the correlation coefficient for non-null pairs in a group.
`covar_pop(y, x)`	Returns the population covariance of input values.
`covar_samp(y, x)`	Returns the sample covariance for non-null pairs in a group.
`entropy(x)`	Returns the log-2 entropy of count input-values.
`kurtosis_pop(x)`	Returns the excess kurtosis (Fisher's definition) of all input values. Bias correction is not applied.
`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(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.
`mode(x)`	Returns the most frequent value for the values within x. NULL values are ignored.
`quantile_cont(x, pos)`	Returns the interpolated `pos`-quantile of `x` for `0 <= pos <= 1`, i.e., orders the values of `x` and returns the `pos * (n_nonnull_values - 1)`th (zero-indexed) element (or an interpolation between the adjacent elements if the index is not an integer). 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 discrete `pos`-quantile of `x` for `0 <= pos <= 1`, i.e., orders the values of `x` and returns the `floor(pos * (n_nonnull_values - 1))`th (zero-indexed) element. If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding discrete quantiles.
`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.
`regr_count(y, x)`	Returns the number of non-null number pairs in a group.
`regr_intercept(y, x)`	Returns the intercept of the univariate linear regression line for non-null pairs in a group.
`regr_r2(y, x)`	Returns the coefficient of determination for non-null pairs in a group.

Function	Description
`regr_slope(y, x)`	Returns the slope of the linear regression line for non-null pairs in a group.
`regr_sxx(y, x)`	-
`regr_sxy(y, x)`	Returns the population covariance of input values.
`regr_syy(y, x)`	-
`skewness(x)`	Returns the skewness of all input values.
`stddev_pop(x)`	Returns the population standard deviation.
`stddev_samp(x)`	Returns the sample standard deviation.
`var_pop(x)`	Returns the population variance.
`var_samp(x)`	Returns the sample variance of all input values.

`corr(y, x)`

Description	Returns the correlation coefficient for non-NULL pairs in a group.
Formula	`covar_pop(y, x) / (stddev_pop(x) * stddev_pop(y))`
Alias(es)	-

`covar_pop(y, x)`

Description	Returns the population covariance of input values.
Formula	`(sum(x*y) - sum(x) * sum(y) / count(*)) / count(*)`
Alias(es)	-

`covar_samp(y, x)`

Description	Returns the sample covariance for non-NULL pairs in a group.
Formula	`(sum(x*y) - sum(x) * sum(y) / count(*)) / (count(*) - 1)`
Alias(es)	-

`entropy(x)`

Description	Returns the log-2 entropy of count input-values.
Formula	-
Alias(es)	-

`kurtosis_pop(x)`

Description	Returns the excess kurtosis (Fisher's definition) of all input values. Bias correction is not applied.
Formula	-
Alias(es)	-

`kurtosis(x)`

Description	Returns the excess kurtosis (Fisher's definition) of all input values, with a bias correction according to the sample size.
Formula	-
Alias(es)	-

`mad(x)`

Description	Returns the median absolute deviation for the values within x. `NULL` values are ignored. Temporal types return a positive `INTERVAL`.
Formula	`median(abs(x - median(x)))`
Alias(es)	-

`median(x)`

Description	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.
Formula	`quantile_cont(x, 0.5)`
Alias(es)	-

`mode(x)`

Description	Returns the most frequent value for the values within x. `NULL` values are ignored.
Formula	-
Alias(es)	-

`quantile_cont(x, pos)`

Description	Returns the interpolated `pos`-quantile of x for `0 <= pos <= 1`, i.e., orders the values of x and returns the `pos * (n_nonnull_values - 1)`th (zero-indexed) element (or an interpolation between the adjacent elements if the index is not an integer). If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding interpolated quantiles.
Formula	-

Alias(es)	-

`quantile_disc(x, pos)`

Description	Returns the discrete `pos`-quantile of `x` for `0 <= pos <= 1`, i.e., orders the values of `x` and returns the `floor(pos * (n_nonnull_values - 1))`th (zero-indexed) element. If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding discrete quantiles.
Formula	-
Alias(es)	`quantile`

`regr_avgx(y, x)`

Description	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.
Formula	-
Alias(es)	-

`regr_avgy(y, x)`

Description	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.
Formula	-
Alias(es)	-

`regr_count(y, x)`

Description	Returns the number of non-null number pairs in a group.
Formula	`(sum(x*y) - sum(x) * sum(y) / count(*)) / count(*)`
Alias(es)	-

`regr_intercept(y, x)`

Description	Returns the intercept of the univariate linear regression line for non-null pairs in a group.
Formula	`avg(y) - regr_slope(y, x) * avg(x)`
Alias(es)	-

`regr_r2(y, x)`

Description	Returns the coefficient of determination for non-null pairs in a group.
Formula	-
Alias(es)	-

`regr_slope(y, x)`

Description	Returns the slope of the linear regression line for non-null pairs in a group.
Formula	`covar_pop(x, y) / var_pop(x)`
Alias(es)	-

`regr_sxx(y, x)`

Description	-
Formula	`regr_count(y, x) * var_pop(x)`
Alias(es)	-

`regr_sxy(y, x)`

Description	Returns the population covariance of input values.
Formula	`regr_count(y, x) * covar_pop(y, x)`
Alias(es)	-

`regr_syy(y, x)`

Description	-
Formula	`regr_count(y, x) * var_pop(y)`
Alias(es)	-

`skewness(x)`

Description	Returns the skewness of all input values.
Formula	-
Alias(es)	-

`stddev_pop(x)`

Description	Returns the population standard deviation.
Formula	`sqrt(var_pop(x))`
Alias(es)	-

`stddev_samp(x)`

Description	Returns the sample standard deviation.
Formula	`sqrt(var_samp(x))`
Alias(es)	`stddev(x)`

`var_pop(x)`

Description	Returns the population variance.
Formula	-
Alias(es)	-

`var_samp(x)`

Description	Returns the sample variance of all input values.
Formula	`(sum(x^2) - sum(x)^2 / count(x)) / (count(x) - 1)`
Alias(es)	`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 column [(ASC|DESC)])	mode(column ORDER BY column [(ASC|DESC)])
percentile_cont(fraction) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_cont(column, fraction ORDER BY column [(ASC|DESC)])
percentile_cont(fractions) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_cont(column, fractions ORDER BY column [(ASC|DESC)])
percentile_disc(fraction) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_disc(column, fraction ORDER BY column [(ASC|DESC)])

Function	Equivalent
percentile_disc(fractions) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_disc(column, fractions ORDER BY column [(ASC|DESC)])

Miscellaneous Aggregate Functions

Function	Description	Alias
`grouping()`	For queries with GROUP BY and either `ROLLUP` or `GROUPING SETS`: Returns an integer identifying which of the argument expressions where used to group on to create the current supper-aggregate row.	`grouping_id()`

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.

> **Warning.** Constraints have a strong impact on performance: they slow down loading and updates but speed up certain queries. Please consult the Performance Guide for details.

Syntax

Check Constraint

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 Constraint

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 and Unique Constraint

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
```

```
CREATE TABLE students (id INTEGER, name VARCHAR, PRIMARY KEY (id, name));
INSERT INTO students VALUES (1, 'Student 1');
INSERT INTO students VALUES (1, 'Student 2');
INSERT INTO students VALUES (1, 'Student 1');
```

```
Constraint Error: Duplicate key "id: 1, name: Student 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`.

```
CREATE TABLE students(id INTEGER PRIMARY KEY, name VARCHAR, email VARCHAR UNIQUE);
INSERT INTO students VALUES (1, 'Student 1', 'student1@uni.com');
INSERT INTO students values (2, 'Student 2', 'student1@uni.com');
```

```
Constraint Error: Duplicate key "email: student1@uni.com" violates unique constraint.
```

```
INSERT INTO students(id, name) VALUES (3, 'Student 3');
INSERT INTO students(name, email) VALUES ('Student 3', 'student3@uni.com');
```

```
Constraint Error: NOT NULL constraint failed: students.id
```

> **Warning.** Indexes have certain limitations that might result in constraints being evaluated too eagerly, see the indexes section for more details.

Foreign Keys

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.

> **Warning.** 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`.

> **Warning.** 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` and `DROP INDEX`

To create an index, use the `CREATE INDEX` statement. To drop an index, use the `DROP INDEX` statement.

Limitations of ART Indexes

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.

> As expected, indexes have a strong effect on performance, slowing down loading and updates, but speeding up certain queries. Please consult the Performance Guide for details.

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.
If this is an unexpected constraint violation please double check with the known index limitations section
in our documentation (https://duckdb.org/docs/sql/indexes).
```

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. For example, in the following query, `SET id = 1` causes a `Constraint Error` to occur.

```
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO students VALUES (1, 'Student 1');

UPDATE students
SET id = 1, name = 'Student 2'
WHERE id = 1;
```

```
Constraint Error: Duplicate key "id: 1" violates primary key constraint.
If this is an unexpected constraint violation please double check with the known index limitations section
in our documentation (https://duckdb.org/docs/sql/indexes).
```

Currently, this is an expected limitation of DuckDB – 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. DuckDB's implementation is based on PostgreSQL's information schema.

Tables

`information_schema.schemata`: 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	`'my_db'`
`schema_name`	Name of the schema.	VARCHAR	`'main'`
`schema_owner`	Name of the owner of the schema. Not yet implemented.	VARCHAR	`'duckdb'`
`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`

`information_schema.tables`: 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	`'my_db'`
`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`

Column	Description	Type	Example
`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`
`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'`

`information_schema.columns`: 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 (always the current database).	`VARCHAR`	`'my_db'`
`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)'`
`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`

Column	Description	Type	Example
`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`
`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`

`information_schema.character_sets`: Character Sets

Column	Description	Type	Example
`character_set_catalog`	Currently not implemented – always `NULL`.	`VARCHAR`	`NULL`
`character_set_schema`	Currently not implemented – always `NULL`.	`VARCHAR`	`NULL`
`character_set_name`	Name of the character set, currently implemented as showing the name of the database encoding.	`VARCHAR`	`'UTF8'`
`character_repertoire`	Character repertoire, showing UCS if the encoding is `UTF8`, else just the encoding name.	`VARCHAR`	`'UCS'`
`form_of_use`	Character encoding form, same as the database encoding.	`VARCHAR`	`'UTF8'`
`default_collate_catalog`	Name of the database containing the default collation (always the current database).	`VARCHAR`	`'my_db'`
`default_collate_schema`	Name of the schema containing the default collation.	`VARCHAR`	`'pg_catalog'`
`default_collate_name`	Name of the default collation.	`VARCHAR`	`'ucs_basic'`

`information_schema.key_column_usage`: Key Column Usage

Column	Description	Type	Example
constraint_catalog	Name of the database that contains the constraint (always the current database).	VARCHAR	'my_db'
constraint_schema	Name of the schema that contains the constraint.	VARCHAR	'main'
constraint_name	Name of the constraint.	VARCHAR	'exams_exam_id_fkey'
table_catalog	Name of the database that contains the table that contains the column that is restricted by this constraint (always the current database).	VARCHAR	'my_db'
table_schema	Name of the schema that contains the table that contains the column that is restricted by this constraint.	VARCHAR	'main'
table_name	Name of the table that contains the column that is restricted by this constraint.	VARCHAR	'exams'
column_name	Name of the column that is restricted by this constraint.	VARCHAR	'exam_id'
ordinal_position	Ordinal position of the column within the constraint key (count starts at 1).	INTEGER	1
position_in_unique_constraint	For a foreign-key constraint, ordinal position of the referenced column within its unique constraint (count starts at 1); otherwise NULL.	INTEGER	1

`information_schema.referential_constraints`: Referential Constraints

Column	Description	Type	Example
constraint_catalog	Name of the database containing the constraint (always the current database).	VARCHAR	'my_db'
constraint_schema	Name of the schema containing the constraint.	VARCHAR	main
constraint_name	Name of the constraint.	VARCHAR	exam_id_students_id_fkey
unique_constraint_catalog	Name of the database that contains the unique or primary key constraint that the foreign key constraint references.	VARCHAR	'my_db'
unique_constraint_schema	Name of the schema that contains the unique or primary key constraint that the foreign key constraint references.	VARCHAR	'main'
unique_constraint_name	Name of the unique or primary key constraint that the foreign key constraint references.	VARCHAR	'students_id_pkey'
match_option	Match option of the foreign key constraint. Always NONE.	VARCHAR	NONE
update_rule	Update rule of the foreign key constraint. Always NO ACTION.	VARCHAR	NO ACTION

Column	Description	Type	Example
`delete_rule`	Delete rule of the foreign key constraint. Always `NO ACTION`.	`VARCHAR`	`NO ACTION`

`information_schema.table_constraints`: Table Constraints

Column	Description	Type	Example
`constraint_catalog`	Name of the database that contains the constraint (always the current database).	`VARCHAR`	`'my_db'`
`constraint_schema`	Name of the schema that contains the constraint.	`VARCHAR`	`'main'`
`constraint_name`	Name of the constraint.	`VARCHAR`	`'exams_exam_id_fkey'`
`table_catalog`	Name of the database that contains the table (always the current database).	`VARCHAR`	`'my_db'`
`table_schema`	Name of the schema that contains the table.	`VARCHAR`	`'main'`
`table_name`	Name of the table.	`VARCHAR`	`'exams'`
`constraint_type`	Type of the constraint: `CHECK`, `FOREIGN KEY`, `PRIMARY KEY`, or `UNIQUE`.	`VARCHAR`	`'FOREIGN KEY'`
`is_deferrable`	`YES` if the constraint is deferrable, `NO` if not.	`VARCHAR`	`'NO'`
`initially_deferred`	`YES` if the constraint is deferrable and initially deferred, `NO` if not.	`VARCHAR`	`'NO'`
`enforced`	Always `YES`.	`VARCHAR`	`'YES'`
`nulls_distinct`	If the constraint is a unique constraint, then `YES` if the constraint treats nulls as distinct or `NO` if it treats nulls as not distinct, otherwise `NULL` for other types of constraints.	`VARCHAR`	`'YES'`

Catalog Functions

Several functions are also provided to see details about the catalogs and schemas that are configured in the database.

Function	Description	Example	Result
`current_catalog()`	Return the name of the currently active catalog. Default is memory.	`current_catalog()`	`'memory'`
`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.

> 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:

The `duckdb_views()` *table function* returns all views, including those marked internal:

```
SELECT * FROM duckdb_views();
```

The `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`
`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	Description	Type
`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`
`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`
`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`

Column	Description	Type
`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`
`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`.	`VARCHAR`

`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`
`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 (including macros) 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[]`
`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
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, foreign keys, and UNIQUE constraints 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_memory

The duckdb_memory() function provides metadata about DuckDB's buffer manager.

Column	Description	Type
tag	The memory tag. It has one of the following values: BASE_TABLE, HASH_TABLE, PARQUET_READER, CSV_READER, ORDER_BY, ART_INDEX, COLUMN_DATA, METADATA, OVERFLOW_STRINGS, IN_MEMORY_TABLE, ALLOCATOR, EXTENSION.	VARCHAR
memory_usage_bytes	The memory used (in bytes).	BIGINT

Column	Description	Type
`temporary_storage_bytes`	The disk storage used (in bytes).	`BIGINT`

`duckdb_optimizers`

The `duckdb_optimizers()` function provides metadata about the optimization rules (e.g., `expression_rewriter`, `filter_pushdown`) available in the DuckDB instance. These can be selectively turned off using `PRAGMA disabled_optimizers`.

Column	Description	Type
`name`	The name of the optimization rule.	`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`
`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_secrets`

The `duckdb_secrets()` function provides metadata about the secrets available in the DuckDB instance.

Column	Description	Type
`name`	The name of the secret.	`VARCHAR`
`type`	The type of the secret, e.g., `S3`, `GCS`, `R2`, `AZURE`.	`VARCHAR`
`provider`	The provider of the secret.	`VARCHAR`
`persistent`	Denotes whether the secret is persisent.	`BOOLEAN`
`storage`	The backend for storing the secret.	`VARCHAR`
`scope`	The scope of the secret.	`VARCHAR[]`
`secret_string`	Returns the content of the secret as a string. Sensitive pieces of information, e.g., they access key, are redacted.	`VARCHAR`

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`
`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.

> 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`

Column	Description	Type
`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`
`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_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.	`BIGINT`

`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`
`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_names`.	`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`
`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`

Column	Description	Type
`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`.

Keywords and Identifiers

Identifiers

Similarly to other SQL dialects and programming languages, identifiers in DuckDB's SQL are subject to several rules.

- Unquoted identifiers need to conform to a number of rules:
  - They must not be a reserved keyword (see `duckdb_keywords()`), e.g., `SELECT 123 AS SELECT` will fail.
  - They must not start with a number or special character, e.g., `SELECT 123 AS 1col` is invalid.
  - They cannot contain whitespaces (including tabs and newline characters).
- Identifiers can be quoted using double-quote characters (`"`). Quoted identifiers can use any keyword, whitespace or special character, e.g., `"SELECT"` and `" § 🦆 ¶ "` are valid identifiers.
- Double quotes can be escaped by repeating the quote character, e.g., to create an identifier named `IDENTIFIER "X"`, use `"IDENTIFIER ""X"""`.

Deduplicating Identifiers

In some cases, duplicate identifiers can occur, e.g., column names may conflict when unnesting a nested data structure. In these cases, DuckDB automatically deduplicates column names by renaming them according to the following rules:

- For a column named `<name>`, the first instance is not renamed.
- Subsequent instances are renamed to `<name>_<count>`, where `<count>` starts at 1.

For example:

```
SELECT *
FROM (SELECT UNNEST({'a': 42, 'b': {'a': 88, 'b': 99}}, recursive := true));
```

a	a_1	b
42	88	99

Database Names

Database names are subject to the rules for identifiers.

Additionally, it is best practice to avoid DuckDB's two internal database schema names, `system` and `temp`. By default, persistent databases are named after their filename without the extension. Therefore, the filenames `system.db` and `temp.db` (as well as `system.duckdb` and `temp.duckdb`) result in the database names `system` and `temp`, respectively. If you need to attach to a database that has one of these names, use an alias, e.g.:

```
ATTACH 'temp.db' AS temp2;
USE temp2;
```

Rules for Case-Sensitivity

Keywords and Function Names

SQL keywords and function names are case-insensitive in DuckDB.

For example, the following two queries are equivalent:

```
select COS(Pi()) as CosineOfPi;
SELECT cos(pi()) AS CosineOfPi;
```

CosineOfPi
-1.0

Case-Sensitivity of Identifiers

Identifiers in DuckDB are always case-insensitive, similarly to PostgreSQL. However, unlike PostgreSQL (and some other major SQL implementation), DuckDB also treats quoted identifiers as case-sensitive.

Despite treating identifiers in a case-insensitive manner, each character's case (uppercase/lowercase) is maintained as originally specified by the user even if a query uses different cases when referring to the identifier. For example:

```
CREATE TABLE tbl AS SELECT cos(pi()) AS CosineOfPi;
SELECT cosineofpi FROM tbl;
```

CosineOfPi
-1.0

To change this behavior, set the `preserve_identifier_case` configuration option to `false`.

Handling Conflicts

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 INTEGER, x INTEGER);
CREATE TABLE t2 (IdField INTEGER, y INTEGER);
INSERT INTO t1 VALUES (1, 123);
INSERT INTO t2 VALUES (1, 456);
SELECT * FROM t1 NATURAL JOIN t2;
```

idfield	x	y
1	123	456

Disabling Preserving Cases

With the `preserve_identifier_case` configuration option set to `false`, all identifiers are turned into lowercase:

```
SET preserve_identifier_case = false;
CREATE TABLE tbl AS SELECT cos(pi()) AS CosineOfPi;
SELECT CosineOfPi FROM tbl;
```

cosineofpi
-1.0

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., `SET 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.

> **Best practice.** 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

DuckDB supports window functions, which can use multiple rows to calculate a value for each row. Window functions are blocking operators, i.e., they require their entire input to be buffered, making them one of the most memory-intensive operators in SQL.

Examples

Generate a `row_number` column with 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.

Name	Description
`cume_dist()`	The cumulative distribution: (number of partition rows preceding or peer with current row) / total partition rows.
`dense_rank()`	The rank of the current row *without gaps;* this function counts peer groups.
`first(expr[ IGNORE NULLS])`	Returns `expr` evaluated at the row that is the first row (with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.

Name	Description
`first_value(expr[ IGNORE NULLS])`	Returns `expr` evaluated at the row that is the first row (with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.
`lag(expr[, offset[, default]][ IGNORE NULLS])`	Returns `expr` evaluated at the row that is `offset` rows (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) before the current row within the window frame; 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`.
`last(expr[ IGNORE NULLS])`	Returns `expr` evaluated at the row that is the last row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.
`last_value(expr[ IGNORE NULLS])`	Returns `expr` evaluated at the row that is the last row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.
`lead(expr[, offset[, default]][ IGNORE NULLS])`	Returns `expr` evaluated at the row that is `offset` rows after the current row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) within the window frame; 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`.
`nth_value(expr, nth[ IGNORE NULLS])`	Returns `expr` evaluated at the nth row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame (counting from 1); `NULL` if no such row.
`ntile(num_buckets)`	An integer ranging from 1 to `num_buckets`, dividing the partition as equally as possible.
`percent_rank()`	The relative rank of the current row: `(rank() - 1) / (total partition rows - 1)`.
`rank_dense()`	The rank of the current row *with gaps;* same as `row_number` of its first peer.
`rank()`	The rank of the current row *with gaps;* same as `row_number` of its first peer.
`row_number()`	The number of the current row within the partition, counting from 1.

`cume_dist()`

Description	The cumulative distribution: (number of partition rows preceding or peer with current row) / total partition rows.
Return Type	`DOUBLE`
Example	`cume_dist()`

`dense_rank()`

Description	The rank of the current row *without gaps;* this function counts peer groups.
Return Type	`BIGINT`
Example	`dense_rank()`

`first(expr[ IGNORE NULLS])`

Description	Returns `expr` evaluated at the row that is the first row (with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.
Return Type	Same type as `expr`
Example	`first(column)`
Alias	`first_value(column)`

`first_value(expr[ IGNORE NULLS])`

Description	Returns `expr` evaluated at the row that is the first row (with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.
Return Type	Same type as `expr`
Example	`first_value(column)`
Alias	`first(column)`

`lag(expr[, offset[, default]][ IGNORE NULLS])`

Description	Returns `expr` evaluated at the row that is `offset` rows (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) before the current row within the window frame; 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`.
Return Type	Same type as `expr`
Aliases	`lag(column, 3, 0)`

`last(expr[ IGNORE NULLS])`

Description	Returns `expr` evaluated at the row that is the last row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.
Return Type	Same type as `expr`
Example	`last(column)`
Alias	`last_value(column)`

`last_value(expr[ IGNORE NULLS])`

Description	Returns `expr` evaluated at the row that is the last row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.
Return Type	Same type as `expr`
Example	`last_value(column)`
Alias	`last(column)`

`lead(expr[, offset[, default]][ IGNORE NULLS])`

Description	Returns `expr` evaluated at the row that is `offset` rows after the current row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) within the window frame; 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`.
Return Type	Same type as `expr`
Aliases	`lead(column, 3, 0)`

`nth_value(expr, nth[ IGNORE NULLS])`

Description	Returns `expr` evaluated at the nth row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame (counting from 1); `NULL` if no such row.
Return Type	Same type as `expr`
Aliases	`nth_value(column, 2)`

`ntile(num_buckets)`

Description	An integer ranging from 1 to `num_buckets`, dividing the partition as equally as possible.
Return Type	`BIGINT`
Example	`ntile(4)`

`percent_rank()`

Description	The relative rank of the current row: `(rank() - 1) / (total partition rows - 1)`.
Return Type	`DOUBLE`
Example	`percent_rank()`

`rank_dense()`

Description	The rank of the current row *with gaps;* same as `row_number` of its first peer.
Return Type	`BIGINT`
Example	`rank_dense()`
Alias	`rank()`

`rank()`

Description	The rank of the current row *with gaps;* same as `row_number` of its first peer.

Return Type	`BIGINT`
Example	`rank()`
Alias	`rank_dense()`

`row_number()`

Description	The number of the current row within the partition, counting from 1.
Return Type	`BIGINT`
Example	`row_number()`

Aggregate Window Functions

All aggregate functions can be used in a windowing context, including the optional `FILTER` clause. The `first` and `last` aggregate functions are shadowed by the respective general-purpose window functions, with the minor consequence that the `FILTER` clause is not available for these but `IGNORE NULLS` is.

Nulls

All general-purpose window functions that accept `IGNORE NULLS` respect nulls by default. This default behavior can optionally be made explicit via `RESPECT NULLS`.

In contrast, all aggregate window functions (except for `list` and its aliases, which can be made to ignore nulls via a `FILTER`) ignore nulls and do not accept `RESPECT NULLS`. For example, `sum(column) OVER (ORDER BY time) AS cumulativeColumn` computes a cumulative sum where rows with a `NULL` value of `column` have the same value of `cumulativeColumn` as the row that preceeds them.

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, available as a CSV file (`power-plant-generation-history.csv`). To load the data, run:

```
CREATE TABLE "Generation History" AS
    FROM 'power-plant-generation-history.csv';
```

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
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 "Generation History"
ORDER BY 1, 2;
```

The result will be the following:

Plant	Date	Row
Boston	2019-01-02	1
Boston	2019-01-03	2

Plant	Date	Row
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 `INTERVAL`s). The default values for frames are from `UNBOUNDED PRECEDING` to `CURRENT ROW`. It is invalid for a frame to start after it ends. Using the EXCLUDE clause, rows around the current row can be excluded from the frame.

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-day Moving Average
Boston	2019-01-02	517450.75
Boston	2019-01-03	508793.20
Boston	2019-01-04	508529.83
...	...	...
Boston	2019-01-13	499793.00
Worcester	2019-01-02	104768.25
Worcester	2019-01-03	102713.00
Worcester	2019-01-04	102249.50
...	...	...

`EXCLUDE` Clause

The `EXCLUDE` clause allows rows around the current row to be excluded from the frame. It has the following options:

- `EXCLUDE NO OTHERS`: exclude nothing (default)
- `EXCLUDE CURRENT ROW`: exclude the current row from the window frame
- `EXCLUDE GROUP`: exclude the current row and all its peers (according to the columns specified by `ORDER BY`) from the window frame
- `EXCLUDE TIES`: exclude only the current row's peers from the window frame

`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 the `SELECT statement`.

Filtering the Results of Window Functions Using `QUALIFY`

Window functions are executed after the `WHERE` and `HAVING` clauses have been already evaluated, so it's not possible to use these clauses to filter the results of window functions The `QUALIFY` clause avoids the need for a subquery or `WITH` clause to perform this filtering.

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;
```

PostgreSQL Compatibility

DuckDB's SQL dialect closely follows the conventions of the PostgreSQL dialect. The few exceptions to this are listed on this page.

`UNION` of Boolean and Integer Values

The following query fails in PostgreSQL but successfully completes in DuckDB:

```
SELECT true AS x
UNION
SELECT 2;
```

PostgreSQL returns an error:

```
ERROR:  UNION types boolean and integer cannot be matched
```

DuckDB performs an enforced cast, therefore, it completes the query and returns the following:

x
1
2

Extensions

Extensions

Overview

DuckDB has a flexible extension mechanism that allows for dynamically loading extensions. These may extend DuckDB's functionality by providing support for additional file formats, introducing new types, and domain-specific functionality.

> 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).

Listing Extensions

To get a list of extensions, use `duckdb_extensions`:

```
SELECT extension_name, installed, description
FROM duckdb_extensions();
```

extension_name	installed	description
arrow	false	A zero-copy data integration between Apache Arrow and DuckDB
autocomplete	false	Adds support for autocomplete in the shell
...	...	...

This list will show which extensions are available, which extensions are installed, at which version, where it is installed, and more. The list includes most, but not all, available core extensions. For the full list, we maintain a list of official extensions.

Built-In Extensions

DuckDB's binary distribution comes standard with a few built-in extensions. They are statically linked into the binary and can be used as is. For example, to use the built-in `json` extension to read a JSON file:

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

To make the DuckDB distribution lightweight, only a few essential extensions are built-in, varying slightly per distribution. Which extension is built-in on which platform is documented in the list of official extensions.

Installing More Extensions

To make an extension that is not built-in available in DuckDB, two steps need to happen:

1. **Extension installation** is the process of downloading the extension binary and verifying its metadata. During installation, DuckDB stores the downloaded extension and some metadata in a local directory. From this directory DuckDB can then load the Extension whenever it needs to. This means that installation needs to happen only once.

2. **Extension loading** is the process of dynamically loading the binary into a DuckDB instance. DuckDB will search the local extension directory for the installed extension, then load it to make its features available. This means that every time DuckDB is restarted, all extensions that are used need to be (re)loaded

There are 2 main methods of making DuckDB perform the **installation** and **loading** steps for an installable extension: **explicitly** and through **autoloading**.

Autoloading Extensions

For many of DuckDB's core extensions, explicitly loading and installing extensions is not necessary. DuckDB contains an autoloading mechanism which can install and load the core extensions as soon as they are used in a query. For example, when running:

```
SELECT *
FROM 'https://raw.githubusercontent.com/duckdb/duckdb-web/main/data/weather.csv';
```

DuckDB will automatically install and load the `httpfs` extension. No explicit `INSTALL` or `LOAD` statements are required.

Not all extensions can be autoloaded. This can have various reasons: some extensions make several changes to the running DuckDB instance, making autoloading technically not (yet) possible. For others, it is prefered to have users opt-in to the extension explicitly before use due to the way they modify behaviour in DuckDB.

To see which extensions can be autoloaded, check the official extensions list.

Explicit `INSTALL` and `LOAD`

In DuckDB extensions can also explicitly installed and loaded. Both non-autoloadable and autoloadable extensions can be installed this way. To explicitly install and load an extension, DuckDB has the dedicated SQL statements `LOAD` and `INSTALL`. For example, to install and load the `spatial` extension, run:

```
INSTALL spatial;
LOAD spatial;
```

With these statements, DuckDB will ensure the spatial extension is installed (ignoring the `INSTALL` statement if it is already), then proceed to `LOAD` the spatial extension (again ignoring the statement if it is already loaded).

After installing/loading an extension, the `duckdb_extensions` function can be used to get more information.

Installing Extensions through Client APIs

For many clients, using SQL to load and install extensions is the preferred method. However, some clients have a dedicated API to install and load extensions. For example the Python API client, which has dedicated `install_extension(name: str)` and `load_extension(name: str)` methods. For more details on a specific Client API, refer to the Client API docs

Updating Extensions

> This feature was introduced in DuckDB 0.10.3.

While built-in extensions are tied to a DuckDB release due to their nature of being built into the DuckDB binary, installable extensions can and do receive updates. To ensure all currently installed extensions are on the most recent version, call:

```
UPDATE EXTENSIONS;
```

For more details on extension version refer to Extension Versioning.

Installation Location

By default, extensions are installed under the user's home directory:

`~/.duckdb/extensions/<duckdb_version>/<platform_name>/`

For stable DuckDB releases, the `<duckdb_version>` will be equal to the version tag of that release. For nightly DuckDB builds, it will be equal to the short git hash of the build. So for example, the extensions for DuckDB version v0.10.3 on macOS ARM64 (Apple Silicon) are installed to `~/.duckdb/extensions/v0.10.3/osx_arm64/`. An example installation path for a nightly DuckDB build could be `~/.duckdb/extensions/fc2e4b26a6/linux_amd64_gcc4`.

To change the default location where DuckDB stores its extensions, use the `extension_directory` configuration option:

```
SET extension_directory = '/path/to/your/extension/directory';
```

Binary Compatibility

To avoid binary compatibility issues, the binary extensions distributed by DuckDB are tied both to a specific DuckDB version and a platform. This means that DuckDB can automatically detect binary compatibility between it and a loadable extension. When trying to load an extension that was compiled for a different version or platform, DuckDB will throw an error and refuse to load the extension.

See the Working with Extensions page for details on available platforms.

Developing Extensions

The same API that the official extensions use is available for developing extensions. This allows users to extend the functionality of DuckDB such that it suits their domain the best. A template for creating extensions is available in the `extension-template` repository. This template also holds some documentation on how to get started building your own extension.

Extension Signing

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.

Unsigned Extensions

> **Warning.** Only load unsigned extensions from sources you trust. Also, avoid loading them over HTTP.

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:

```
duckdb -unsigned
```

Now any extension can be loaded, signed or not:

```
LOAD './some/local/ext.duckdb_extension';
```

For Client APIs, the `allow_unsigned_extensions` database configuration options needs to be set, see the respective Client API docs. For example, for the Python client, see the Loading and Installing Extensions section in the Python API documentation.

Working with Extensions

For advanced installation instructions and more details on extensions, see the Working with Extensions page.

Official Extensions

List of Official Extensions

Name	GitHub	Description	Autoloadable	Aliases
arrow	GitHub	A zero-copy data integration between Apache Arrow and DuckDB	no	
autocomplete		Adds support for autocomplete in the shell	yes	
aws	GitHub	Provides features that depend on the AWS SDK	yes	
azure	GitHub	Adds a filesystem abstraction for Azure blob storage to DuckDB	yes	
excel	GitHub	Adds support for Excel-like format strings	yes	
fts		Adds support for Full-Text Search Indexes	yes	
httpfs		Adds support for reading and writing files over an HTTP(S) or S3 connection	yes	http, https, s3
iceberg	GitHub	Adds support for Apache Iceberg	no	
icu		Adds support for time zones and collations using the ICU library	yes	
inet		Adds support for IP-related data types and functions	yes	
jemalloc		Overwrites system allocator with jemalloc	no	
json		Adds support for JSON operations	yes	
mysql	GitHub	Adds support for reading from and writing to a MySQL database	no	
parquet		Adds support for reading and writing Parquet files	(built-in)	
postgres	GitHub	Adds support for reading from and writing to a Postgres database	yes	postgres_scanner
spatial	GitHub	Geospatial extension that adds support for working with spatial data and functions	no	
sqlite	GitHub	Adds support for reading from and writing to SQLite database files	yes	sqlite_scanner, sqlite3
substrait	GitHub	Adds support for the Substrait integration	no	
tpcds		Adds TPC-DS data generation and query support	yes	
tpch		Adds TPC-H data generation and query support	yes	

Name	GitHub	Description	Autoloadable	Aliases
vss	GitHub	Adds support for vector similarity search queries	no	

Default Extensions

Different DuckDB clients ship a different set of extensions. We summarize the main distributions in the table below.

Name	CLI (duckdb.org)	CLI (Homebrew)	Python	R	Java	Node.js
autocomplete	yes	yes				
excel	yes					
fts	yes		yes			
httpfs			yes			
icu	yes	yes	yes		yes	yes
json	yes	yes	yes		yes	yes
parquet	yes	yes	yes	yes	yes	yes
tpcds			yes			
tpch	yes		yes			

The jemalloc extension's availability is based on the operating system. Starting with version 0.10.1, `jemalloc` is a built-in extension on Linux x86_64 (AMD64) distributions, while it will be optionally available on Linux ARM64 distributions and on macOS (via compiling from source). On Windows, it is not available.

Working with Extensions

Platforms

Extension binaries must be built for each platform. Pre-built binaries are distributed for several platforms (see below). For platforms where packages for certain extensions are not available, users can build them from source and install the resulting binaries manually.

All official extensions are distributed for the following platforms.

Platform name	Description
`linux_amd64`	Linux AMD64 (Node.js packages, etc.)
`linux_amd64_gcc4`	Linux AMD64 (Python packages, CLI, etc.)
`linux_arm64`	Linux ARM64 (e.g., AWS Graviton)
`osx_amd64`	macOS (Intel CPUs)
`osx_arm64`	macOS (Apple Silicon: M1, M2, M3 CPUs)
`windows_amd64`	Windows on Intel and AMD CPUs (x86_64)

> For some Linux ARM distributions (e.g., Python), two different binaries are distributed. These target either the `linux_arm64` or `linux_arm64_gcc4` platforms. Note that extension binaries are distributed for the first, but not the second. Effectively that means that on these platforms your glibc version needs to be 2.28 or higher to use the distributed extension binaries.

Some extensions are distributed for the following platforms:

- `windows_amd64_rtools`
- `wasm_eh` and `wasm_mvp` (see DuckDB-Wasm's extensions)

For platforms outside the ones listed above, we do not officially distribute extensions (e.g., `linux_arm64_gcc4`, `windows_amd64_mingw`).

Sharing Extensions between Clients

The shared installation location allows extensions to be shared between the client APIs *of the same DuckDB version*, as long as they share the same `platfrom` or ABI. For example, if an extension is installed with version 0.10.0 of the CLI client on macOS, 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 0.10.0.

Extension Repositories

By default, DuckDB extensions are installed from a single repository containing extensions built and signed by the core DuckDB team. This ensures the stability and security of the core set of extensions. These extensions live in the default `core` repository which points to `http://extensions.duckdb.org`.

Besides the core repository, DuckDB also supports installing extensions from other repositories. For example, the `core_nightly` repository contains nightly builds for core extensions that are built for the latest stable release of DuckDB. This allows users to try out new features in extensions before they are officially published.

Installing Extensions from a Repository

To install extensions from the default repository (default repository: `core`):

```
INSTALL httpfs;
```

To explicitly install an extension from the core repository, run either of:

```
INSTALL httpfs FROM core;
```

Or:

```
INSTALL httpfs FROM 'http://extensions.duckdb.org';
```

To install an extension from the core nightly repository:

```
INSTALL spatial FROM core_nightly;
```

Or:

```
INSTALL spatial FROM 'http://nightly-extensions.duckdb.org';
```

To install an extensions from a custom repository unknown to DuckDB:

```
INSTALL custom_extension FROM 'https://my-custom-extension-repository';
```

Alternatively, the `custom_extension_repository` setting can be used to change the default repository used by DuckDB:

```
SET custom_extension_repository = 'http://nightly-extensions.duckdb.org';
```

While any url or local path can be used as a repository, currently DuckDB contains the following predefined repositories:

alias	Url	Description
core	`http://extensions.duckdb.org`	DuckDB core extensions
core_nightly	`http://nightly-extensions.duckdb.org`	Nightly builds for `core`
local_build_debug	`./build/debug/repository`	Repository created when building DuckDB from source in debug mode (for development)
local_build_release	`./build/release/repository`	Repository created when building DuckDB from source in release mode (for development)

Working with Multiple Repositories

When working with extensions from different repositories, especially mixing `core` and `core_nightly`, it is important to keep track of the origins and version of the different extensions. For this reason, DuckDB keeps track of this in the extension installation metadata. For example:

```
INSTALL httpfs FROM core;
INSTALL aws FROM core_nightly;
SELECT extensions_name, extensions_version, installed_from, install_mode FROM duckdb_extensions();
```

Would output:

extensions_name	extensions_version	installed_from	install_mode
httpfs	62d61a417f	core	REPOSITORY
aws	42c78d3	core_nightly	REPOSITORY

extensions_name	extensions_version	installed_from	install_mode
...	...	...	...

Creating a Custom Repository

A DuckDB repository is an HTTP, HTTPS, S3, or local file based directory that serves the extensions files in a specific structure. This structure is describe here, and is the same for local paths and remote servers, for example:

```
base_repository_path_or_url
└── v0.10.3
    └── osx_arm64
        ├── autocomplete.duckdb_extension
        ├── httpfs.duckdb_extension
        ├── icu.duckdb_extension
        ├── inet.duckdb_extension
        ├── json.duckdb_extension
        ├── parquet.duckdb_extension
        ├── tpcds.duckdb_extension
        ├── tpcds.duckdb_extension
        └── tpch.duckdb_extension
```

See the `extension-template` repository for all necessary code and scripts to set up a repository.

When installing an extension from a custom repository, DuckDB will search for both a gzipped and non-gzipped version. For example:

```
INSTALL icu FROM '<custom repository>';
```

The execution of this statement will first look `icu.duckdb_extension.gz`, then `icu.duckdb_extension` in the repository's directory structure.

If the custom repository is served over HTTPS or S3, the `httpfs` extension is required. DuckDB will attempt to autoload the `httpfs` extension when an installation over HTTPS or S3 is attempted.

Downloading Extensions Directly from S3

Downloading an extension directly can be helpful when building a Lambda service 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 download an extension file directly, use the following format:

```
http://extensions.duckdb.org/v<duckdb_version>/<platform_name>/<extension_name>.duckdb_extension.gz
```

For example:

```
http://extensions.duckdb.org/v{{ site.currentduckdbversion }}/windows_amd64/json.duckdb_extension.gz
```

Loading and Installing an Extension from Explicit Paths

Installing Extensions from an Explicit Path

`INSTALL` can be used with the path to either a `.duckdb_extension` file. `.duckdb_extension.gz` files need to be decompressed before issuing `INSTALL name.duckdb_extension;`.

For example, if the file was available into the same directory as where DuckDB is being executed, you can install it as follows:

```
INSTALL 'path/to/httpfs.duckdb_extension';
```

It is also possible to specify remote paths.

Force Installing Extensions

When DuckDB installs an extension, it is copied to a local directory to be cached, avoiding any network traffic. Any subsequent calls to `INSTALL <extension_name>` will use the local version instead of downloading the extension again. To force re-downloading the extension, run:

```
FORCE INSTALL extension_name;
```

Force installing can also be used to overwrite an extension with an extension with the same name from another repository,

For example, first, `spatial` is installed from the core repository:

```
INSTALL spatial;
```

Then, to overwrite this installation with the `spatial` extension from the `core_nightly` repository:

```
FORCE INSTALL spatial FROM core_nightly;
```

Loading Extension from a Path

`LOAD` can be used with the path to a `.duckdb_extension`. For example, if the file was available at the (relative) path `path/to/httpfs.duckdb_extension`, you can load it as follows:

```
LOAD 'path/to/httpfs.duckdb_extension';
```

This will skip any currently installed file in the specifed path.

Using remote paths for compressed files is currently not possible.

Building and Installing Extensions

For building and installing extensions from source, see the building guide.

Statically Linking Extensions

To statically link extensions, follow the developer documentation's "Using extension config files" section.

Versioning of Extensions

Extension Versioning

Just like DuckDB itself, DuckDB extensions have a version. This version can be used by users to determine which features are available in the extension they have installed, and by developers to understand bug reports. DuckDB extensions can be versioned in different ways:

Extensions whose source lives in DuckDB's main repository (in-tree extensions) are tagged with the short git hash of the repository. For example, the parquet extension is built into DuckDB version `v0.10.3` (which has commit `70fd6a8a24`):

```
SELECT extension_name, extension_version, install_mode FROM duckdb_extensions() WHERE extension_
name='parquet';
```

extension_name	extension_version	install_mode
parquet	70fd6a8a24	STATICALLY_LINKED

Extensions whose source lives in a separate repository (out-of-tree extensions) have their own version. This version is **either** the short git hash of the separate repository, **or** the git version tag in Semantic Versioning format. For example, in DuckDB version `v0.10.3`, the azure extension could be versioned as follows:

```
SELECT extension_name, extension_version, install_mode
FROM duckdb_extensions()
WHERE extension_name = 'azure';
```

extension_name	extension_version	install_mode
azure	49b63dc	REPOSITORY

Updating Extensions

> This feature was introduced in DuckDB 0.10.3.

DuckDB has a dedicated statement that will automatically update all extensions to their latest version. The output will give the user information on which extensions were updated to/from which version. For example:

```
UPDATE EXTENSIONS;
```

extension_name	repository	update_result	previous_version	current_version
httpfs	core	NO_UPDATE_AVAILABLE	70fd6a8a24	70fd6a8a24
delta	core	UPDATED	d9e5cc1	04c61e4
azure	core	NO_UPDATE_AVAILABLE	49b63dc	49b63dc
aws	core_nightly	NO_UPDATE_AVAILABLE	42c78d3	42c78d3

Note that DuckDB will look for updates in the source repository for each extension. So if an extension was installed from core_nightly, it will be updated with the latest nightly build.

The update statement can also be provided with a list of specific extensions to update:

```
UPDATE EXTENSIONS (httpfs, azure);
```

extension_name	repository	update_result	previous_version	current_version
httpfs	core	NO_UPDATE_AVAILABLE	70fd6a8a24	70fd6a8a24
azure	core	NO_UPDATE_AVAILABLE	49b63dc	49b63dc

Target DuckDB Version

Currently, when extensions are compiled, they are tied to a specific version of DuckDB. What this means is that, for example, an extension binary compiled for v0.9.2 does not work for v0.10.3. In most cases, this will not cause any issues and is fully transparent; DuckDB will automatically ensure it installs the correct binary for its version. For extension developers, this means that they must ensure that new binaries are created whenever a new version of DuckDB is released. However, note that DuckDB provides an extension template that makes this fairly simple.

Arrow Extension

The `arrow` extension implements 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

The `autocomplete` extension adds supports for autocomplete in the CLI client. The extension is shipped by default with the CLI client.

Behavior

For the behavior of the `autocomplete` extension, see the documentation of the CLI client.

Functions

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

suggestion	suggestion_start
CHECKPOINT	0
DEALLOCATE	0
UPDATE	0
DROP	0

AWS Extension

The aws extension adds functionality (e.g., authentication) on top of the `httpfs` extension's S3 capabilities, using the AWS SDK.

Installing and Loading

To install and load the aws extension, run:

```
INSTALL aws;
LOAD aws;
```

Features

Function	Type	Description
`load_aws_credentials`	PRAGMA function	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
AKIAIOSFODNN7EXAMPLE		NULL	us-east-2

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
minio_duckdb_user_2		NULL	NULL

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
minio_duckdb_user_2	minio_duckdb_user_password_2	NULL	NULL

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 them as follows:

```
INSTALL aws;
INSTALL httpfs;
LOAD aws;
LOAD httpfs;
```

Usage

See the httpfs extension's S3 capabilities for instructions.

Azure Extension

The `azure` extension is a loadable extension that adds a filesystem abstraction for the Azure Blob storage to DuckDB.

Installing and Loading

To install and load the `azure` extension, run:

```
INSTALL azure;
LOAD azure;
```

Usage

Once the authentication is set up, you can query Azure storage as follows:

For *Azure Blob Storage*

Allowed URI schemes: `az` or `azure`

```
SELECT count(*)
FROM 'az://<my_container>/<path>/<my_file>.<parquet_or_csv>';
```

Globs are also supported:

```
SELECT *
FROM 'az://<my_container>/<path>/*.csv';
```

```
SELECT *
FROM 'az://<my_container>/<path>/**';
```

Or with a fully qualified path syntax:

```
SELECT count(*)
FROM 'az://<my_storage_account>.blob.core.windows.net/<my_container>/<path>/<my_file>.<parquet_or_csv>';
```

```
SELECT *
FROM 'az://<my_storage_account>.blob.core.windows.net/<my_container>/<path>/*.csv';
```

For *Azure Data Lake Storage (ADLS)*

Allowed URI schemes: `abfss`

```
SELECT count(*)
FROM 'abfss://<my_filesystem/<path>/<my_file>.<parquet_or_csv>';
```

Globs are also supported:

```
SELECT *
FROM 'abfss://<my_filesystem/<path>/*.csv';
```

```
SELECT *
FROM 'abfss://<my_filesystem/<path>/**';
```

Or with a fully qualified path syntax:

```
SELECT count(*)
FROM 'abfss://<my_storage_account>.dfs.core.windows.net/<my_filesystem>/<path>/<my_file>.<parquet_or_
csv>';
```

```
SELECT *
FROM 'abfss://<my_storage_account>.dfs.core.windows.net/<my_filesystem>/<path>/*.csv';
```

Configuration

Use the following configuration options how the extension reads remote files:

Name	Description	Type	Default
`azure_http_stats`	Include http info from Azure Storage in the `EXPLAIN ANALYZE` statement.	`BOOLEAN`	`false`
`azure_read_transfer_concurrency`	Maximum number of threads the Azure client can use for a single parallel read. If `azure_read_transfer_chunk_size` is less than `azure_read_buffer_size` then setting this > 1 will allow the Azure client to do concurrent requests to fill the buffer.	`BIGINT`	`5`
`azure_read_transfer_chunk_size`	Maximum size in bytes that the Azure client will read in a single request. It is recommended that this is a factor of `azure_read_buffer_size`.	`BIGINT`	`1024*1024`
`azure_read_buffer_size`	Size of the read buffer. It is recommended that this is evenly divisible by `azure_read_transfer_chunk_size`.	`UBIGINT`	`1024*1024`
`azure_transport_option_type`	Underlying adapter to use in the Azure SDK. Valid values are: `default` or `curl`.	`VARCHAR`	`default`
`azure_context_caching`	Enable/disable the caching of the underlying Azure SDK HTTP connection in the DuckDB connection context when performing queries. If you suspect that this is causing some side effect, you can try to disable it by setting it to false (not recommended).	`BOOLEAN`	`true`

> Setting `azure_transport_option_type` explicitly to `curl` with have the following effect:
>
> - On Linux, this may solve certificates issue (`Error: Invalid Error: Fail to get a new connection for: https://<storage account name>.blob.core.windows.net/. Problem with the SSL CA cert (path? access rights?)`) because when specifying the extension will try to find the bundle certificate in various paths (that is not done by *curl* by default and might be wrong due to static linking).
> - On Windows, this replaces the default adapter (*WinHTTP*) allowing you to use all *curl* capabilities (for example using a socks proxies).
> - On all operating systems, it will honor the following environment variables:
>   - CURL_CA_INFO: Path to a PEM encoded file containing the certificate authorities sent to libcurl. Note that this option is known to only work on Linux and might throw if set on other platforms.
>   - CURL_CA_PATH: Path to a directory which holds PEM encoded file, containing the certificate authorities sent to libcurl.

Example:

```
SET azure_http_stats = false;
SET azure_read_transfer_concurrency = 5;
SET azure_read_transfer_chunk_size = 1_048_576;
SET azure_read_buffer_size = 1_048_576;
```

Authentication

The Azure extension has two ways to configure the authentication. The preferred way is to use Secrets.

Authentication with Secret

Multiple Secret Providers are available for the Azure extension:

> - If you need to define different secrets for different storage accounts you can use the SCOPE configuration.
> - If you use fully qualified path then the ACCOUNT_NAME attribute is optional.

`CONFIG` Provider

The default provider, `CONFIG` (i.e., user-configured), allows access to the storage account using a connection string or anonymously. For example:

```
CREATE SECRET secret1 (
    TYPE AZURE,
    CONNECTION_STRING '<value>'
);
```

If you do not use authentication, you still need to specify the storage account name. For example:

```
CREATE SECRET secret2 (
    TYPE AZURE,
    PROVIDER CONFIG,
    ACCOUNT_NAME '<storage account name>'
);
```

The default `PROVIDER` is `CONFIG`.

`CREDENTIAL_CHAIN` Provider

The `CREDENTIAL_CHAIN` provider allows connecting using credentials automatically fetched by the Azure SDK via the Azure credential chain. By default, the `DefaultAzureCredential` chain used, which tries credentials according to the order specified by the Azure documentation. For example:

```
CREATE SECRET secret3 (
    TYPE AZURE,
    PROVIDER CREDENTIAL_CHAIN,
    ACCOUNT_NAME '<storage account name>'
);
```

DuckDB also allows specifying a specific chain using the `CHAIN` keyword. This takes a semicolon-separated list (`a;b;c`) of providers that will be tried in order. For example:

```
CREATE SECRET secret4 (
    TYPE AZURE,
    PROVIDER CREDENTIAL_CHAIN,
    CHAIN 'cli;env',
    ACCOUNT_NAME '<storage account name>'
);
```

The possible values are the following: `cli`; `managed_identity`; `env`; `default`;

If no explicit `CHAIN` is provided, the default one will be `default`

`SERVICE_PRINCIPAL` Provider

The `SERVICE_PRINCIPAL` provider allows connecting using a Azure Service Principal (SPN).

Either with a secret:

```
CREATE SECRET azure_spn (
    TYPE AZURE,
    PROVIDER SERVICE_PRINCIPAL,
    TENANT_ID '<tenant id>',
    CLIENT_ID '<client id>',
    CLIENT_SECRET '<client secret>',
    ACCOUNT_NAME '<storage account name>'
);
```

Or with a certificate:

```
CREATE SECRET azure_spn_cert (
    TYPE AZURE,
    PROVIDER SERVICE_PRINCIPAL,
    TENANT_ID '<tenant id>',
    CLIENT_ID '<client id>',
    CLIENT_CERTIFICATE_PATH '<client cert path>',
    ACCOUNT_NAME '<storage account name>'
);
```

Configuring a Proxy

To configure proxy information when using secrets, you can add `HTTP_PROXY`, `PROXY_USER_NAME`, and `PROXY_PASSWORD` in the secret definition. For example:

```
CREATE SECRET secret5 (
    TYPE AZURE,
    CONNECTION_STRING '<value>',
    HTTP_PROXY 'http://localhost:3128',
    PROXY_USER_NAME 'john',
    PROXY_PASSWORD 'doe'
);
```

> - When using secrets, the `HTTP_PROXY` environment variable will still be honored except if you provide an explicit value for it.

> - When using secrets, the SET variable of the *Authentication with variables* session will be ignored.
> - The Azure CREDENTIAL_CHAIN provider, the actual token is fetched at query time, not at the time of creating the secret.

Authentication with Variables (Deprecated)

```
SET variable_name = variable_value;
```

Where variable_name can be one of the following:

Name	Description	Type	Default
azure_storage_connection_string	Azure connection string, used for authenticating and configuring Azure requests.	STRING	-
azure_account_name	Azure account name, when set, the extension will attempt to automatically detect credentials (not used if you pass the connection string).	STRING	-
azure_endpoint	Override the Azure endpoint for when the Azure credential providers are used.	STRING	blob.core.windows.net
azure_credential_chain	Ordered list of Azure credential providers, in string format separated by ;. For example: 'cli;managed_identity;env'. See the list of possible values in the CREDENTIAL_CHAIN provider section. Not used if you pass the connection string.	STRING	-
azure_http_proxy	Proxy to use when login & performing request to Azure.	STRING	HTTP_PROXY environment variable (if set).
azure_proxy_user_name	Http proxy username if needed.	STRING	-
azure_proxy_password	Http proxy password if needed.	STRING	-

Additional Information

Difference between ADLS and Blob Storage

Even though ADLS implements similar functionality as the Blob storage, there are some important performance benefits to using the ADLS endpoints for globbing, especially when using (complex) glob patterns.

To demonstrate, lets look at an example of how the a glob is performed internally using respectively the Glob and ADLS endpoints.

Using the following filesystem:

```
root
├── l_receipmonth=1997-10
│   ├── l_shipmode=AIR
│   │   └── data_0.csv
```

```
│   ├── l_shipmode=SHIP
│   │   └── data_0.csv
│   └── l_shipmode=TRUCK
│       └── data_0.csv
├── l_receipmonth=1997-11
│   ├── l_shipmode=AIR
│   │   └── data_0.csv
│   ├── l_shipmode=SHIP
│   │   └── data_0.csv
│   └── l_shipmode=TRUCK
│       └── data_0.csv
└── l_receipmonth=1997-12
    ├── l_shipmode=AIR
    │   └── data_0.csv
    ├── l_shipmode=SHIP
    │   └── data_0.csv
    └── l_shipmode=TRUCK
        └── data_0.csv
```

The following query performed through the blob endpoint

```
SELECT count(*)
FROM 'az://root/l_receipmonth=1997-*/l_shipmode=SHIP/*.csv';
```

will perform the following steps:

- List all the files with the prefix `root/l_receipmonth=1997-`

  ```
  - root/l_receipmonth=1997-10/l_shipmode=SHIP/data_0.csv
  - root/l_receipmonth=1997-10/l_shipmode=AIR/data_0.csv
  - root/l_receipmonth=1997-10/l_shipmode=TRUCK/data_0.csv
  - root/l_receipmonth=1997-11/l_shipmode=SHIP/data_0.csv
  - root/l_receipmonth=1997-11/l_shipmode=AIR/data_0.csv
  - root/l_receipmonth=1997-11/l_shipmode=TRUCK/data_0.csv
  - root/l_receipmonth=1997-12/l_shipmode=SHIP/data_0.csv
  - root/l_receipmonth=1997-12/l_shipmode=AIR/data_0.csv
  - root/l_receipmonth=1997-12/l_shipmode=TRUCK/data_0.csv
  ```

- Filter the result with the requested pattern `root/l_receipmonth=1997-*/l_shipmode=SHIP/*.csv`

  ```
  - root/l_receipmonth=1997-10/l_shipmode=SHIP/data_0.csv
  - root/l_receipmonth=1997-11/l_shipmode=SHIP/data_0.csv
  - root/l_receipmonth=1997-12/l_shipmode=SHIP/data_0.csv
  ```

Meanwhile, the same query performed through the datalake endpoint,

```
SELECT count(*)
FROM 'abfss://root/l_receipmonth=1997-*/l_shipmode=SHIP/*.csv';
```

will perform the following steps:

- List all directories in `root/`

  ```
  - root/l_receipmonth=1997-10
  - root/l_receipmonth=1997-11
  - root/l_receipmonth=1997-12
  ```

- Filter and list subdirectories: `root/l_receipmonth=1997-10`, `root/l_receipmonth=1997-11`, `root/l_receipmonth=1997-12`

  ```
  - root/l_receipmonth=1997-10/l_shipmode=SHIP
  - root/l_receipmonth=1997-10/l_shipmode=AIR
  ```

```
- root/l_receipmonth=1997-10/l_shipmode=TRUCK
- root/l_receipmonth=1997-11/l_shipmode=SHIP
- root/l_receipmonth=1997-11/l_shipmode=AIR
- root/l_receipmonth=1997-11/l_shipmode=TRUCK
- root/l_receipmonth=1997-12/l_shipmode=SHIP
- root/l_receipmonth=1997-12/l_shipmode=AIR
- root/l_receipmonth=1997-12/l_shipmode=TRUCK
```

- Filter and list subdirectories: `root/l_receipmonth=1997-10/l_shipmode=SHIP`, `root/l_receipmonth=1997-11/l_shipmode=SHIP`, `root/l_receipmonth=1997-12/l_shipmode=SHIP`

```
- root/l_receipmonth=1997-10/l_shipmode=SHIP/data_0.csv
- root/l_receipmonth=1997-11/l_shipmode=SHIP/data_0.csv
- root/l_receipmonth=1997-12/l_shipmode=SHIP/data_0.csv
```

As you can see because the Blob endpoint does not support the notion of directories, the filter can only be performed after the listing, whereas the ADLS endpoint will list files recursively. Especially with higher partition/directory counts, the performance difference can be very significant.

Excel Extension

The `excel` extension, unlike what its name may suggest, does not provide support for reading Excel files. 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 currently handled through the `spatial` extension: see the Excel Import and Excel Export pages for instructions.

Installing and Loading

The `excel` 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 excel;
LOAD excel;
```

Functions

Function	Description
`excel_text(number, format_string)`	Format the given `number` per the rules given in the `format_string`.
`text(number, format_string)`	Alias for `excel_text`.

Examples

```
SELECT excel_text(1_234_567.897, 'h:mm AM/PM') AS timestamp;
```

timestamp
9:31 PM

```
SELECT excel_text(1_234_567.897, 'h AM/PM') AS timestamp;
```

timestamp
9 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.

Installing and Loading

The `fts` 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 fts;
LOAD fts;
```

Usage

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'`
`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

Name	Type	Description
`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` Function

```
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` Function

```
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.

Name	Type	Description
`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, and 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 authored by ”Muhleisen”. This retrieves ”doc1”:

```
SELECT document_identifier, 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;
```

document_identifier	text_content	score
doc1	The mallard is a dabbling duck that breeds throughout the temperate.	0.0

Search for documents about ”small cats”. This retrieves ”doc2”:

```sql
SELECT document_identifier, 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;
```

document_identifier	text_content	score
doc2	The cat is a domestic species of small carnivorous mammal.	0.0

> **Warning.** The FTS index will not update automatically when input table changes. A workaround of this limitation can be recreating the index to refresh.

httpfs (HTTP and S3)

httpfs Extension for HTTP and S3 Support

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.

Installation and Loading

The `httpfs` extension will be, by default, autoloaded on first use of any functionality exposed by this extension.

To manually install and load the `httpfs` extension, run:

```
INSTALL httpfs;
LOAD httpfs;
```

HTTP(S)

The `httpfs` extension supports connecting to HTTP(S) endpoints.

S3 API

The `httpfs` extension supports connecting to S3 API endpoints.

HTTP(S) Support

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'
]);
```

Using a Custom Certificate File

> This feature is currently only available in the nightly build. It will be released in version 0.10.1.

To use the `httpfs` extension with a custom certificate file, set the following configuration options prior to loading the extension:

```
LOAD httpfs;
SET ca_cert_file = '<certificate_file>';
SET enable_server_cert_verification = true;
```

Hugging Face Support

The `httpfs` extension introduces support for the `hf://` protocol to access data sets hosted in Hugging Face repositories. See the announcement blog post for details.

Usage

Hugging Face repositories can be queried using the following URL pattern:

`hf://datasets/<my_username>/<my_dataset>/<path_to_file>`

For example, to read a CSV file, you can use the following query:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

Where:

- `datasets-examples` is the name of the user/organization
- `doc-formats-csv-1` is the name of the dataset repository
- `data.csv` is the file path in the repository

The result of the query is:

kind	sound
dog	woof
cat	meow
pokemon	pika
human	hello

To read a JSONL file, you can run:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-jsonl-1/data.jsonl';
```

Finally, for reading a Parquet file, use the following query:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-parquet-1/data/train-00000-of-00001.parquet';
```

Each of these commands reads the data from the specified file format and displays it in a structured tabular format. Choose the appropriate command based on the file format you are working with.

Creating a local table

To avoid accessing the remote endpoint for every query, you can save the data in a DuckDB table by running a `CREATE TABLE ... AS` command. For example:

```
CREATE TABLE data AS
    SELECT *
    FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

Then, simply query the `data` table as follows:

```
SELECT *
FROM data;
```

Multiple files

To query all files under a specific directory, you can use a glob pattern. For example:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet';
```

count
173

By using glob patterns, you can efficiently handle large datasets and perform comprehensive queries across multiple files, simplifying your data inspections and processing tasks. Here, you can see how you can look for questions that contain the word “planet” in astronomy:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet'
WHERE question LIKE '%planet%';
```

count
21

Versioning and revisions

In Hugging Face repositories, dataset versions or revisions are different dataset updates. Each version is a snapshot at a specific time, allowing you to track changes and improvements. In git terms, it can be understood as a branch or specific commit.

You can query different dataset versions/revisions by using the following URL:

```
hf://datasets/<my-username>/<my-dataset>@<my_branch>/<path_to_file>
```

For example:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1@~parquet/**/*.parquet';
```

kind	sound
dog	woof
cat	meow
pokemon	pika
human	hello

The previous query will read all parquet files under the `~parquet` revision. This is a special branch where Hugging Face automatically generates the Parquet files of every dataset to enable efficient scanning.

Authentication

Configure your Hugging Face Token in the DuckDB Secrets Manager to access private or gated datasets. First, visit Hugging Face Settings – Tokens to obtain your access token. Second, set it in your DuckDB session using DuckDB's Secrets Manager. DuckDB supports two providers for managing secrets:

`CONFIG`

The user must pass all configuration information into the `CREATE SECRET` statement. To create a secret using the `CONFIG` provider, use the following command:

```
CREATE SECRET hf_token (
    TYPE HUGGINGFACE,
    TOKEN 'your_hf_token'
);
```

`CREDENTIAL_CHAIN`

Automatically tries to fetch credentials. For the Hugging Face token, it will try to get it from `~/.cache/huggingface/token`. To create a secret using the `CREDENTIAL_CHAIN` provider, use the following command:

```
CREATE SECRET hf_token (
    TYPE HUGGINGFACE,
    PROVIDER CREDENTIAL_CHAIN
);
```

S3 API Support

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.

Platforms

The `httpfs` filesystem is tested with AWS S3, Minio, Google Cloud, and lakeFS. Other services that implement the S3 API (such as Cloudflare R2) should also work, but not all features may be supported.

The following table shows 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 and Authentication

The preferred way to configure and authenticate to S3 endpoints is to use secrets. Multiple secret providers are available.

> **Deprecated.** Prior to version 0.10.0, DuckDB did not have a Secrets manager. Hence, the configuration of and authentication to S3 endpoints was handled via variables. See the legacy authentication scheme for the S3 API.

`CONFIG` Provider

The default provider, `CONFIG` (i.e., user-configured), allows access to the S3 bucket by manually providing a key. For example:

```
CREATE SECRET secret1 (
    TYPE S3,
    KEY_ID 'AKIAIOSFODNN7EXAMPLE',
    SECRET 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
    REGION 'us-east-1'
);
```

> **Tip.** If you get an IO Error (`Connection error for HTTP HEAD`), configure the endpoint explicitly via `ENDPOINT 's3.<your-region>.amazonaws.com'`.

Now, to query using the above secret, simply query any `s3://` prefixed file:

```
SELECT *
FROM 's3://my-bucket/file.parquet';
```

`CREDENTIAL_CHAIN` Provider

The `CREDENTIAL_CHAIN` provider allows automatically fetching credentials using mechanisms provided by the AWS SDK. For example, to use the AWS SDK default provider:

```
CREATE SECRET secret2 (
    TYPE S3,
    PROVIDER CREDENTIAL_CHAIN
);
```

Again, to query a file using the above secret, simply query any `s3://` prefixed file.

DuckDB also allows specifying a specific chain using the `CHAIN` keyword. This takes a semicolon-separated list (`a;b;c`) of providers that will be tried in order. For example:

```
CREATE SECRET secret3 (
    TYPE S3,
    PROVIDER CREDENTIAL_CHAIN,
    CHAIN 'env;config'
);
```

The possible values for `CHAIN` are the following:

- `config`
- `sts`
- `sso`
- `env`
- `instance`
- `process`

The `CREDENTIAL_CHAIN` provider also allows overriding the automatically fetched config. For example, to automatically load credentials, and then override the region, run:

```
CREATE SECRET secret4 (
    TYPE S3,
    PROVIDER CREDENTIAL_CHAIN,
    CHAIN 'config',
    REGION 'eu-west-1'
);
```

Overview of S3 Secret Parameters

Below is a complete list of the supported parameters that can be used for both the `CONFIG` and `CREDENTIAL_CHAIN` providers:

Name	Description	Secret	Type	Default
`KEY_ID`	The ID of the key to use	S3, GCS, R2	`STRING`	-
`SECRET`	The secret of the key to use	S3, GCS, R2	`STRING`	-
`REGION`	The region for which to authenticate (should match the region of the bucket to query)	S3, GCS, R2	`STRING`	`us-east-1`
`SESSION_TOKEN`	Optionally, a session token can be passed to use temporary credentials	S3, GCS, R2	`STRING`	-
`ENDPOINT`	Specify a custom S3 endpoint	S3, GCS, R2	`STRING`	`s3.amazonaws.com` for S3,
`URL_STYLE`	Either `vhost` or `path`	S3, GCS, R2	`STRING`	`vhost` for S3, `path` for R2 and GCS
`USE_SSL`	Whether to use HTTPS or HTTP	S3, GCS, R2	`BOOLEAN`	`true`
`URL_COMPATIBILITY_MODE`	Can help when urls contain problematic characters.	S3, GCS, R2	`BOOLEAN`	`true`
`ACCOUNT_ID`	The R2 account ID to use for generating the endpoint url	R2	`STRING`	-

Platform-Specific Secret Types

R2 Secrets

While Cloudflare R2 uses the regular S3 API, DuckDB has a special Secret type, R2, to make configuring it a bit simpler:

```
CREATE SECRET secret5 (
    TYPE R2,
    KEY_ID 'AKIAIOSFODNN7EXAMPLE',
    SECRET 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
    ACCOUNT_ID 'my_account_id'
);
```

Note the addition of the `ACCOUNT_ID` which is used to generate to correct endpoint url for you. Also note that for R2 Secrets can also use both the `CONFIG` and `CREDENTIAL_CHAIN` providers. Finally, R2 secrets are only available when using urls starting with `r2://`, for example:

```
SELECT *
FROM read_parquet('r2://some/file/that/uses/r2/secret/file.parquet');
```

GCS Secrets

While Google Cloud Storage is accessed by DuckDB using the S3 API, DuckDB has a special Secret type, GCS, to make configuring it a bit simpler:

```
CREATE SECRET secret6 (
    TYPE GCS,
    KEY_ID 'my_key',
    SECRET 'my_secret'
);
```

Note that the above secret, will automatically have the correct Google Cloud Storage endpoint configured. Also note that for GCS Secrets can also use both the `CONFIG` and `CREDENTIAL_CHAIN` providers. Finally, GCS secrets are only available when using urls starting with `gcs://` or `gs://`, for example:

```
SELECT *
FROM read_parquet('gcs://some/file/that/uses/gcs/secret/file.parquet');
```

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 named `filename` that encodes the file that a particular row originated from:

```
SELECT *
FROM read_parquet('s3://bucket/*.parquet', filename = true);
```

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, which is available when using HTTP(S) and S3 endpoints.

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 `OVERWRITE_OR_IGNORE` flag is added:

```
COPY table TO 's3://my-bucket/partitioned' (
    FORMAT PARQUET,
    PARTITION_BY (part_col_a, part_col_b),
    OVERWRITE_OR_IGNORE 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.

Name	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

Legacy Authentication Scheme for S3 API

Prior to version 0.10.0, DuckDB did not have a Secrets manager. Hence, the configuration of and authentication to S3 endpoints was handled via variables. This page documents the legacy authentication scheme for the S3 API.

> The recommended way to configuration and authentication of S3 endpoints is to use secrets.

Legacy Authentication Scheme

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 URI 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;
```

Configuration

Some additional configuration options exist for the S3 upload, though the default values should suffice for most use cases.

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`	
`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);
```

count_star()
51793

> The `allow_moved_paths` option ensures that some path resolution is performed, which allows scanning Iceberg tables that are moved.

You can also address specify the current manifest directly in the query, this may be resolved from the catalog prior to the query, in this example the manifest version is a UUID.

```
SELECT count(*)
FROM iceberg_scan('data/iceberg/lineitem_
iceberg/metadata/02701-1e474dc7-4723-4f8d-a8b3-b5f0454eb7ce.metadata.json');
```

This extension can be paired with the `httpfs` extension to access Iceberg tables in object stores such as S3.

```
SELECT count(*)
FROM iceberg_scan('s3://bucketname/lineitem_
iceberg/metadata/02701-1e474dc7-4723-4f8d-a8b3-b5f0454eb7ce.metadata.json', allow_moved_paths = true);
```

Access Iceberg Metadata

```
SELECT *
FROM iceberg_metadata('data/iceberg/lineitem_iceberg', allow_moved_paths = true);
```

manifest_path	manifest_sequence_number	manifest_content	status	content	file_path	file_format	record_count
lineitem_iceberg/metadata/10eaca8a-1e1c-421e-ad6d-b232e5ee23d3-m1.avro	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-b232e5ee23d3-m0.avro	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
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

Limitations

Writing (i.e., exporting to) Iceberg files is currently not supported.

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 and IPv6 Internet addresses. It supports the CIDR notation for subnet masks (e.g., `198.51.100.0/22`, `2001:db8:3c4d::/48`).

Installing and Loading

The `inet` extension will typically automatically load on first use, but to explicitly install and load the extension, run:

```
INSTALL inet;
LOAD inet;
```

Examples

```
SELECT '127.0.0.1'::INET AS ipv4, '2001:db8:3c4d::/48'::INET AS ipv6;
```

ipv4	ipv6
127.0.0.1	2001:db8:3c4d::/48

```
CREATE TABLE tbl (id INTEGER, ip INET);
INSERT INTO tbl VALUES
    (1, '192.168.0.0/16'),
    (2, '127.0.0.1'),
    (3, '8.8.8.8'),
    (4, 'fe80::/10'),
    (5, '2001:db8:3c4d:15::1a2f:1a2b');
SELECT * FROM tbl;
```

id	ip
1	192.168.0.0/16
2	127.0.0.1
3	8.8.8.8
4	fe80::/10
5	2001:db8:3c4d:15::1a2f:1a2b

Operations on `INET` Values

`INET` values can be compared naturally, and IPv4 will sort before IPv6. Additionally, IP addresses can be modified by adding or subtracting integers.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('127.0.0.1'::INET + 10),
    ('fe80::10'::INET - 9),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b');
SELECT cidr FROM tbl ORDER BY cidr ASC;
```

cidr
127.0.0.1
127.0.0.11
2001:db8:3c4d:15::1a2f:1a2b
fe80::7

`host` Function

The host component of an `INET` value can be extracted using the `HOST()` function.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.0.0/16'),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, host(cidr) FROM tbl;
```

cidr	host(cidr)
192.168.0.0/16	192.168.0.0
127.0.0.1	127.0.0.1
2001:db8:3c4d:15::1a2f:1a2b/96	2001:db8:3c4d:15::1a2f:1a2b

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.

Operating System Support

The availability of the `jemalloc` extension depends on the operating system.

Linux

The Linux version of DuckDB ships with the `jemalloc` extension by default.

> DuckDB v0.10.1 introduced a change: on ARM64 architecture, DuckDB is shipped without `jemalloc`, while on x86_64 (AMD64) architectures, it is shipped with `jemalloc`.

To disable the `jemalloc` extension, build DuckDB from source and set the `SKIP_EXTENSIONS` flag as follows:

```
GEN=ninja SKIP_EXTENSIONS="jemalloc" make
```

macOS

The macOS version of DuckDB does not ship with the `jemalloc` extension but can be built from source to include it:

```
GEN=ninja BUILD_JEMALLOC=1 make
```

Windows

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.

Installing and Loading

The `json` extension is shipped by default in DuckDB builds, otherwise, it will be transparently autoloaded on first use. If you would like to install and load it manually, run:

```
INSTALL json;
LOAD json;
```

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 of loading JSON data on the JSON data page:

Create a table with a column for storing JSON data:

```
CREATE TABLE example (j JSON);
```

Insert JSON data into the table:

```
INSERT INTO example VALUES
    ('{ "family": "anatidae", "species": [ "duck", "goose", "swan", null ] }');
```

Retrieve the family key's value:

```
SELECT j.family FROM example;
```

```
"anatidae"
```

Extract the family key's value with a JSONPath expression:

```
SELECT j->'$.family' FROM example;
```

```
"anatidae"
```

Extract the family key's value with a JSONPath expression as a VARCHAR:

```
SELECT j->>'$.family' FROM example;
```

```
anatidae
```

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 (modulo the equality-comparison caveat at the bottom of this page). 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 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
`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`
`format`	Can be one of `['auto', 'unstructured', 'newline_delimited', 'array']`.	VARCHAR	`'array'`
`hive_partitioning`	Whether or not to interpret the path as a Hive partitioned path.	BOOL	`false`
`ignore_errors`	Whether to ignore parse errors (only possible when `format` is `'newline_delimited'`).	BOOL	`false`
`maximum_sample_files`	The maximum number of JSON files sampled for auto-detection.	BIGINT	32
`maximum_object_size`	The maximum size of a JSON object (in bytes).	UINTEGER	16777216

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_json_auto(filename)`	Alias for `read_json` with all auto-detection enabled
`read_ndjson(filename)`	Alias for `read_json` with parameter `format` set to `'newline_delimited'`
`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
`auto_detect`	Whether to auto-detect the names of the keys and data types of the values automatically	`BOOL`	`false`
`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)`
`dateformat`	Specifies the date format to use when parsing dates. See Date Format	`VARCHAR`	`'iso'`
`maximum_depth`	Maximum nesting depth to which the automatic schema detection detects types. Set to -1 to fully detect nested JSON types	`BIGINT`	`-1`
`records`	Can be one of `['auto', 'true', 'false']`	`VARCHAR`	`'records'`
`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`
`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]
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`, e.g.,

```
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_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_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
`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_type(json[, path])`	Return the type of the supplied `json`, which is one of `ARRAY`, `BIGINT`, `BOOLEAN`, `DOUBLE`, `OBJECT`, `UBIGINT`, `VARCHAR`, and `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_valid(json)`	Return whether `json` is valid JSON
`json(json)`	Parse and minify `json`

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 the following:

```
SELECT json_extract('{"duck": [1, 2, 3]}', '$.duck[0]');
```

```
1
```

Note that DuckDB's JSON data type uses 0-based indexing.

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
```

Examples using the anatidae biological family:

```
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 j.family FROM example;
```

```
"anatidae"
```

```
SELECT j.species[0] FROM example;
```

```
"duck"
```

```
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(j) 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`	`->>`

Note that the equality comparison operator (=) has a higher precedence than the -> JSON extract operator. Therefore, surround the uses of the -> operator with parentheses when making equality comparisons. For example:

```
SELECT ((JSON '{"field": 42}')->'field') = 42;
```

> **Warning.** DuckDB's JSON data type uses 0-based indexing.

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[*]' FROM example;
```

```
["duck", "goose", "swan", null]
```

```
SELECT j->>'$.species[*]' FROM example;
```

```
[duck, goose, swan, null]
```

```
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]
```

Note that DuckDB's JSON data type uses 0-based indexing.

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
`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 example1 (k VARCHAR, v INTEGER);
INSERT INTO example1 VALUES ('duck', 42), ('goose', 7);
```

```
SELECT json_group_array(v) FROM example1;
```

```
[42, 7]
```

```
SELECT json_group_object(k, v) FROM example1;
```

```
{"duck":42,"goose":7}
```

```
CREATE TABLE example2 (j JSON);
INSERT INTO example2 VALUES
    ('{"family": "anatidae", "species": ["duck", "goose"], "coolness": 42.42}'),
    ('{"family": "canidae", "species": ["labrador", "bulldog"], "hair": true}');
```

```
SELECT json_group_structure(j) FROM example2;
```

```
{"family":"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"
```

Serializing and Deserializing 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_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.
`json_serialize_sql(varchar, skip_empty := boolean, skip_null := boolean, format := boolean)`	Scalar	Serialize a set of semicolon-separated (`;`) select statements to an equivalent list of `json` serialized statements
`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"
```

Indexing

> **Warning.** Following PostgreSQL's conventions, DuckDB uses 1-based indexing for arrays and lists but 0-based indexing for the JSON data type.

Equality Comparison

> **Warning.** Currently, equality comparison of JSON files can differ based on the context. In some cases, it is based on raw text comparison, while in other cases, it uses logical content comparison.

The following query returns true for all fields:

```
SELECT
    a != b, -- Space is part of physical JSON content. Despite equal logical content, values are treated as
not equal.
    c != d, -- Same.
    c[0] = d[0], -- Equality because space was removed from physical content of fields:
```

```
    a = c[0], -- Indeed, field is equal to empty list without space...
    b != c[0], -- ... but different from empty list with space.
FROM (
    SELECT
        '[]'::JSON AS a,
        '[ ]'::JSON AS b,
        '[[]]'::JSON AS c,
        '[[ ]]'::JSON AS d
  );
```

(a != b)	(c != d)	(c[0] = d[0])	(a = c[0])	(b != c[0])
true	true	true	true	true

MySQL Extension

The `mysql` 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.

Installing and Loading

To install the `mysql` extension, run:

```
INSTALL mysql;
```

The extension is loaded automatically upon first use. If you prefer to load it manually, run:

```
LOAD mysql;
```

Reading Data from MySQL

To make a MySQL database accessible to DuckDB use the `ATTACH` command with the `MYSQL` or the `MYSQL_SCANNER` type:

```
ATTACH 'host=localhost user=root port=0 database=mysql' AS mysqldb (TYPE MYSQL);
USE mysqldb;
```

Configuration

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. Connection information can also be specified with environment variables. If no option is provided explicitly, the MySQL extension tries to read it from an environment variable.

Setting	Default	Environment variable
`database`	`NULL`	`MYSQL_DATABASE`
`host`	`localhost`	`MYSQL_HOST`
`password`		`MYSQL_PWD`
`port`	`0`	`MYSQL_TCP_PORT`
`socket`	`NULL`	`MYSQL_UNIX_PORT`
`user`	current user	`MYSQL_USER`

Reading MySQL Tables

The tables in the MySQL database 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
signed_integers

```
SELECT * FROM signed_integers;
```

t	s	m	i	b
-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);
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, READ_ONLY);
```

Supported Operations

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
42	DuckDB

COPY

```
COPY mysql_db.tbl TO 'data.parquet';
COPY mysql_db.tbl FROM 'data.parquet';
```

You may also create a full copy of the database using the `COPY FROM DATABASE` statement:

```
COPY FROM DATABASE mysql_db TO my_duckdb_db;
```

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 and DROP SCHEMA

```
CREATE SCHEMA mysql_db.s1;
CREATE TABLE mysql_db.s1.integers (i INTEGER);
INSERT INTO mysql_db.s1.integers VALUES (42);
SELECT * FROM mysql_db.s1.integers;
```

i
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;
```

This returns:

i
42

```
ROLLBACK;
SELECT * FROM mysql_db.tmp;
```

This returns an empty table.

> The DDL statements are not transactional in MySQL.

Running SQL Queries in MySQL

The `mysql_query` Table Function

The `mysql_query` table function allows you to run arbitrary read queries within an attached database. `mysql_query` takes the name of the attached MySQL database to execute the query in, as well as the SQL query to execute. The result of the query is returned. Single-quote strings are escaped by repeating the single quote twice.

```
mysql_query(attached_database::VARCHAR, query::VARCHAR)
```

For example:

```
ATTACH 'host=localhost database=mysql' AS mysqldb (TYPE MYSQL);
SELECT * FROM mysql_query('mysqldb', 'SELECT * FROM cars LIMIT 3');
```

> **Warning.** This function is only available on DuckDB v0.10.1+, using the latest MySQL extension. To upgrade your extension, run `FORCE INSTALL mysql;`.

The `mysql_execute` Function

The `mysql_execute` function allows running arbitrary queries within MySQL, including statements that update the schema and content of the database.

```
ATTACH 'host=localhost database=mysql' AS mysqldb (TYPE MYSQL);
CALL mysql_execute('mysqldb', 'CREATE TABLE my_table (i INTEGER)');
```

> **Warning.** This function is only available on DuckDB v0.10.1+, using the latest MySQL extension. To upgrade your extension, run `FORCE INSTALL mysql;`.

Settings

Name	Description	Default
`mysql_bit1_as_boolean`	Whether or not to convert `BIT(1)` columns to `BOOLEAN`	`true`
`mysql_debug_show_queries`	DEBUG SETTING: print all queries sent to MySQL to stdout	`false`
`mysql_experimental_filter_pushdown`	Whether or not to use filter pushdown (currently experimental)	`false`
`mysql_tinyint1_as_boolean`	Whether or not to convert `TINYINT(1)` columns to `BOOLEAN`	`true`

Schema Cache

To avoid having to continuously fetch schema data from MySQL, DuckDB keeps schema information – such as the names of tables, their columns, etc. – cached. If changes are made to the schema through a different connection to the MySQL instance, such as new columns being added to a table, the cached schema information might be outdated. In this case, the function `mysql_clear_cache` can be executed to clear the internal caches.

```
CALL mysql_clear_cache();
```

PostgreSQL Extension

The `postgres` extension allows DuckDB to directly read and write data from a running Postgres database instance. The data can be queried directly from the underlying Postgres database. Data can be loaded from Postgres tables into DuckDB tables, or vice versa. See the official announcement for implementation details and background.

Installing and Loading

To install the `postgres` extension, run:

```
INSTALL postgres;
```

The extension is loaded automatically upon first use. If you prefer to load it manually, run:

```
LOAD postgres;
```

Connecting

To make a PostgreSQL database accessible to DuckDB, use the ATTACH command with the POSTGRES or POSTGRES_SCANNER type.

To connect to the "public" schema of the postgres instance running on localhost in read-write mode, run:

```
ATTACH '' AS postgres_db (TYPE POSTGRES);
```

To connect to the Postgres instance with the given parameters in read-only mode, run:

```
ATTACH 'dbname=postgres user=postgres host=127.0.0.1' AS db (TYPE POSTGRES, READ_ONLY);
```

Configuration

The ATTACH command takes as input either a `libpq` connection string or a PostgreSQL URI.

Below are some example connection strings and commonly used parameters. A full list of available parameters can be found in the Postgres documentation.

```
dbname=postgresscanner
host=localhost port=5432 dbname=mydb connect_timeout=10
```

Name	Description	Default
`dbname`	Database name	[user]
`host`	Name of host to connect to	`localhost`
`hostaddr`	Host IP address	`localhost`
`passfile`	Name of file passwords are stored in	`~/.pgpass`
`password`	Postgres password	(empty)
`port`	Port number	`5432`
`user`	Postgres user name	current user

An example URI is `postgresql://username@hostname/dbname`.

Configuring via Environment Variables

Postgres connection information can also be specified with environment variables. This can be useful in a production environment where the connection information is managed externally and passed in to the environment.

```
export PGPASSWORD="secret"
export PGHOST=localhost
export PGUSER=owner
export PGDATABASE=mydatabase
```

Then, to connect, start the duckdb process and run:

```
ATTACH '' AS p (TYPE POSTGRES);
```

Usage

The tables in the PostgreSQL database can be read as if they were normal DuckDB tables, but the underlying data is read directly from Postgres at query time.

```
SHOW ALL TABLES;
```

name
uuids

```
SELECT * FROM uuids;
```

u
6d3d2541-710b-4bde-b3af-4711738636bf
NULL
00000000-0000-0000-0000-000000000001
ffffffff-ffff-ffff-ffff-ffffffffffff

It might be desirable to create a copy of the Postgres databases in DuckDB to prevent the system from re-reading the tables from Postgres continuously, particularly for large tables.

Data can be copied over from Postgres to DuckDB using standard SQL, for example:

```
CREATE TABLE duckdb_table AS FROM postgres_db.postgres_tbl;
```

Writing Data to Postgres

In addition to reading data from Postgres, the extension allows you to create tables, ingest data into Postgres and make other modifications to a Postgres database using standard SQL queries.

This allows you to use DuckDB to, for example, export data that is stored in a Postgres database to Parquet, or read data from a Parquet file into Postgres.

Below is a brief example of how to create a new table in Postgres and load data into it.

```
ATTACH 'dbname=postgresscanner' AS postgres_db (TYPE POSTGRES);
CREATE TABLE postgres_db.tbl (id INTEGER, name VARCHAR);
INSERT INTO postgres_db.tbl VALUES (42, 'DuckDB');
```

Many operations on Postgres tables are supported. All these operations directly modify the Postgres database, and the result of subsequent operations can then be read using Postgres. 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 'dbname=postgresscanner' AS postgres_db (TYPE POSTGRES, READ_ONLY);
```

Below is a list of supported operations.

CREATE TABLE

```
CREATE TABLE postgres_db.tbl (id INTEGER, name VARCHAR);
```

INSERT INTO

```
INSERT INTO postgres_db.tbl VALUES (42, 'DuckDB');
```

SELECT

```
SELECT * FROM postgres_db.tbl;
```

id	name
42	DuckDB

COPY

You can copy tables back and forth between PostgreSQL and DuckDB:

```
COPY postgres_db.tbl TO 'data.parquet';
COPY postgres_db.tbl FROM 'data.parquet';
```

These copies use Postgres binary wire encoding. DuckDB can also write data using this encoding to a file which you can then load into Postgres using a client of your choosing if you would like to do your own connection management:

```
COPY 'data.parquet' TO 'pg.bin' WITH (FORMAT POSTGRES_BINARY);
```

The file produced will be the equivalent of copying the file to Postgres using DuckDB and then dumping it from Postgres using `psql` or another client:

DuckDB:

```
COPY postgres_db.tbl FROM 'data.parquet';
```

Postgres:

```
\copy tbl TO 'data.bin' WITH (FORMAT BINARY);
```

You may also create a full copy of the database using the COPY FROM DATABASE statement:

```
COPY FROM DATABASE postgres_db TO my_duckdb_db;
```

UPDATE

```
UPDATE postgres_db.tbl
SET name = 'Woohoo'
WHERE id = 42;
```

DELETE

```
DELETE FROM postgres_db.tbl
WHERE id = 42;
```

ALTER TABLE

```
ALTER TABLE postgres_db.tbl
ADD COLUMN k INTEGER;
```

DROP TABLE

```
DROP TABLE postgres_db.tbl;
```

CREATE VIEW

```
CREATE VIEW postgres_db.v1 AS SELECT 42;
```

CREATE SCHEMA / DROP SCHEMA

```
CREATE SCHEMA postgres_db.s1;
CREATE TABLE postgres_db.s1.integers (i INTEGER);
INSERT INTO postgres_db.s1.integers VALUES (42);
SELECT * FROM postgres_db.s1.integers;
```

i
42

```
DROP SCHEMA postgres_db.s1;
```

DETACH

```
DETACH postgres_db;
```

Transactions

```
CREATE TABLE postgres_db.tmp (i INTEGER);
BEGIN;
INSERT INTO postgres_db.tmp VALUES (42);
SELECT * FROM postgres_db.tmp;
```

This returns:

i
42

```
ROLLBACK;
SELECT * FROM postgres_db.tmp;
```

This returns an empty table.

Running SQL Queries in Postgres

The `postgres_query` Table Function

The `postgres_query` table function allows you to run arbitrary read queries within an attached database. `postgres_query` takes the name of the attached Postgres database to execute the query in, as well as the SQL query to execute. The result of the query is returned. Single-quote strings are escaped by repeating the single quote twice.

```
postgres_query(attached_database::VARCHAR, query::VARCHAR)
```

For example:

```
ATTACH 'dbname=postgresscanner' AS postgres_db (TYPE POSTGRES);
SELECT * FROM postgres_query('postgres_db', 'SELECT * FROM cars LIMIT 3');
```

brand	model	color
Ferrari	Testarossa	red
Aston Martin	DB2	blue
Bentley	Mulsanne	gray

The `postgres_execute` Function

The `postgres_execute` function allows running arbitrary queries within Postgres, including statements that update the schema and content of the database.

```
ATTACH 'dbname=postgresscanner' AS postgres_db (TYPE POSTGRES);
CALL postgres_execute('postgres_db', 'CREATE TABLE my_table (i INTEGER)');
```

> **Warning.** This function is only available on DuckDB v0.10.1+, using the latest Postgres extension. To upgrade your extension, run `FORCE INSTALL postgres;`.

Settings

The extension exposes the following configuration parameters.

Name	Description	Default
`pg_debug_show_queries`	DEBUG SETTING: print all queries sent to Postgres to stdout	`false`
`pg_connection_cache`	Whether or not to use the connection cache	`true`
`pg_experimental_filter_pushdown`	Whether or not to use filter pushdown (currently experimental)	`false`
`pg_array_as_varchar`	Read Postgres arrays as varchar - enables reading mixed dimensional arrays	`false`
`pg_connection_limit`	The maximum amount of concurrent Postgres connections	`64`
`pg_pages_per_task`	The amount of pages per task	`1000`
`pg_use_binary_copy`	Whether or not to use BINARY copy to read data	`true`

Schema Cache

To avoid having to continuously fetch schema data from Postgres, DuckDB keeps schema information – such as the names of tables, their columns, etc. – cached. If changes are made to the schema through a different connection to the Postgres instance, such as new columns being added to a table, the cached schema information might be outdated. In this case, the function `pg_clear_cache` can be executed to clear the internal caches.

```
CALL pg_clear_cache();
```

> **Deprecated.** The old `postgres_attach` function is deprecated. It is recommended to switch over to the new `ATTACH` syntax.

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)
`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)

Scalar functions	GEOMETRY	POINT_2D	LINESTRING_2D	POLYGON_2D	BOX_2D
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)	🦆	🦆	🦆	🦆	🦆
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)	🦆				

Scalar functions	GEOMETRY	POINT_2D	LINESTRING_2D	POLYGON_2D	BOX_2D
`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)`	✔				
`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)
`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
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

short_name	long_name	can_create	can_copy	can_open	help_url
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
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
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

short_name	long_name	can_create	can_copy	can_open	help_url
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
node	122351	{bicycle=yes, button_operated=yes, crossing=traffic_signals, highway=crossing, tactile_paving=no, traffic_signals:sound=yes, traffic_signals:vibration=yes}	NULL	53.5492951	9.977553	NULL	NULL
node	122397	{crossing=no, highway=traffic_signals, traffic_signals=signal, traffic_signals:direction=forward}	NULL	53.52099010	10.00096924	NULL	NULL
node	122493	{TMC:cid_58:tabcd_1:Class=Point, TMC:cid_58:tabcd_1:Direction=negative, TMC:cid_58:tabcd_1:LCLversion=9.00, TMC:cid_58:tabcd_1:LocationCode=10744, ...}	NULL	53.12961460	8.09704173	NULL	NULL
node	123566	{highway=traffic_signals}	NULL	54.61726820	8.97085571	NULL	NULL

kind	id	tags	refs	lat	lon	ref_ roles	ref_ types
node	125801	{TMC:cid_58:tabcd_1:Class=Point, TMC:cid_58:tabcd_1:Direction=negative, TMC:cid_58:tabcd_1:LCLversion=10.1, TMC:cid_58:tabcd_1:LocationCode=25041, ...}	NULL	53.07068500	8.07009939	NULL	NULL

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.

Limitations

Raster types are not supported and there is currently no plan to add them to the extension.

SQLite Extension

The SQLite extension allows DuckDB to directly read and write data from a SQLite database file. The data can be queried directly from the underlying SQLite tables. Data can be loaded from SQLite tables into DuckDB tables, or vice versa.

Installing and Loading

To install the `sqlite` extension, run:

```
INSTALL sqlite;
```

The extension is loaded automatically upon first use. If you prefer to load it manually, run:

```
LOAD sqlite;
```

Usage

To make a SQLite file accessible to DuckDB, use the `ATTACH` statement with the `SQLITE` or `SQLITE_SCANNER` type. Attached SQLite databases support both read and write operations.

For example, to attach to the `sakila.db` file, run:

```
ATTACH 'sakila.db' (TYPE SQLITE);
USE sakila;
```

The tables in the file can be read as if they were normal DuckDB tables, but the underlying data is read directly from the SQLite tables in the file at query time.

```
SHOW TABLES;
```

name
actor
address
category
city
country
customer
customer_list
film
film_actor
film_category
film_list
film_text
inventory

name
language
payment
rental
sales_by_film_category
sales_by_store
staff
staff_list
store

You can query the tables using SQL, e.g., using the example queries from `sakila-examples.sql`:

```
SELECT
    cat.name AS category_name,
    sum(ifnull(pay.amount, 0)) AS 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;
```

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.

Opening SQLite Databases Directly

SQLite databases can also be opened directly and can be used transparently instead of a DuckDB database file. In any client, when connecting, a path to a SQLite database file can be provided and the SQLite database will be opened instead.

For example, with the shell, a SQLite database can be opened as follows:

```
duckdb sakila.db
```

```
SELECT first_name
FROM actor
LIMIT 3;
```

first_name
PENELOPE
NICK
ED

Writing Data to SQLite

In addition to reading data from SQLite, the extension also allows you to create new SQLite database files, create tables, ingest data into SQLite and make other modifications to SQLite database files using standard SQL queries.

This allows you to use DuckDB to, for example, export data that is stored in a SQLite database to Parquet, or read data from a Parquet file into SQLite.

Below is a brief example of how to create a new SQLite database and load data into it.

```
ATTACH 'new_sqlite_database.db' AS sqlite_db (TYPE SQLITE);
CREATE TABLE sqlite_db.tbl (id INTEGER, name VARCHAR);
INSERT INTO sqlite_db.tbl VALUES (42, 'DuckDB');
```

The resulting SQLite database can then be read into from SQLite.

```
sqlite3 new_sqlite_database.db
```

```
SQLite version 3.39.5 2022-10-14 20:58:05
sqlite> SELECT * FROM tbl;
```

```
id  name
--  ------
42  DuckDB
```

Many operations on SQLite tables are supported. All these operations directly modify the SQLite database, and the result of subsequent operations can then be read using SQLite.

Concurrency

DuckDB can read or modify a SQLite database while DuckDB or SQLite reads or modifies the same database from a different thread or a separate process. More than one thread or process can read the SQLite database at the same time, but only a single thread or process can write to the database at one time. Database locking is handled by the SQLite library, not DuckDB. Within the same process, SQLite uses mutexes. When accessed from different processes, SQLite uses file system locks. The locking mechanisms also depend on SQLite configuration, like WAL mode. Refer to the SQLite documentation on locking for more information.

> **Warning.** Linking multiple copies of the SQLite library into the same application can lead to application errors. See sqlite_scanner Issue #82 for more information.

Supported Operations

Below is a list of supported operations.

CREATE TABLE

```
CREATE TABLE sqlite_db.tbl (id INTEGER, name VARCHAR);
```

INSERT INTO

```
INSERT INTO sqlite_db.tbl VALUES (42, 'DuckDB');
```

SELECT

```
SELECT * FROM sqlite_db.tbl;
```

id	name
42	DuckDB

COPY

```
COPY sqlite_db.tbl TO 'data.parquet';
COPY sqlite_db.tbl FROM 'data.parquet';
```

UPDATE

```
UPDATE sqlite_db.tbl SET name = 'Woohoo' WHERE id = 42;
```

DELETE

```
DELETE FROM sqlite_db.tbl WHERE id = 42;
```

ALTER TABLE

```
ALTER TABLE sqlite_db.tbl ADD COLUMN k INTEGER;
```

DROP TABLE

```
DROP TABLE sqlite_db.tbl;
```

CREATE VIEW

```
CREATE VIEW sqlite_db.v1 AS SELECT 42;
```

Transactions

```
CREATE TABLE sqlite_db.tmp (i INTEGER);
```

```
BEGIN;
INSERT INTO sqlite_db.tmp VALUES (42);
SELECT * FROM sqlite_db.tmp;
```

i
42

```
ROLLBACK;
SELECT * FROM sqlite_db.tmp;
```

i

> **Deprecated.** The old `sqlite_attach` function is deprecated. It is recommended to switch over to the new ATTACH syntax.

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.

> 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 INTEGER);
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\x01\x12\xC5\x01\x0A\xB8\x01:\xB
level\x12\x11\x0A\x07\xB2\x01\x04\x08\x0D\x18\x01\x0A\x04*\x02\x10\x01\x18\x02\x1AJ\x1AH\x08\x03\x1A\x04\x0A\x02\x10
\x1A\x1E\x08\x01\x1A\x04*\x02\x10\x01\x22\x0C\x1A\x0A\x12\x08\x0A\x04\x12\x02\x08\x01\x22\x00\x22\x06\x1A\x04\x0A\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 =
{"extensions":[{"extensionFunction":{"functionAnchor":1,"name":"lte"}},{"extensionFunction":{"functionAnchor":2,"na
not_
null"}},{"extensionFunction":{"functionAnchor":3,"name":"and"}},{"extensionFunction":{"functionAnchor":4,"name":"co
level"],"struct":{"types":[{"varchar":{"length":13,"nullability":"NULLABILITY_
NULLABLE"}},{"i32":{"nullability":"NULLABILITY_NULLABLE"}}],"nullability":"NULLABILITY_
REQUIRED"}},"filter":{"scalarFunction":{"functionReference":3,"outputType":{"bool":{"nullability":"NULLABILITY_
NULLABLE"}},"arguments":[{"value":{"scalarFunction":{"functionReference":1,"outputType":{"i32":{"nullability":"NULL
NULLABLE"}},"arguments":[{"value":{"selection":{"directReference":{"structField":{"field":1}},"rootReference":{}}}}
NULLABLE"}},"arguments":[{"value":{"selection":{"directReference":{"structField":{"field":1}},"rootReference":{}}}}
NULLABLE"}}}}]}},"expressions":[{"selection":{"directReference":{"structField":{}},"rootReference":{}}}]}},"names":
```

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\x01\x12\xC5\x01\x0A\xB8\x01:\xB
level\x12\x11\x0A\x07\xB2\x01\x04\x08\x0D\x18\x01\x0A\x04*\x02\x10\x01\x18\x02\x1AJ\x1AH\x08\x03\x1A\x04\x0A\x02\x10
\x1A\x1E\x08\x01\x1A\x04*\x02\x10\x01\x22\x0C\x1A\x0A\x12\x08\x0A\x04\x12\x02\x08\x01\x22\x00\x22\x06\x1A\x04\x0A\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 INTEGER)")
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.

```
dbExecute(con, "CREATE TABLE crossfit (exercise TEXT, difficulty_level INTEGER)")
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.

```
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.

```
result <- duckdb::duckdb_prepare_substrait(con, proto_bytes)
df <- dbFetch(result)
```

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	sum(ss_net_profit)
able	-10354620.18
ation	-10576395.52
bar	-10625236.01
ese	-10076698.16
ought	-10994052.78

Limitations

The `tpchds(<query_id>)` function runs a fixed TPC-DS query with pre-defined bind parameters (a.k.a. substitution parameters). It is not possible to change the query parameters using the `tpcds` extension.

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, otherwise it will be transparently autoloaded on first use. If you would like to install and load it manually, run:

```
INSTALL tpch;
LOAD tpch;
```

Usage

Generating Data

To generate data for scale factor 1, use:

```
CALL dbgen(sf = 1);
```

Calling `dbgen` does not clean up existing TPC-H tables. To clean up existing tables, use `DROP TABLE` before running `dbgen`:

```
DROP TABLE IF EXISTS customer;
DROP TABLE IF EXISTS lineitem;
DROP TABLE IF EXISTS nation;
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS part;
DROP TABLE IF EXISTS partsupp;
DROP TABLE IF EXISTS region;
DROP TABLE IF EXISTS supplier;
```

Running a Query

To run a query, e.g., query 4, use:

```
PRAGMA tpch(4);
```

o_orderpriority	order_count
1-URGENT	10594
2-HIGH	10476
3-MEDIUM	10410
4-NOT SPECIFIED	10556
5-LOW	10487

Listing Queries

To list all 22 queries, run:

```
FROM tpch_queries();
```

This function returns a table with columns `query_nr` and `query`.

Listing Expected Answers

To produced the expected results for all queries on scale factors 0.01, 0.1, and 1, run:

```
FROM tpch_answers();
```

This function returns a table with columns `query_nr`, `scale_factor`, and `answer`.

Data Generator Parameters

The data generator function `dbgen` has the following parameters:

Name	Type	Description
`catalog`	VARCHAR	Target catalog
`children`	UINTEGER	Number of partitions
`overwrite`	BOOLEAN	(Not used)
`sf`	DOUBLE	Scale factor
`step`	UINTEGER	Defines the partition to be generated, indexed from 0 to `children` - 1. Must be defined when the `children` arguments is defined
`suffix`	VARCHAR	Append the `suffix` to table names

Generating Larger Than Memory Data Sets

To generate data sets for large scale factors, which yield larger than memory data sets, run the `dbgen` function in steps. For example, you may generate SF300 in 10 steps:

```
CALL dbgen(sf = 300, children = 10, step = 0);
CALL dbgen(sf = 300, children = 10, step = 1);
...
CALL dbgen(sf = 300, children = 10, step = 9);
```

Limitations

- The data generator function `dbgen` is single-threaded and does not support concurrency. Running multiple steps to parallelize over different partitions is also not supported at the moment.
- The `tpch(<query_id>)` function runs a fixed TPC-H query with pre-defined bind parameters (a.k.a. substitution parameters). It is not possible to change the query parameters using the `tpch` extension.

Vector Similarity Search Extension

The `vss` extension is an experimental extension for DuckDB that adds indexing support to accelerate vector similarity search queries using DuckDB's new fixed-size ARRAY type.

See the announcement blog post.

Usage

To create a new HNSW index on a table with an `ARRAY` column, use the `CREATE INDEX` statement with the `USING HNSW` clause. For example:

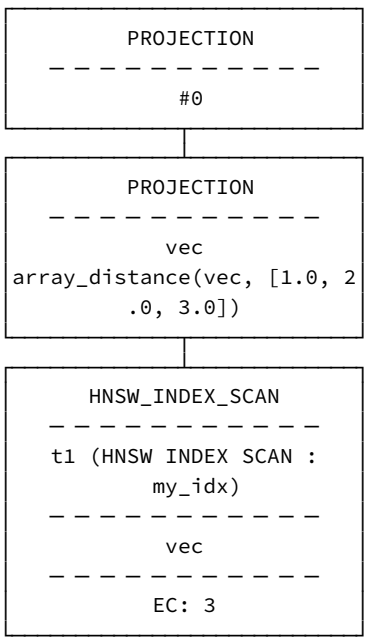
```
CREATE TABLE my_vector_table (vec FLOAT[3]);
INSERT INTO my_vector_table SELECT array_value(a, b, c) FROM range(1, 10) ra(a), range(1, 10) rb(b),
range(1, 10) rc(c);
CREATE INDEX my_hnsw_index ON my_vector_table USING HNSW (vec);
```

The index will then be used to accelerate queries that use a `ORDER BY` clause evaluating one of the supported distance metric functions against the indexed columns and a constant vector, followed by a `LIMIT` clause. For example:

```
SELECT * FROM my_vector_table ORDER BY array_distance(vec, [1, 2, 3]::FLOAT[3]) LIMIT 3;
```

We can verify that the index is being used by checking the `EXPLAIN` output and looking for the `HNSW_INDEX_SCAN` node in the plan:

```
EXPLAIN SELECT * FROM my_vector_table ORDER BY array_distance(vec, [1, 2, 3]::FLOAT[3]) LIMIT 3;
```



By default the HNSW index will be created using the euclidean distance `l2sq` (L2-norm squared) metric, matching DuckDBs `array_distance` function, but other distance metrics can be used by specifying the `metric` option during index creation. For example:

```
CREATE INDEX my_hnsw_cosine_index
ON my_vector_table
USING HNSW (vec)
WITH (metric = 'cosine');
```

The following table shows the supported distance metrics and their corresponding DuckDB functions

Metric	Function	Description
`l2sq`	`array_distance`	Euclidean distance
`cosine`	`array_cosine_similarity`	Cosine similarity
`ip`	`array_inner_product`	Inner product

Note that while each HNSW index only applies to a single column you can create multiple HNSW indexes on the same table each individually indexing a different column. Additionally, you can also create mulitple HNSW indexes to the same column, each supporting a different distance metric.

Index options

Besides the `metric` option, the HNSW index creation statement also supports the following options to control the hyperparameters of the index construction and search process:

Option	Default	Description
`ef_construction`	`128`	The number of candidate vertices to consider during the construction of the index. A higher value will result in a more accurate index, but will also increase the time it takes to build the index.
`ef_search`	`64`	The number of candidate vertices to consider during the search phase of the index. A higher value will result in a more accurate index, but will also increase the time it takes to perform a search.
`M`	`16`	The maximum number of neighbors to keep for each vertex in the graph. A higher value will result in a more accurate index, but will also increase the time it takes to build the index.
`M0`	`2 * M`	The base connectivity, or the number of neighbors to keep for each vertex in the zero-th level of the graph. A higher value will result in a more accurate index, but will also increase the time it takes to build the index.

Additionally, you can also override the `ef_search` parameter set at index construction time by setting the `SET hnsw_ef_search = <int>` configuration option at runtime. This can be useful if you want to trade search performance for accuracy or vice-versa on a per-connection basis. You can also unset the override by calling `RESET hnsw_ef_search`.

Persistence

Due to some known issues related to peristence of custom extension indexes, the HNSW index can only be created on tables in in-memory databases by default, unless the `SET hnsw_enable_experimental_persistence = <bool>` configuration option is set to `true`.

The reasoning for locking this feature behind an experimental flag is that ”WAL” recovery is not yet properly implemented for custom indexes, meaning that if a crash occurs or the database is shut down unexpectedly while there are uncommited changes to a HNSW-indexed table, you can end up with **data loss or corruption of the index**.

If you enable this option and experience an unexpected shutdown, you can try to recover the index by first starting DuckDB separately, loading the `vss` extension and then ATTACHing the database file, which ensures that the HNSW index functionality is available during WAL-playback, allowing DuckDB's recovery process to proceed without issues. But we still recommend that you do not use this feature in production environments.

With the `hnsw_enable_experimental_persistence` option enabled, the index will be persisted into the DuckDB database file (if you run DuckDB with a disk-backed database file), which means that after a database restart, the index can be loaded back into memory

from disk instead of having to be re-created. With that in mind, there are no incremental updates to persistent index storage, so every time DuckDB performs a checkpoint the entire index will be serialized to disk and overwrite itself. Similarly, after a restart of the database, the index will be deserialized back into main memory in its entirety. Although this will be deferred until you first access the table associated with the index. Depending on how large the index is, the deserialization process may take some time, but it should still be faster than simply dropping and re-creating the index.

Inserts, Updates, Deletes and Re-Compaction

The HNSW index does support inserting, updating and deleting rows from the table after index creation. However, there are two things to keep in mind:

- It's faster to create the index after the table has been populated with data as the initial bulk load can make better use of parallelism on large tables.
- Deletes are not immediately reflected in the index, but are instead ”marked” as deleted, which can cause the index to grow stale over time and negatively impact query quality and performance.

To remedy the last point, you can call the `PRAGMA hnsw_compact_index('<index name>')` pragma function to trigger a re-compaction of the index pruning deleted items, or re-create the index after a significant number of updates.

Limitations

- Only vectors consisting of `FLOAT`s (32-bit, single precision) are supported at the moment.
- The index itself is not buffer managed and must be able to fit into RAM memory.
- The size of the index in memory does not count towards DuckDB's `memory_limit` configuration parameter.
- HNSW indexes can only be created on tables in in-memory databases, unless the `SET hnsw_enable_experimental_persistence = <bool>` configuration option is set to `true`, see Persistence for more information.

Guides

Guides

The guides section contains compact how-to guides that are focused on achieving a single goal. For an API references and examples, see the rest of the documentation.

Note that there are many tools using DuckDB, which are not covered in the official guides. To find a list of these tools, check out the Awesome DuckDB repository.

> **Tip.** For a short introductory tutorial, check out the Analyzing Railway Traffic in the Netherlands tutorial.

Data Import and Export

- Data import overview

CSV Files

- How to load a CSV file into a table
- How to export a table to a CSV file

Parquet Files

- How to load a Parquet file into a table
- How to export a table to a Parquet file
- How to run a query directly on a Parquet file

HTTP(S), S3 and GCP

- How to load a Parquet file directly from HTTP(S)
- How to load a Parquet file directly from S3
- How to export a Parquet file to S3
- How to load a Parquet file from S3 Express One
- How to load a Parquet file directly from GCS
- How to load a Parquet file directly from Cloudflare R2
- How to load an Iceberg table directly from S3

JSON Files

- How to load a JSON file into a table
- How to export a table to a JSON file

Excel Files with the Spatial Extension

- How to load an Excel file into a table
- How to export a table to an Excel file

Querying Other Database Systems

- How to directly query a PostgreSQL database
- How to directly query a SQLite database
- How to directly query a MySQL database

Directly Reading Files

- How to directly read a binary file
- How to directly read a text file

Performance

- My workload is slow (troubleshooting guide)
- How to design the schema for optimal performance
- What is the ideal hardware environment for DuckDB
- What performance implications do Parquet files and (compressed) CSV files have
- How to tune workloads
- Benchmarks

Meta Queries

- How to list all tables
- How to view the schema of the result of a query
- How to quickly get a feel for a dataset using summarize
- How to view the query plan of a query
- How to profile a query

ODBC

- How to set up an ODBC application (and more!)

Python Client

- How to install the Python client
- How to execute SQL queries
- How to easily query DuckDB in Jupyter Notebooks
- How to use Multiple Python Threads with DuckDB
- How to use fsspec filesystems with DuckDB

Pandas

- How to execute SQL on a Pandas DataFrame
- How to create a table from a Pandas DataFrame
- How to export data to a Pandas DataFrame

Apache Arrow

- How to execute SQL on Apache Arrow
- How to create a DuckDB table from Apache Arrow
- How to export data to Apache Arrow

Relational API

- How to query Pandas DataFrames with the Relational API

Python Library Integrations

- How to use Ibis to query DuckDB with or without SQL
- How to use DuckDB with Polars DataFrames via Apache Arrow

SQL Features

- Friendly SQL
- As-of join
- Full-text search

SQL Editors and IDEs

- How to set up the DBeaver SQL IDE

Data Viewers

- How to visualize DuckDB databases with Tableau
- How to draw command-line plots with DuckDB and YouPlot

Data Viewers

Tableau – A Data Visualization Tool

Tableau is a popular commercial data visualization 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.

> 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.

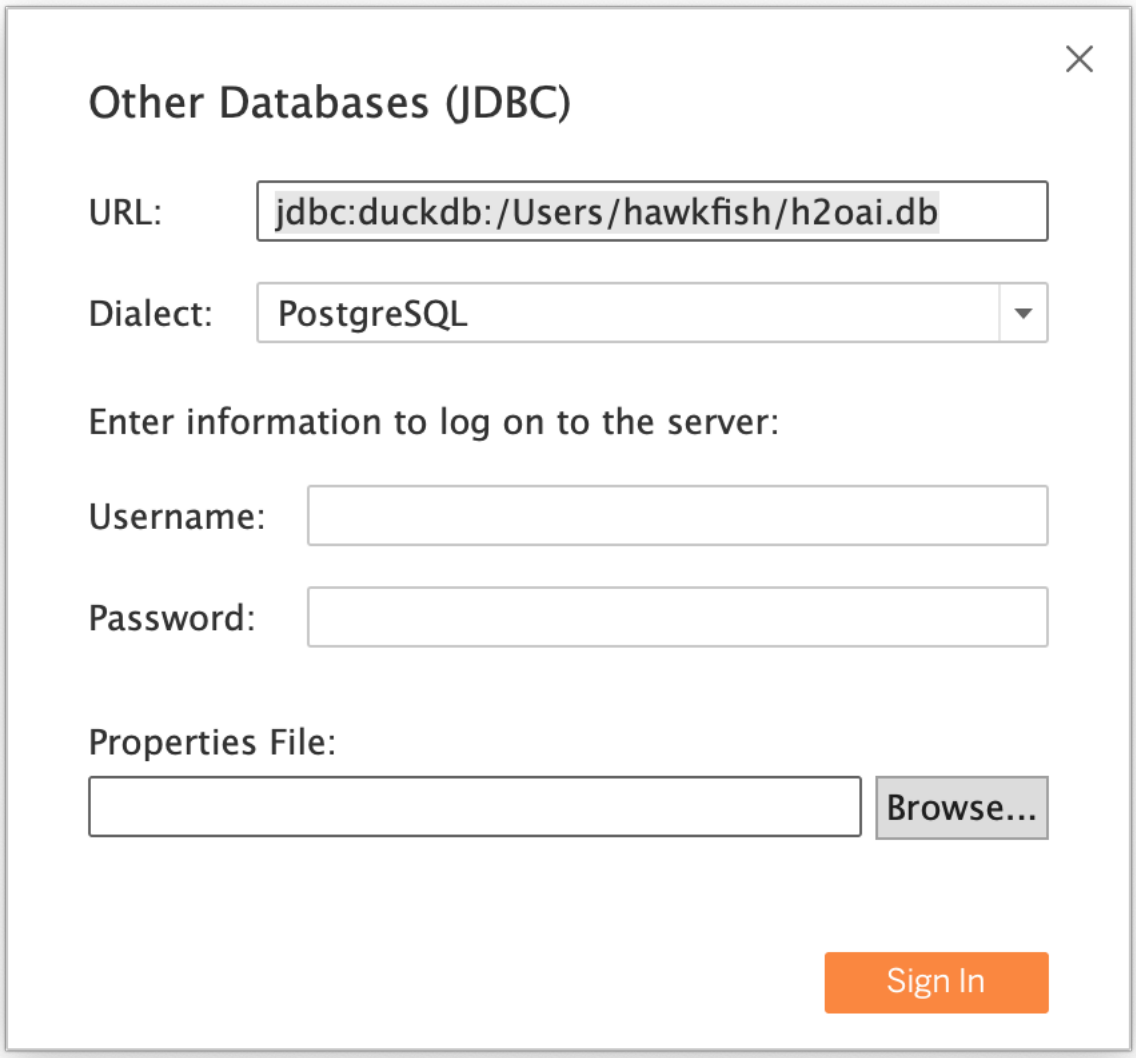
Download the JAR file.

- 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

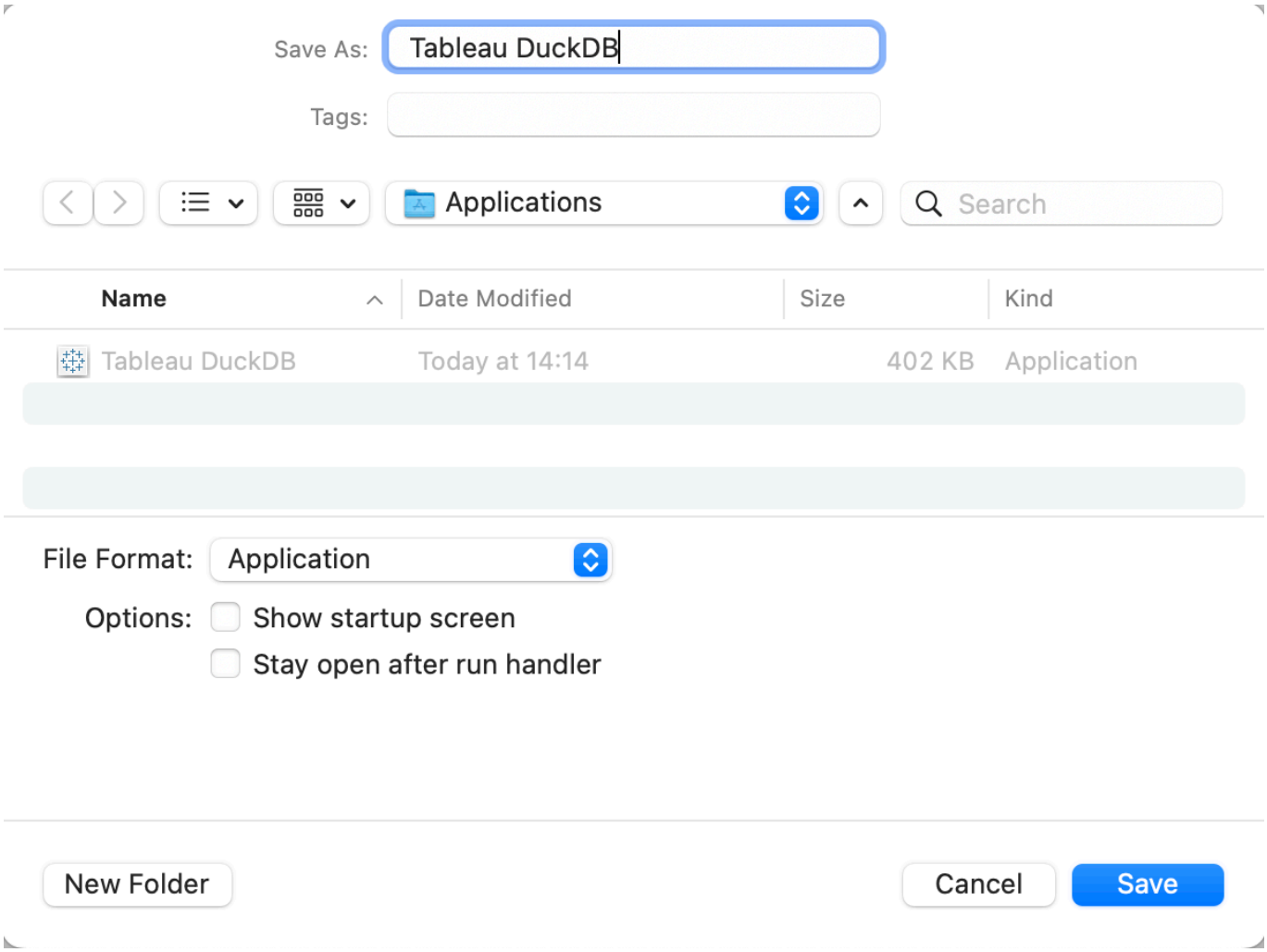
Copy the Taco file to the `/Users/[User]/Documents/My Tableau Repository/Connectors` folder. Then launch Tableau Desktop from the Terminal with the command line argument to disable signature validation:

```
/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:



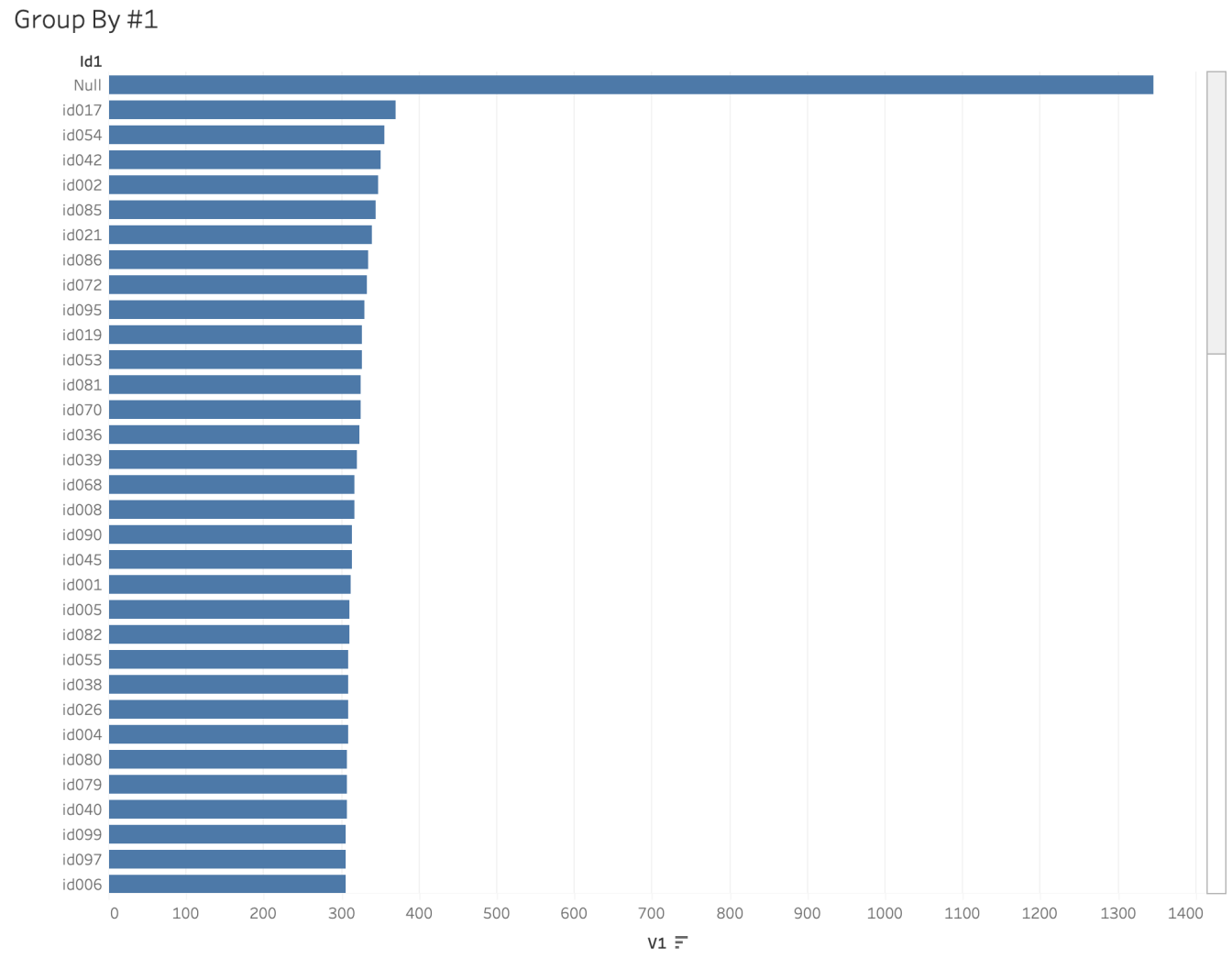
You can then double-click it to launch Tableau. You will need to change the application name in the script when you get upgrades.

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 H2O.ai benchmark query from the Parquet test file:



CLI Charting with YouPlot

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 (<query>) TO '/dev/stdout' WITH (FORMAT 'csv', HEADER);
   ```

   The full DuckDB command below outputs the query in CSV format with a header:

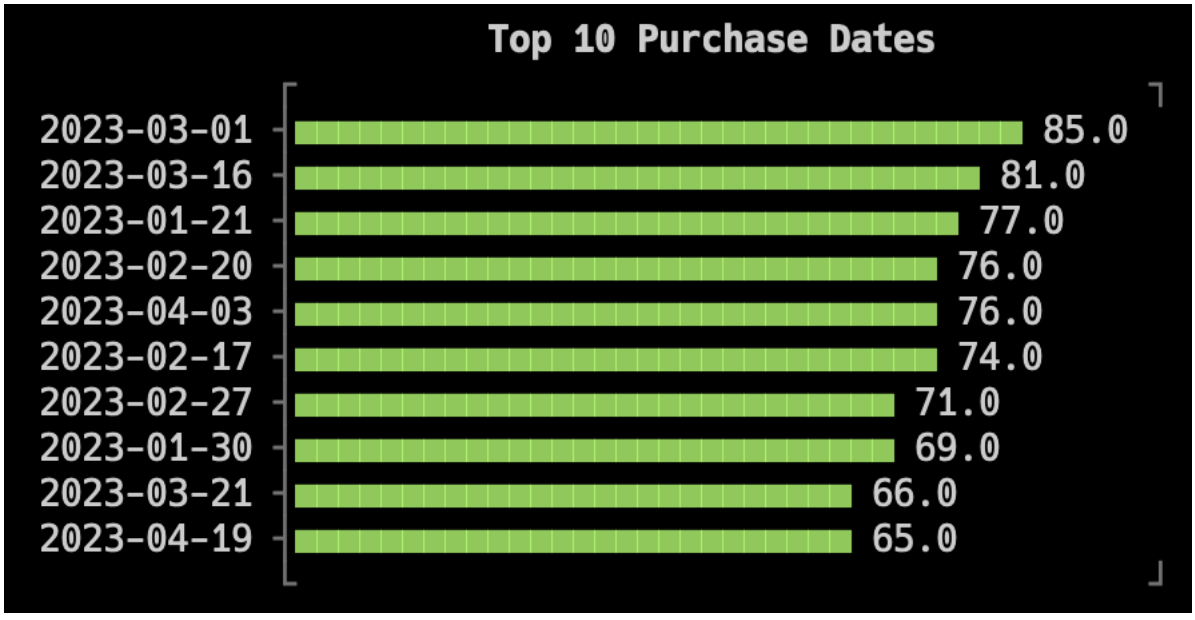
   ```
   duckdb -s "COPY (SELECT date, sum(purchases) AS total_purchases FROM read_json_auto('input.json')
   GROUP BY 1 ORDER BY 2 DESC LIMIT 10) TO '/dev/stdout' WITH (FORMAT 'csv', HEADER)"
   ```

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`).

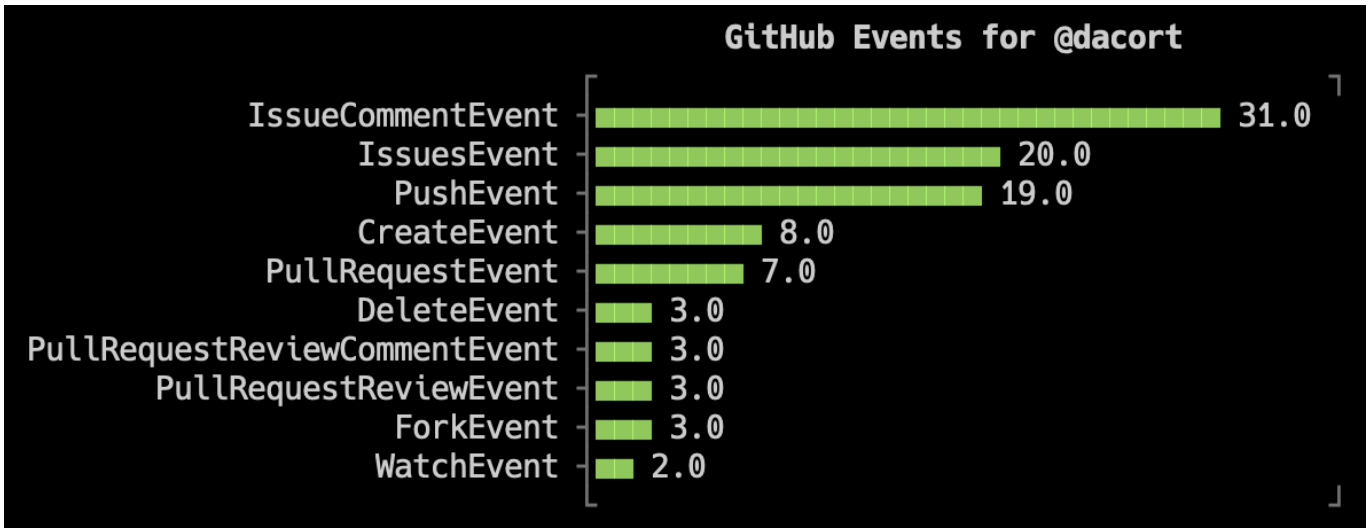


Bonus Round! stdin + stdout

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"
```



Database Integration

Database Integration

MySQL Import

To run a query directly on a running MySQL database, the `mysql` extension is required.

Installation and Loading

The extension can be installed use the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL mysql;
```

To load the `mysql` extension for usage, use the `LOAD` SQL command:

```
LOAD mysql;
```

Usage

After the `mysql` extension is installed, you can attach to a MySQL database using the following command:

```
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db (TYPE mysql_scanner, READ_ONLY);
USE mysql_db;
```

The string used by `ATTACH` is a PostgreSQL-style connection string (*not* a MySQL connection string!). It is a list of connection arguments provided in `{key}={value}` format. Below is a list of valid arguments. Any options not provided are replaced by their default values.

Setting	Default
`database`	`NULL`
`host`	`localhost`
`password`	
`port`	`0`
`socket`	`NULL`
`user`	current user

You can directly read and write the MySQL database:

```
CREATE TABLE tbl (id INTEGER, name VARCHAR);
INSERT INTO tbl VALUES (42, 'DuckDB');
```

For a list of supported operations, see the MySQL extension documentation.

PostgreSQL Import

To run a query directly on a running PostgreSQL database, the `postgres` extension is required.

Installation and Loading

The extension 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;
```

Usage

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" in the database "mydb"
SELECT * FROM postgres_scan('host=localhost port=5432 dbname=mydb', 'public', 'mytable');
```

The first parameter to the `postgres_scan` function is the PostgreSQL connection string, a list of connection arguments provided in `{key}={value}` format. Below is a list of valid arguments.

Name	Description	Default
`host`	Name of host to connect to	`localhost`
`hostaddr`	Host IP address	`localhost`
`port`	Port number	`5432`
`user`	Postgres user name	[OS user name]
`password`	Postgres password	
`dbname`	Database name	[user]
`passfile`	Name of file passwords are stored in	`~/.pgpass`

Alternatively, the entire database can be attached using the `ATTACH` command. This allows you to query all tables stored within the PostgreSQL database as if it was a regular database.

```
-- Attach the Postgres database using the given connection string
ATTACH 'host=localhost port=5432 dbname=mydb' AS test (TYPE postgres);
-- The table "tbl_name" can now be queried as if it is a regular table
SELECT * FROM test.tbl_name;
-- Switch the active database to "test"
USE test;
-- List all tables in the file
SHOW TABLES;
```

For more information see the PostgreSQL extension documentation.

SQLite Import

To run a query directly on a SQLite file, the `sqlite` extension is required.

Installation and Loading

The extension 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;
```

Usage

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 `ATTACH` command. This allows you to query all tables stored within a SQLite database file as if they were a regular database.

```
-- Attach the SQLite file "test.db"
ATTACH 'test.db' AS test (TYPE sqlite);
-- The table "tbl_name" can now be queried as if it is a regular table
SELECT * FROM test.tbl_name;
-- Switch the active database to "test"
USE test;
-- List all tables in the file
SHOW TABLES;
```

For more information see the SQLite extension documentation.

File Formats

File Formats

CSV Import

To read data from a CSV file, use the `read_csv` function in the `FROM` clause of a query.

```
SELECT * FROM read_csv('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('input.csv');
```

We can use DuckDB's optional FROM-first syntax to omit `SELECT *`:

```
CREATE TABLE new_tbl AS FROM read_csv('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('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.

Directly Reading Files

DuckDB allows directly reading files via the `read_text` and `read_blob` functions. These functions accept a filename, a list of filenames or a glob pattern, and output the content of each file as a VARCHAR or BLOB, respectively, as well as additional metadata such as the file size and last modified time.

`read_text`

The `read_text` table function reads from the selected source(s) to a VARCHAR.

```
SELECT size, parse_path(filename), content
FROM read_text('test/sql/table_function/files/*.txt');
```

size	parse_path(filename)	content
12	[test, sql, table_function, files, one.txt]	Hello World!
2	[test, sql, table_function, files, three.txt]	42
10	[test, sql, table_function, files, two.txt]	Föö Bär

The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead.

`read_blob`

The `read_blob` table function reads from the selected source(s) to a BLOB.

```
SELECT size, content, filename
FROM read_blob('test/sql/table_function/files/*');
```

size	content	filename
178	PK\x03\x04\x0A\x00\x00\x00\x00\x00\xACi=X\x14t\xCE\xC7\x0A…	test/sql/table_function/files/four.blob
12	Hello World!	test/sql/table_function/files/one.txt
2	42	test/sql/table_function/files/three.txt
10	F\xC3\xB6\xC3\xB6 B\xC3\xA4r	test/sql/table_function/files/two.txt

Schema

The schemas of the tables returned by `read_text` and `read_blob` are identical:

```
DESCRIBE FROM read_text('README.md');
```

column_name	column_type	null	key	default	extra
filename	VARCHAR	YES	NULL	NULL	NULL
content	VARCHAR	YES	NULL	NULL	NULL
size	BIGINT	YES	NULL	NULL	NULL
last_modified	TIMESTAMP	YES	NULL	NULL	NULL

Handling Missing Metadata

In cases where the underlying filesystem is unable to provide some of this data due (e.g., because HTTPFS can't always return a valid timestamp), the cell is set to `NULL` instead.

Support for Projection Pushdown

The table functions also utilize projection pushdown to avoid computing properties unnecessarily. So you could e.g., use this to glob a directory full of huge files to get the file size in the size column, as long as you omit the content column the data wont be read into DuckDB.

Excel Import

Installing the Extension

To read data from an Excel file, install and load the spatial extension. This is only needed once per DuckDB connection.

```
INSTALL spatial;
LOAD spatial;
```

Importing Excel Sheets

Use the `st_read` function in the FROM clause of a query:

```
SELECT * FROM st_read('test_excel.xlsx');
```

The `layer` parameter allows specifying the name of the Excel worksheet.

```
SELECT * FROM st_read('test_excel.xlsx', layer = 'Sheet1');
```

Creating a New Table

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 st_read('test_excel.xlsx', layer = 'Sheet1');
```

Loading to an Existing Table

To load data into an existing table from a query, use `INSERT INTO` from a SELECT statement.

```
INSERT INTO tbl
    SELECT * FROM st_read('test_excel.xlsx', layer = 'Sheet1');
```

Options

Several configuration options are also available for the underlying GDAL library that is doing the XLSX parsing. You can pass them via the `open_options` parameter of the `st_read` function as a list of `'KEY=VALUE'` strings.

Importing a Sheet with/without a Header

The option HEADERS has three possible values:

- FORCE: treat the first row as a header
- DISABLE treat the first row as a row of data
- AUTO attempt auto-detection (default)

For example, to treat the first row as a header, run:

```
SELECT *
FROM st_read(
    'test_excel.xlsx',
    layer = 'Sheet1',
    open_options = ['HEADERS=FORCE']
);
```

Detecting Types

The option `FIELD_TYPE` defines how field types should be treated:

- `STRING`: all fields should be loaded as strings (`VARCHAR` type)
- `AUTO`: field types should be auto-detected (default)

For example, to treat the first row as a header and use auto-detection for types, run:

```
SELECT *
FROM st_read(
    'test_excel.xlsx',
    layer = 'Sheet1',
    open_options = ['HEADERS=FORCE', 'FIELD_TYPES=AUTO']
);
```

To treat the fields as strings:

```
SELECT *
FROM st_read(
    'test_excel.xlsx',
    layer = 'Sheet1',
    open_options = ['FIELD_TYPES=STRING']
);
```

See Also

DuckDB can also export Excel files. For additional details on Excel support, see the spatial extension page, the GDAL XLSX driver page, and the GDAL configuration options page.

Excel Export

Installing the Extension

To export the data from a table to an Excel file, install and load the spatial extension. This is only needed once per DuckDB connection.

```
INSTALL spatial;
LOAD spatial;
```

Exporting Excel Sheets

Then use the `COPY` statement. The file will contain one worksheet with the same name as the file, but without the `.xlsx` extension.

```
COPY tbl TO 'output.xlsx' WITH (FORMAT GDAL, DRIVER 'xlsx');
```

The result of a query can also be directly exported to an Excel file.

```
COPY (SELECT * FROM tbl) TO 'output.xlsx' WITH (FORMAT GDAL, DRIVER 'xlsx');
```

> Dates and timestamps are currently not supported by the `xlsx` writer. Cast columns of those types to VARCHAR prior to creating the `xlsx` file.

See Also

DuckDB can also import Excel files. For additional details, see the spatial extension page and the GDAL XLSX driver page.

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.

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.

Querying Parquet Files

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".

Network & Cloud Storage

Cloud Storage

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)`:

```
SELECT * FROM read_parquet('https://<domain>/path/to/file.parquet');
```

For example:

```
SELECT * FROM read_parquet('https://duckdb.org/data/prices.parquet');
```

The function `read_parquet` can be omitted if the URL ends with `.parquet`:

```
SELECT * FROM read_parquet('https://duckdb.org/data/holdings.parquet');
```

Moreover, the `read_parquet` function itself can also be omitted thanks to DuckDB's replacement scan mechanism:

```
SELECT * FROM 'https://duckdb.org/data/holdings.parquet';
```

S3 Parquet Import

Prerequisites

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;
```

Credentials and Configuration

After loading the `httpfs` extension, set up the credentials and S3 region to read data:

```
CREATE SECRET (
    TYPE S3,
    KEY_ID 'AKIAIOSFODNN7EXAMPLE',
    SECRET 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
    REGION 'us-east-1'
);
```

> **Tip.** If you get an IO Error (`Connection error for HTTP HEAD`), configure the endpoint explicitly via `ENDPOINT 's3.<your-region>.amazonaws.com'`.

Alternatively, use the aws extension to retrieve the credentials automatically:

```
CREATE SECRET (
    TYPE S3,
    PROVIDER CREDENTIAL_CHAIN
);
```

Querying

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>');
```

Google Cloud Storage (GCS) and Cloudflare R2

DuckDB can also handle Google Cloud Storage (GCS) and Cloudflare R2 via the S3 API. See the relevant guides for details.

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 to write data. Note that the `region` parameter should match the region of the bucket you want to access.

```
CREATE SECRET (
    TYPE S3,
    KEY_ID 'AKIAIOSFODNN7EXAMPLE',
    SECRET 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
    REGION 'us-east-1'
);
```

> **Tip.** If you get an IO Error (`Connection error for HTTP HEAD`), configure the endpoint explicitly via `ENDPOINT 's3.<your-region>.amazonaws.com'`.

Alternatively, use the aws extension to retrieve the credentials automatically:

```
CREATE SECRET (
    TYPE S3,
    PROVIDER CREDENTIAL_CHAIN
);
```

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 provide the credentials as follows:

```
CREATE SECRET (
    TYPE GCS,
    KEY_ID 'AKIAIOSFODNN7EXAMPLE',
    SECRET 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY'
);
```

After setting up the GCS credentials, you can export using:

```
COPY <table_name> TO 'gs://gcs_bucket/file.parquet';
```

S3 Iceberg Import

Prerequisites

To load an Iceberg file from S3, both the `httpfs` and `iceberg` extensions are required. They can be installed use the `INSTALL` SQL command. The extensions only need to be installed once.

```
INSTALL httpfs;
INSTALL iceberg;
```

To load the extensions for usage, use the LOAD command:

```
LOAD httpfs;
LOAD iceberg;
```

Credentials

After loading the extensions, set up the credentials and S3 region to read data. You may either use an access key and secret, or a token.

```
CREATE SECRET (
    TYPE S3,
    KEY_ID 'AKIAIOSFODNN7EXAMPLE',
    SECRET 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
    REGION 'us-east-1'
);
```

Alternatively, use the aws extension to retrieve the credentials automatically:

```
CREATE SECRET (
    TYPE S3,
    PROVIDER CREDENTIAL_CHAIN
);
```

Loading Iceberg Tables from S3

After the extensions are set up and the S3 credentials are correctly configured, Iceberg table can be read from S3 using the following command:

```
SELECT *
FROM iceberg_scan('s3://<bucket>/<iceberg-table-folder>/metadata/<id>.metadata.json');
```

Note that you need to link directly to the manifest file. Otherwise you'll get an error like this:

```
Error: IO Error: Cannot open file "s3://<bucket>/<iceberg-table-folder>/metadata/version-hint.text": No
such file or directory
```

S3 Express One

In late 2023, AWS announced the S3 Express One Zone, a high-speed variant of traditional S3 buckets. DuckDB can read S3 Express One buckets using the `httpfs` extension.

Credentials and Configuration

The configuration of S3 Express One buckets is similar to regular S3 buckets with one exception: we have to specify the endpoint according to the following pattern:

```
s3express-<availability zone>.<region>.amazonaws.com
```

where the `<availability zone>` (e.g., `use-az5`) can be obtained from the S3 Express One bucket's configuration page and the `<region>` is the AWS region (e.g., `us-east-1`).

For example, to allow DuckDB to use an S3 Express One bucket, configure the Secrets manager as follows:

```
CREATE SECRET (
    TYPE S3,
    REGION 'us-east-1',
    KEY_ID 'AKIAIOSFODNN7EXAMPLE',
    SECRET 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
    ENDPOINT 's3express-use1-az5.us-east-1.amazonaws.com'
);
```

Instance Location

For best performance, make sure that the EC2 instance is in the same availability zone as the S3 Express One bucket you are querying. To determine the mapping between zone names and zone IDs, use the `aws ec2 describe-availability-zones` command:

```
aws ec2 describe-availability-zones --output json | \
    jq -r  '.AvailabilityZones[] | select(.ZoneName == "us-east-1f") | .ZoneId'
# use1-az5
aws ec2 describe-availability-zones --output json | \
    jq -r  '.AvailabilityZones[] | select(.ZoneId == "use1-az5") | .ZoneName'
# us-east-1f
```

Querying

You can query the S3 Express One bucket as any other S3 bucket:

```
SELECT *
FROM 's3://express-bucket-name--use1-az5--x-s3/my-file.parquet';
```

Performance

We ran two experiments on a `c7gd.12xlarge` instance using the LDBC SF300 Comments `creationDate` Parquet file file (also used in the microbenchmarks of the performance guide).

Experiment	File size	Runtime
Loading only from Parquet	4.1 GB	3.5s
Creating local table from Parquet	4.1 GB	5.1s

The "loading only" variant is running the load as part of an `EXPLAIN ANALYZE` statement to measure the runtime without account creating a local table, while the "creating local table" variant uses `CREATE TABLE ... AS` to create a persistent table on the local disk.

Google Cloud Storage Import

Prerequisites

The Google Cloud Storage (GCS) can be used via the `httpfs` extension. This can be installed with the `INSTALL httpfs` SQL command. This only needs to be run once.

Credentials and Configuration

You need to create HMAC keys and declare them:

```
CREATE SECRET (
    TYPE GCS,
    KEY_ID 'AKIAIOSFODNN7EXAMPLE',
    SECRET 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY'
);
```

Querying

After setting up the GCS credentials, you can query the GCS data using:

```
SELECT *
FROM read_parquet('gs://<gcs_bucket>/<file.parquet>');
```

Attaching to a Database

You can attach to a database file in read-only mode:

```
LOAD httpfs;
ATTACH 'gs://<gcs_bucket>/<file.duckdb>' AS <duckdb_database> (READ_ONLY);
```

> Databases in Google Cloud Storage can only be attached in read-only mode.

Cloudflare R2 Import

Prerequisites

For Cloudflare R2, the S3 Compatibility API allows you to use DuckDB's S3 support to read and write from R2 buckets. This requires the `httpfs` extension, which can be installed use the `INSTALL` SQL command. This only needs to be run once.

Credentials and Configuration

You will need to generate an S3 auth token and create an R2 secret in DuckDB:

```
CREATE SECRET (
    TYPE R2,
    KEY_ID 'AKIAIOSFODNN7EXAMPLE',
    SECRET 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
    ACCOUNT_ID 'my_account_id'
);
```

Querying

After setting up the R2 credentials, you can query the R2 data using:

```
SELECT * FROM read_parquet('r2://<r2_bucket_name>/<file>');
```

Attach to a DuckDB Database over HTTPS or S3

You can establish a read-only connection to a DuckDB instance via HTTPS or the S3 API.

Prerequisites

This guide requires the `httpfs` extension, which can be installed using the `INSTALL httpfs` SQL command. This only needs to be run once.

Attaching to a Database over HTTPS

To connect to a DuckDB database via HTTPS, use the ATTACH statement as follows:

```
LOAD httpfs;
ATTACH 'https://blobs.duckdb.org/databases/stations.duckdb' AS stations_db (READ_ONLY);
```

Then, the database can be queried using:

```
SELECT count(*) AS num_stations
FROM stations_db.stations;
```

num_stations
578

Attaching to a Database over the S3 API

To connect to a DuckDB database via the S3 API, configure the authentication for your bucket (if required). Then, use the ATTACH statement as follows:

```
LOAD httpfs;
ATTACH 's3://duckdb-blobs/databases/stations.duckdb' AS stations_db (READ_ONLY);
```

The database can be queried using:

```
SELECT count(*) AS num_stations
FROM stations_db.stations;
```

num_stations
578

> Connecting to S3-compatible APIs such as the Google Cloud Storage (`gs://`) is also supported.

Limitations

- The `httpfs` extension has to be loaded manually, auto-loading is currently not supported.
- Only read-only connections are allowed, writing the database via the HTTPS protocol or the S3 API is not possible.

Meta Queries

Describe

Describing a Table

In order to view the schema of a table, use `DESCRIBE` or `SHOW` followed by the table name.

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j VARCHAR);
DESCRIBE tbl;
SHOW tbl; -- equivalent to DESCRIBE tbl;
```

column_name	column_type	null	key	default	extra
i	INTEGER	NO	PRI	NULL	NULL
j	VARCHAR	YES	NULL	NULL	NULL

Describing a Query

In order to view the schema of the result of a query, prepend `DESCRIBE` to a query.

```
DESCRIBE SELECT * FROM tbl;
```

column_name	column_type	null	key	default	extra
i	INTEGER	YES	NULL	NULL	NULL
j	VARCHAR	YES	NULL	NULL	NULL

Note that there are subtle differences: compared to the result when describing a table, nullability (`null`) and key information (`key`) are lost.

Using `DESCRIBE` in a Subquery

`DESCRIBE` can be used a subquery. This allows creating a table from the description, for example:

```
CREATE TABLE tbl_description AS SELECT * FROM (DESCRIBE tbl);
```

Describing Remote Tables

It is possible to describe remote tables via the `httpfs` extension using the `DESCRIBE TABLE` statement. For example:

```
DESCRIBE TABLE 'https://blobs.duckdb.org/data/Star_Trek-Season_1.csv';
```

column_name	column_type	null	key	default	extra
season_num	BIGINT	YES	NULL	NULL	NULL
episode_num	BIGINT	YES	NULL	NULL	NULL
aired_date	DATE	YES	NULL	NULL	NULL
cnt_kirk_hookups	BIGINT	YES	NULL	NULL	NULL
cnt_downed_redshirts	BIGINT	YES	NULL	NULL	NULL
bool_aliens_almost_took_over_planet	BIGINT	YES	NULL	NULL	NULL
bool_aliens_almost_took_over_enterprise	BIGINT	YES	NULL	NULL	NULL
cnt_vulcan_nerve_pinch	BIGINT	YES	NULL	NULL	NULL
cnt_warp_speed_orders	BIGINT	YES	NULL	NULL	NULL
highest_warp_speed_issued	BIGINT	YES	NULL	NULL	NULL
bool_hand_phasers_fired	BIGINT	YES	NULL	NULL	NULL
bool_ship_phasers_fired	BIGINT	YES	NULL	NULL	NULL
bool_ship_photon_torpedos_fired	BIGINT	YES	NULL	NULL	NULL
cnt_transporter_pax	BIGINT	YES	NULL	NULL	NULL
cnt_damn_it_jim_quote	BIGINT	YES	NULL	NULL	NULL
cnt_im_givin_her_all_shes_got_quote	BIGINT	YES	NULL	NULL	NULL
cnt_highly_illogical_quote	BIGINT	YES	NULL	NULL	NULL
bool_enterprise_saved_the_day	BIGINT	YES	NULL	NULL	NULL

EXPLAIN: Inspect Query Plans

In order to view the query plan of a query, prepend `EXPLAIN` to a query.

```
EXPLAIN SELECT * FROM tbl;
```

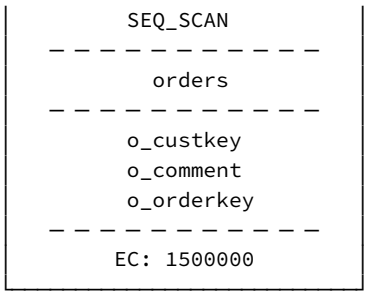
By default only the final physical plan is shown. In order to see the unoptimized and optimized logical plans, change the `explain_output` setting:

```
SET explain_output = 'all';
```

Below is an example of running `EXPLAIN` on Q13 of the TPC-H benchmark on the scale factor 1 data set.

```
EXPLAIN
    SELECT
        c_count,
        count(*) AS custdist
    FROM (
        SELECT
            c_custkey,
            count(o_orderkey)
        FROM
            customer
        LEFT OUTER JOIN orders ON c_custkey = o_custkey
        AND o_comment NOT LIKE '%special%requests%'
    GROUP BY
        c_custkey) AS c_orders (c_custkey,
            c_count)
    GROUP BY
        c_count
```

```
    ORDER BY
        custdist DESC,
        c_count DESC;
```



See Also

For more information, see the Profiling page.

EXPLAIN ANALYZE: Profile Queries

In order to profile a query, prepend `EXPLAIN ANALYZE` to a query.

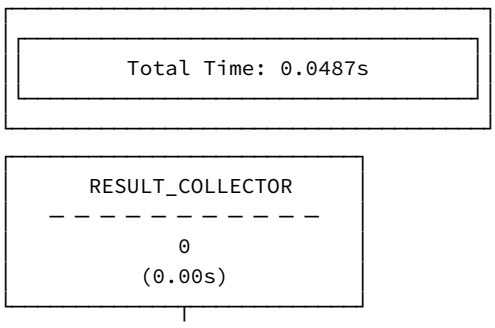
```
EXPLAIN ANALYZE SELECT * FROM tbl;
```

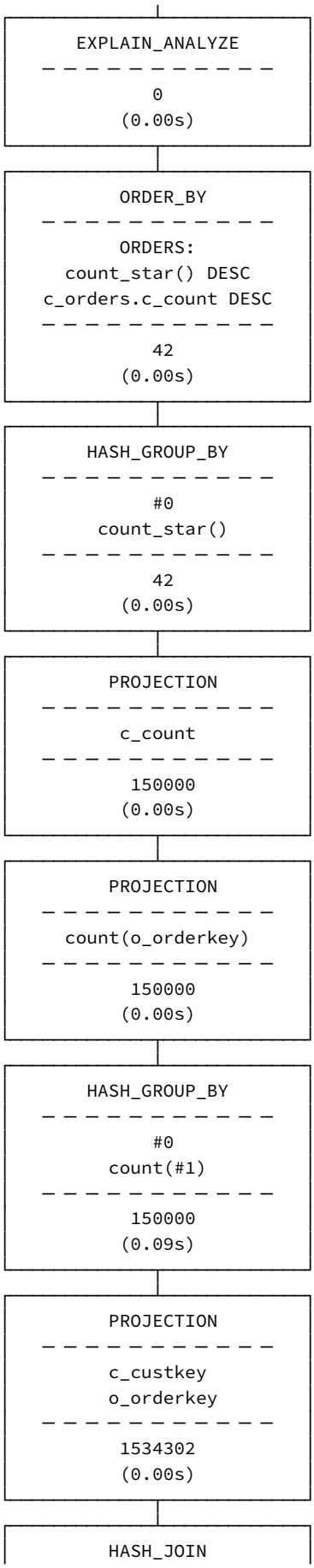
The query plan will be pretty-printed to the screen using timings for every operator.

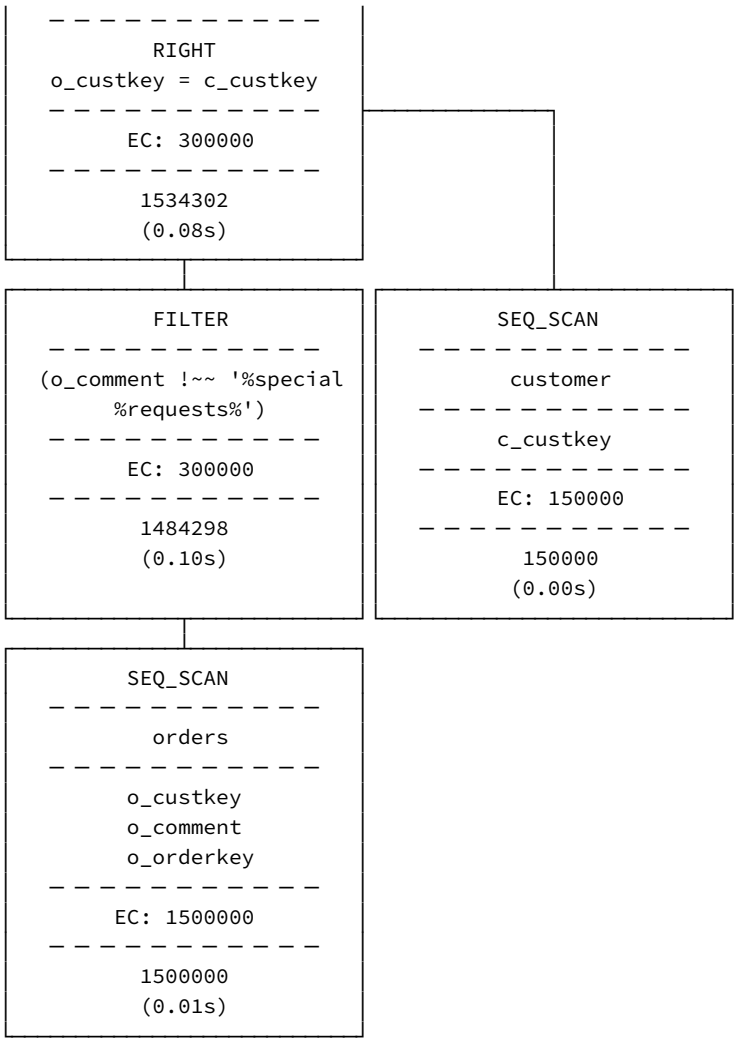
Note that the **cumulative** wall-clock time that is spent on every operator is shown. When multiple threads are processing the query in parallel, the total processing time of the query may be lower than the sum of all the times spent on the individual operators.

Below is an example of running `EXPLAIN ANALYZE` on Q13 of the TPC-H benchmark on the scale factor 1 data set.

```
EXPLAIN ANALYZE
    SELECT
        c_count,
        count(*) AS custdist
    FROM (
        SELECT
            c_custkey,
            count(o_orderkey)
        FROM
            customer
        LEFT OUTER JOIN orders ON c_custkey = o_custkey
        AND o_comment NOT LIKE '%special%requests%'
    GROUP BY
        c_custkey) AS c_orders (c_custkey,
            c_count)
    GROUP BY
        c_count
    ORDER BY
        custdist DESC,
        c_count DESC;
```




EXPLAIN_ANALYZE
0
(0.00s)
ORDER_BY
ORDERS:
count_star() DESC
c_orders.c_count DESC
42
(0.00s)
HASH_GROUP_BY
#0
count_star()
42
(0.00s)
PROJECTION
c_count
150000
(0.00s)
PROJECTION
count(o_orderkey)
150000
(0.00s)
HASH_GROUP_BY
#0
count(#1)
150000
(0.09s)
PROJECTION
c_custkey
o_orderkey
1534302
(0.00s)
HASH_JOIN



See Also

For more information, see the Profiling page.

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

`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.

See Also

The SQL-standard `information_schema` views are also defined. Moreover, DuckDB defines `sqlite_master` and many PostgreSQL system catalog tables for compatibility with SQLite and PostgreSQL respectively.

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 (`min`, `max`, `approx_unique`, `avg`, `std`, `q25`, `q50`, `q75`, `count`), and return these along the column name, column type, and the percentage of `NULL` values in the column.

Usage

In order to summarize the contents of a table, use `SUMMARIZE` followed by the table name.

```sql
SUMMARIZE tbl;
```

In order to summarize a query, prepend `SUMMARIZE` to a query.

```sql
SUMMARIZE SELECT * FROM tbl;
```

Example

Below is an example of `SUMMARIZE` on the `lineitem` table of TPC-H `SF1` table, generated using the `tpch` extension.

```sql
INSTALL tpch;
LOAD tpch;
CALL dbgen(sf = 1);
```

```sql
SUMMARIZE lineitem;
```

column_name	column_type	min	max	approx_unique	avg	std	q25	q50	q75	count	null_percentage
l_orderkey	INTEGER	1	6000000	1508227	3000279.604204982	1732187.8734803519	1509447	2989869	4485232	6001215	0.0%
l_partkey	INTEGER	1	200000	202598	100017.98932999402	57735.69082650496	49913	99992	150039	6001215	0.0%
l_suppkey	INTEGER	1	10000	10061	5000.602606138924	2886.9619987306114	2501	4999	7500	6001215	0.0%
l_linenumber	INTEGER	1	7	7	3.0005757167506912	1.7324314036519328	2	3	4	6001215	0.0%

column_ name	column_ type	min	max	approx_ unique	avg	std	q25	q50	q75	count	null_ percentage
l_ quantity	DECIMAL(15,2)	1.00	50.00	50	25.507967136654827	14.426262537016918	13	26	38	6001215	0.0%
l_ extendedprice	DECIMAL(15,2)	901.00	104949.50	923139	38255.138484656854	23300.43871096221	18756	36724	55159	6001215	0.0%
l_ discount	DECIMAL(15,2)	0.00	0.10	11	0.04999943011540163	0.03161985510812596	0	0	0	6001215	0.0%
l_tax	DECIMAL(15,2)	0.00	0.08	9	0.04001350893110812	0.025816551798842728	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	2516	NULL	NULL	NULL	NULL	NULL	6001215	0.0%
l_ commitdate	DATE	1992-01-31	1998-10-31	2460	NULL	NULL	NULL	NULL	NULL	6001215	0.0%
l_ receiptdate	DATE	1992-01-04	1998-12-31	2549	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? furiously iro	3558599	NULL	NULL	NULL	NULL	NULL	6001215	0.0%

Using SUMMARIZE in a Subquery

SUMMARIZE can be used a subquery. This allows creating a table from the summary, for example:

```sql
CREATE TABLE tbl_summary AS SELECT * FROM (SUMMARIZE tbl);
```

Summarizing Remote Tables

It is possible to summarize remote tables via the `httpfs` extension using the SUMMARIZE TABLE statement. For example:

```sql
SUMMARIZE TABLE 'https://blobs.duckdb.org/data/Star_Trek-Season_1.csv';
```

DuckDB Environment

DuckDB provides a number of functions and PRAGMA options to retrieve information on the running DuckDB instance and its environment.

Version

The `version()` function returns the version number of DuckDB.

```
SELECT version();
```

version()
v0.10.2

Using a PRAGMA:

```
PRAGMA version;
```

library_version	source_id
v0.10.2	1601d94f94

Platform

The platform information consists of the operating system, system architecture, and, optionally, the compiler. The platform is used when installing extensions. To retrieve the platform, use the following PRAGMA:

```
PRAGMA platform;
```

On macOS, running on Apple Silicon architecture, the result is:

platform
osx_arm64

On Windows, running on an AMD64 architecture, the platform is `windows_amd64`. On CentOS 7, running on the AMD64 architecture, the platform is `linux_amd64_gcc4`. On Ubuntu 22.04, running on the ARM64 architecture, the platform is `linux_arm64`.

Extensions

To get a list of DuckDB extension and their status (e.g., `loaded`, `installed`), use the `duckdb_extensions()` function:

```
SELECT *
FROM duckdb_extensions();
```

Meta Table Functions

DuckDB has the following built-in table functions to obtain metadata about available catalog objects:

- `duckdb_columns()`: columns
- `duckdb_constraints()`: constraints
- `duckdb_databases()`: lists the databases that are accessible from within the current DuckDB process
- `duckdb_dependencies()`: dependencies between objects
- `duckdb_extensions()`: extensions
- `duckdb_functions()`: functions
- `duckdb_indexes()`: secondary indexes
- `duckdb_keywords()`: DuckDB's keywords and reserved words
- `duckdb_optimizers()`: the available optimization rules in the DuckDB instance

- `duckdb_schemas()`: schemas
- `duckdb_sequences()`: sequences
- `duckdb_settings()`: settings
- `duckdb_tables()`: base tables
- `duckdb_types()`: data types
- `duckdb_views()`: views
- `duckdb_temporary_files()`: the temporary files DuckDB has written to disk, to offload data from memory

ODBC

ODBC 101: A Duck Themed Guide to ODBC

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.

> 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.

The DuckDB ODBC driver has the following handle types.

Environment

Handle name	Environment
Type name	`SQL_HANDLE_ENV`
:--	:--------
Description	Manages the environment settings for ODBC operations, and provides a global context in which to access data.
Use case	Initializing ODBC, managing driver behavior, resource allocation
Additional information	Must be allocated once per application upon starting, and freed at the end.

Connection

Handle name	Connection
Type name	SQL_HANDLE_DBC
:--	:--------
Description	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.
Use case	Establishing a connection to a database, managing the connection state
Additional information	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.

Statement

Handle name	Statement
Type name	SQL_HANDLE_STMT
:--	:--------
Description	Handles the execution of SQL statements, as well as the returned result sets.
Use case	Executing SQL queries, fetching result sets, managing statement options.
Additional information	To facilitate the execution of concurrent queries, multiple handles can be allocated per connection.

Descriptor

Handle name	Descriptor
Type name	SQL_HANDLE_DESC
:--	:--------
Description	Describes the attributes of a data structure or parameter, and allows the application to specify the structure of data to be bound/retrieved.
Use case	Describing table structures, result sets, binding columns to application buffers
Additional information	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 on the "Choosing a Data Source or Driver" page of the SQL Server documentation.

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++.

> To install the driver as well as anything else you will need follow these instructions.

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:

On macOS:

```
brew install unixodbc
```

On Ubuntu and Debian:

```
sudo apt-get install -y unixodbc-dev
```

On Fedora, CentOS, and Red Hat:

```
sudo yum install -y 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

2.a. Connecting with SQLConnect

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;
```

2.b. Connecting with SQLDriverConnect

Alternatively, you can connect to the ODBC driver using `SQLDriverConnect`. `SQLDriverConnect` accepts a connection string in which you can configure the database using any of the available DuckDB configuration options.

```
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);

SQLCHAR str[1024];
SQLSMALLINT strl;
std::string dsn = "DSN=DuckDB;allow_unsigned_extensions=true;access_mode=READ_ONLY"
SQLDriverConnect(dbc, nullptr, (SQLCHAR*)dsn.c_str(), SQL_NTS, str, sizeof(str), &strl, SQL_DRIVER_
COMPLETE)

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)
```

Performance

Performance Guide

DuckDB aims to automatically achieve high performance by using well-chosen default configurations and having a forgiving architecture. Of course, there are still opportunities for tuning the system for specific workloads. The Performance Guide's page contain guidelines and tips for achieving good performance when loading and processing data with DuckDB.

The guides include several microbenchmarks. You may find details about these on the Benchmarks page.

Data Import

Recommended Import Methods

When importing data from other systems to DuckDB, there are several considerations to take into account. We recommend importing using the following order:

1. For systems which are supported by a DuckDB scanner extension, it's preferable to use the scanner. DuckDB currently offers scanners for MySQL, PostgreSQL, and SQLite.
2. If there is a bulk export feature in the data source system, export the data to Parquet or CSV format, then load it using DuckDB's Parquet or CSV loader.
3. If the approaches above are not applicable, consider using the DuckDB appender, currently available in the C, C++, Go, Java, and Rust APIs.
4. If the data source system supports Apache Arrow and the data transfer is a recurring task, consider using the DuckDB Arrow extension.

Methods to Avoid

If possible, avoid looping row-by-row (tuple-at-a-time) in favor of bulk operations. Performing row-by-row inserts (even with prepared statements) is detrimental to performance and will result in slow load times.

> **Best practice.** Unless your data is small (<100k rows), avoid using inserts in loops.

Schema

Types

It is important to use the correct type for encoding columns (e.g., `BIGINT`, `DATE`, `DATETIME`). While it is always possible to use string types (VARCHAR, etc.) to encode more specific values, this is not recommended. Strings use more space and are slower to process in operations such as filtering, join, and aggregation.

When loading CSV files, you may leverage the CSV reader's auto-detection mechanism to get the correct types for CSV inputs.

If you run in a memory-constrained environment, using smaller data types (e.g., `TINYINT`) can reduce the amount of memory and disk space required to complete a query. DuckDB's bitpacking compression means small values stored in larger data types will not take up larger sizes on disk, but they will take up more memory during processing.

> **Best practice.** Use the most restrictive types possible when creating columns. Avoid using strings for encoding more specific data items.

Microbenchmark: Using Timestamps

We illustrate the difference in aggregation speed using the `creationDate` column of the LDBC Comment table on scale factor 300. This table has approx. 554 million unordered timestamp values. We run a simple aggregation query that returns the average day-of-the month from the timestamps in two configurations.

First, we use a `DATETIME` to encode the values and run the query using the `extract` datetime function:

```
SELECT avg(extract('day' FROM creationDate)) FROM Comment;
```

Second, we use the `VARCHAR` type and use string operations:

```
SELECT avg(CAST(creationDate[9:10] AS INTEGER)) FROM Comment;
```

The results of the microbenchmark are as follows:

Column type	Storage size	Query time
`DATETIME`	3.3 GB	0.9 s
`VARCHAR`	5.2 GB	3.9 s

The results show that using the `DATETIME` value yields smaller storage sizes and faster processing.

Microbenchmark: Joining on Strings

We illustrate the difference caused by joining on different types by computing a self-join on the LDBC Comment table at scale factor 100. The table has 64-bit integer identifiers used as the `id` attribute of each row. We perform the following join operation:

```
SELECT count(*) AS count
FROM Comment c1
JOIN Comment c2 ON c1.ParentCommentId = c2.id;
```

In the first experiment, we use the correct (most restrictive) types, i.e., both the `id` and the `ParentCommentId` columns are defined as `BIGINT`. In the second experiment, we define all columns with the `VARCHAR` type. While the results of the queries are the same for all both experiments, their runtime vary significantly. The results below show that joining on `BIGINT` columns is approx. 1.8× faster than performing the same join on `VARCHAR`-typed columns encoding the same value.

Join column payload type	Join column schema type	Example value	Query time
`BIGINT`	`BIGINT`	`70368755640078`	1.2 s
`BIGINT`	`VARCHAR`	`'70368755640078'`	2.1 s

> **Best practice.** Avoid representing numeric values as strings, especially if you intend to perform e.g., join operations on them.

Constraints

DuckDB allows defining constraints such as `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY`. These constraints can be beneficial for ensuring data integrity but they have a negative effect on load performance as they necessitate building indexes and performing checks. Moreover, they *very rarely improve the performance of queries* as DuckDB does not rely on these indexes for join and aggregation operators (see indexing for more details).

> **Best practice.** Do not define constraints unless your goal is to ensure data integrity.

Microbenchmark: The Effect of Primary Keys

We illustrate the effect of using primary keys with the LDBC Comment table at scale factor 300. This table has approx. 554 million entries. We first create the schema without a primary key, then load the data. In the second experiment, we create the schema with a primary key, then load the data. In both cases, we take the data from `.csv.gz` files, and measure the time required to perform the loading.

Operation	Execution time
Load without primary key	92.2 s
Load with primary key	286.8 s

In this case, primary keys will only have a (small) positive effect on highly selective queries such as when filtering on a single identifier. They do not have an effect on join and aggregation operators.

> **Best practice.** For best bulk load performance, avoid defining primary key constraints if possible.

Indexing

DuckDB has two types of indexes: zonemaps and ART indexes.

Zonemaps

DuckDB automatically creates zonemaps (also known as min-max indexes) for the columns of all general-purpose data types. These indexes are used for predicate pushdown into scan operators and computing aggregations. This means that if a filter criterion (like `WHERE column1 = 123`) is in use, DuckDB can skip any row group whose min-max range does not contain that filter value (e.g., a block with a min-max range of 1000 to 2000 will be omitted when comparing for `= 123` or `< 400`).

The Effect of Ordering on Zonemaps

The more ordered the data within a column, the more useful the zonemap indexes will be. For example, in the worst case, a column could contain a random number on every row. DuckDB will be unlikely to be able to skip any row groups. The best case of ordered data commonly arises with `DATETIME` columns. If specific columns will be queried with selective filters, it is best to pre-order data by those columns when inserting it. Even an imperfect ordering will still be helpful.

Microbenchmark: The Effect of Ordering

For an example, let's repeat the microbenchmark for timestamps with a timestamp column that sorted using an ascending order vs. an unordered one.

Column type	Ordered	Storage size	Query time
`DATETIME`	yes	1.3 GB	0.6 s
`DATETIME`	no	3.3 GB	0.9 s

The results show that simply keeping the column order allows for improved compression, yielding a 2.5x smaller storage size. It also allows the computation to be 1.5x faster.

Ordered Integers

Another practical way to exploit ordering is to use the `INTEGER` type with automatic increments rather than `UUID` for columns that will be queried using selective filters. `UUID`s will likely be inserted in a random order, so many row groups in the table will need to be scanned to find a specific `UUID` value, while an ordered `INTEGER` column will allow all row groups to be skipped except the one that contains the value.

ART Indexes

DuckDB allows defining Adaptive Radix Tree (ART) indexes in two ways. First, such an index is created implicitly for columns with `PRIMARY KEY`, `FOREIGN KEY`, and `UNIQUE` constraints. Second, explicitly running a the `CREATE INDEX` statement creates an ART index on the target column(s).

The tradeoffs of having an ART index on a column are as follows:

1. It enables efficient constraint checking upon changes (inserts, updates, and deletes) for non-bulky changes.
2. Having an ART index makes changes to the affected column(s) slower compared to non-indexed performance. That is because of index maintenance for these operations.

Regarding query performance, an ART index has the following effects:

1. It speeds up point queries and other highly selective queries using the indexed column(s), where the filtering condition returns approx. 0.1% of all rows or fewer. When in doubt, use `EXPLAIN` to verify that your query plan uses the index scan.
2. An ART index has no effect on the performance of join, aggregation, and sorting queries.

Indexes are serialized to disk and deserialized lazily, i.e., when the database is reopened, operations using the index will only load the required parts of the index. Therefore, having an index will not cause any slowdowns when opening an existing database.

> **Best practice.** We recommend following these guidelines:
>
> - Only use primary keys, foreign keys, or unique constraints, if these are necessary for enforcing constraints on your data.
> - Do not define explicit indexes unless you have highly selective queries.
> - If you define an ART index, do so after bulk loading the data to the table. Adding an index prior to loading, either explicitly or via primary/foreign keys, is detrimental to load performance.

Environment

The environment where DuckDB is run has an obvious impact on performance. This page focuses on the effects of the hardware configuration and the operating system used.

Hardware Configuration

CPU and Memory

As a rule of thumb, DuckDB requires a **minimum** of 125 MB of memory per thread. For example, if you use 8 threads, you need at least 1 GB of memory. For ideal performance, aggregation-heavy workloads require approx. 5 GB memory per thread and join-heavy workloads require approximately 10 GB memory per thread.

> **Best practice.** Aim for 5-10 GB memory per thread.

> **Tip.** If you have a limited amount of memory, try to limit the number of threads, e.g., by issuing `SET threads = 4;`.

Disk

DuckDB is capable of operating both as an in-memory and as a disk-based database system. In the latter case, it can spill to disk to process larger-than-memory workloads (a.k.a. out-of-core processing). In these cases, a fast disk is highly beneficial. However, if the workload fits in memory, the disk speed only has a limited effect on performance.

In general, network-based storage will result in slower DuckDB workloads than using local disks. This includes network disks such as NFS, network drives such as SMB and Samba, and network-backed cloud disks such as AWS EBS. However, different network disks can have vastly varying IO performance, ranging from very slow to almost as fast as local. Therefore, for optimal performance, only use network disks that can provide high IO performance.

> **Best practice.** Fast disks are important if your workload is larger than memory and/or fast data loading is important. Only use network-backed disks if they guarantee high IO.

Operating System

We recommend using the latest stable version of operating systems: macOS, Windows, and Linux are all well-tested and DuckDB can run on them with high performance. Among Linux distributions, we recommended using Ubuntu Linux LTS due to its stability and the fact that most of DuckDB's Linux test suite jobs run on Ubuntu workers.

File Formats

Handling Parquet Files

DuckDB has advanced support for Parquet files, which includes directly querying Parquet files. When deciding on whether to query these files directly or to first load them to the database, you need to consider several factors.

Reasons for Querying Parquet Files

Availability of basic statistics: Parquet files use a columnar storage format and contain basic statistics such as zonemaps. Thanks to these features, DuckDB can leverage optimizations such as projection and filter pushdown on Parquet files. Therefore, workloads that combine projection, filtering, and aggregation tend to perform quite well when run on Parquet files.

Storage considerations: Loading the data from Parquet files will require approximately the same amount of space for the DuckDB database file. Therefore, if the available disk space is constrained, it is worth running the queries directly on Parquet files.

Reasons against Querying Parquet Files

Lack of advanced statistics: The DuckDB database format has the hyperloglog statistics that Parquet files do not have. These improve the accuracy of cardinality estimates, and are especially important if the queries contain a large number of join operators.

Tip. If you find that DuckDB produces a suboptimal join order on Parquet files, try loading the Parquet files to DuckDB tables. The improved statistics likely help obtain a better join order.

Repeated queries: If you plan to run multiple queries on the same data set, it is worth loading the data into DuckDB. The queries will always be somewhat faster, which over time amortizes the initial load time.

High decompression times: Some Parquet files are compressed using heavyweight compression algorithms such as gzip. In these cases, querying the Parquet files will necessitate an expensive decompression time every time the file is accessed. Meanwhile, lightweight compression methods like Snappy, LZ4, and zstd, are faster to decompress. You may use the `parquet_metadata` function to find out the compression algorithm used.

Microbenchmark: Running TPC-H on a DuckDB Database vs. Parquet

The queries on the TPC-H benchmark run approximately 1.1-5.0x slower on Parquet files than on a DuckDB database.

> **Best practice.** If you have the storage space available, and have a join-heavy workload and/or plan to run many queries on the same dataset, load the Parquet files into the database first. The compression algorithm and the row group sizes in the Parquet files have a large effect on performance: study these using the `parquet_metadata` function.

The Effect of Row Group Sizes

DuckDB works best on Parquet files with row groups of 100K-1M rows each. The reason for this is that DuckDB can only parallelize over row groups – so if a Parquet file has a single giant row group it can only be processed by a single thread. You can use the `parquet_metadata` function to figure out how many row groups a Parquet file has. When writing Parquet files, use the `row_group_size` option.

Microbenchmark: Running Aggregation Query at Different Row Group Sizes

We run a simple aggregation query over Parquet files using different row group sizes, selected between 960 and 1,966,080. The results are as follows.

Row group size	Execution time
960	8.77s
1920	8.95s
3840	4.33s
7680	2.35s
15360	1.58s
30720	1.17s
61440	0.94s
122880	0.87s
245760	0.93s
491520	0.95s
983040	0.97s
1966080	0.88s

The results show that row group sizes <5,000 have a strongly detrimental effect, making runtimes more than 5-10x larger than ideally-sized row groups, while row group sizes between 5,000 and 20,000 are still 1.5-2.5x off from best performance. Above row group size of 100,000, the differences are small: the gap is about 10% between the best and the worst runtime.

Parquet File Sizes

DuckDB can also parallelize across multiple Parquet files. It is advisable to have at least as many total row groups across all files as there are CPU threads. For example, with a machine having 10 threads, both 10 files with 1 row group or 1 file with 10 row groups will achieve full parallelism. It is also beneficial to keep the size of individual Parquet files moderate.

> **Best practice.** The ideal range is between 100MB and 10GB per individual Parquet file.

Hive Partitioning for Filter Pushdown

When querying many files with filter conditions, performance can be improved by using a Hive-format folder structure to partition the data along the columns used in the filter condition. DuckDB will only need to read the folders and files that meet the filter criteria. This can be especially helpful when querying remote files.

More Tips on Reading and Writing Parquet Files

For tips on reading and writing Parquet files, see the Parquet Tips page.

Loading CSV Files

CSV files are often distributed in compressed format such as GZIP archives (`.csv.gz`). DuckDB can decompress these files on the fly. In fact, this is typically faster than decompressing the files first and loading them due to reduced IO.

Schema	Load Time
Load from GZIP-compressed CSV files (`.csv.gz`)	107.1s
Decompressing (using parallel `gunzip`) and loading from decompressed CSV files	121.3s

Loading Many Small CSV Files

The CSV reader runs the CSV sniffer on all files. For many small files, this may cause an unnecessarily high overhead. A potential optimization to speed this up is to turn the sniffer off. Assuming that all files have the same CSV dialect and colum names/types, get the sniffer options as follows:

```
.mode line
SELECT Prompt FROM sniff_csv('part-0001.csv');
```

```
Prompt = FROM read_csv('file_path.csv', auto_detect=false, delim=',', quote='"', escape='"', new_line='\n',
skip=0, header=true, columns={'hello': 'BIGINT', 'world': 'VARCHAR'});
```

Then, you can adjust `read_csv` command, by e.g., applying filename expansion (globbing), and run with the rest of the options detected by the sniffer:

```
FROM read_csv('part-*.csv', auto_detect=false, delim=',', quote='"', escape='"', new_line='\n', skip=0,
header=true, columns={'hello': 'BIGINT', 'world': 'VARCHAR'});
```

Tuning Workloads

Parallelism (Multi-Core Processing)

The Effect of Row Groups on Parallelism

DuckDB parallelizes the workload based on *row groups*, i.e., groups of rows that are stored together at the storage level. A row group in DuckDB's database format consists of max. 122,880 rows. Parallelism starts at the level of row groups, therefore, for a query to run on k threads, it needs to scan at least k * 122,880 rows.

Too Many Threads

Note that in certain cases DuckDB may launch *too many threads* (e.g., due to HyperThreading), which can lead to slowdowns. In these cases, it's worth manually limiting the number of threads using `SET threads = X`.

Larger-Than-Memory Workloads (Out-of-Core Processing)

A key strength of DuckDB is support for larger-than-memory workloads, i.e., it is able to process data sets that are larger than the available system memory (also known as *out-of-core processing*). It can also run queries where the intermediate results cannot fit into memory. This section explains the prerequisites, scope, and known limitations of larger-than-memory processing in DuckDB.

Spilling to Disk

Larger-than-memory workloads are supported by spilling to disk. If DuckDB is connected to a persistent database file, DuckDB will create a temporary directory named `<database_file_name>.tmp` when the available memory is no longer sufficient to continue processing.

If DuckDB is running in in-memory mode, it cannot use disk to offload data if it does not fit into main memory. To enable offloading in the absence of a persistent database file, use the `SET temp_directory` statement:

```
SET temp_directory = '/path/to/temp_dir.tmp/';
```

Blocking Operators

Some operators cannot output a single row until the last row of their input has been seen. These are called *blocking operators* as they require their entire input to be buffered, and are the most memory-intensive operators in relational database systems. The main blocking operators are the following:

- *sorting:* `ORDER BY`,
- *grouping:* `GROUP BY`,
- *windowing:* `OVER ... (PARTITION BY ... ORDER BY ...)`,
- *joining:* `JOIN`.

DuckDB supports larger-than-memory processing for all of these operators.

Limitations

DuckDB strives to always complete workloads even if they are larger-than-memory. That said, there are some limitations at the moment:

- If multiple blocking operators appear in the same query, DuckDB may still throw an out-of-memory exception due to the complex interplay of these operators.
- Some aggregate functions, such as `list()` and `string_agg()`, do not support offloading to disk.
- Aggregate functions that use sorting are holistic, i.e., they need all inputs before the aggregation can start. As DuckDB cannot yet offload some complex intermediate aggregate states to disk, these functions can cause an out-of-memory exception when run on large data sets.
- The `PIVOT` operation internally uses the `list()` function, therefore it is subject to the same limitation.

Profiling

If your queries are not performing as well as expected, it's worth studying their query plans:

- Use `EXPLAIN` to print the physical query plan without running the query.

- Use `EXPLAIN ANALYZE` to run and profile the query. This will show the CPU time that each step in the query takes. Note that due to multi-threading, adding up the individual times will be larger than the total query processing time.

Query plans can point to the root of performance issues. A few general directions:

- Avoid nested loop joins in favor of hash joins.
- A scan that does not include a filter pushdown for a filter condition that is later applied performs unnecessary IO. Try rewriting the query to apply a pushdown.
- Bad join orders where the cardinality of an operator explodes to billions of tuples should be avoided at all costs.

Prepared Statements

Prepared statements can improve performance when running the same query many times, but with different parameters. When a statement is prepared, it completes several of the initial portions of the query execution process (parsing, planning, etc.) and caches their output. When it is executed, those steps can be skipped, improving performance. This is beneficial mostly for repeatedly running small queries (with a runtime of < 100ms) with different sets of parameters.

Note that it is not a primary design goal for DuckDB to quickly execute many small queries concurrently. Rather, it is optimized for running larger, less frequent queries.

Querying Remote Files

DuckDB uses synchronous IO when reading remote files. This means that each DuckDB thread can make at most one HTTP request at a time. If a query must make many small requests over the network, increasing DuckDB's `threads` setting to larger than the total number of CPU cores (approx. 2-5 times CPU cores) can improve parallelism and performance.

Avoid Reading Unnecessary Data

The main bottleneck in workloads reading remote files is likely to be the IO. This means that minimizing the unnecessarily read data can be highly beneficial.

Some basic SQL tricks can help with this:

- Avoid `SELECT *`. Instead, only select columns that are actually used. DuckDB will try to only download the data it actually needs.
- Apply filters on remote parquet files when possible. DuckDB can use these filters to reduce the amount of data that is scanned.
- Either sort or partition data by columns that are regularly used for filters: this increases the effectiveness of the filters in reducing IO.

To inspect how much remote data is transferred for a query, `EXPLAIN ANALYZE` can be used to print out the total number of requests and total data transferred for queries on remote files.

Avoid Reading Data More Than Once

DuckDB does not cache data from remote files automatically. This means that running a query on a remote file twice will download the required data twice. So if data needs to be accessed multiple times, storing it locally can make sense. To illustrate this, lets look at an example:

Consider the following queries:

```
SELECT col_a + col_b FROM 's3://bucket/file.parquet' WHERE col_a > 10;
SELECT col_a * col_b FROM 's3://bucket/file.parquet' WHERE col_a > 10;
```

These queries download the columns `col_a` and `col_b` from `s3://bucket/file.parquet` twice. Now consider the following queries:

```
CREATE TABLE local_copy_of_file AS
    SELECT col_a, col_b FROM 's3://bucket/file.parquet' WHERE col_a > 10;

SELECT col_a + col_b FROM local_copy_of_file;
SELECT col_a * col_b FROM local_copy_of_file;
```

Here DuckDB will first copy `col_a` and `col_b` from `s3://bucket/file.parquet` into a local table, and then query the local in-memory columns twice. Note also that the filter `WHERE col_a > 10` is also now applied only once.

An important side note needs to be made here though. The first two queries are fully streaming, with only a small memory footprint, whereas the second requires full materialization of columns `col_a` and `col_b`. This means that in some rare cases (e.g., with a high-speed network, but with very limited memory available) it could actually be beneficial to download the data twice.

Best Practices for Using Connections

DuckDB will perform best when reusing the same database connection many times. Disconnecting and reconnecting on every query will incur some overhead, which can reduce performance when running many small queries. DuckDB also caches some data and metadata in memory, and that cache is lost when the last open connection is closed. Frequently, a single connection will work best, but a connection pool may also be used.

Using multiple connections can parallelize some operations, although it is typically not necessary. DuckDB does attempt to parallelize as much as possible within each individual query, but it is not possible to parallelize in all cases. Making multiple connections can process more operations concurrently. This can be more helpful if DuckDB is not CPU limited, but instead bottlenecked by another resource like network transfer speed.

The `preserve_insertion_order` Option

When importing or exporting data sets (from/to the Parquet or CSV formats), which are much larger than the available memory, an out of memory error may occur:

```
Error: Out of Memory Error: failed to allocate data of size ... (.../... used)
```

In these cases, consider setting the `preserve_insertion_order` configuration option to `false`:

```
SET preserve_insertion_order = false;
```

This allows the systems to re-order any results that do not contain `ORDER BY` clauses, potentially reducing memory usage.

My Workload Is Slow

If you find that your workload in DuckDB is slow, we recommend performing the following checks. More detailed instructions are linked for each point.

1. Do you have enough memory? DuckDB works best if you have 5-10GB memory per CPU core.
2. Are you using a fast disk? Network-attached disks can cause the workload to slow down, especially for larger than memory workloads.
3. Are you using indexes or constraints (primary key, unique, etc.)? If possible, try disabling them, which boosts load and update performance.
4. Are you using the correct types? For example, use `TIMESTAMP` to encode datetime values.
5. Are you reading from Parquet files? If so, do they have row group sizes between 100k and 1M and file sizes between 100MB to 10GB?
6. Does the query plan look right? Study it with `EXPLAIN`.
7. Is the workload running in parallel? Use `htop` or the operating system's task manager to observe this.
8. Is DuckDB using too many threads? Try limiting the amount of threads.

Are you aware of other common issues? If so, please click the *Report content issue* link below and describe them along with their workarounds.

Benchmarks

For several of the recommendations in our performance guide, we use microbenchmarks to back up our claims. For these benchmarks, we use data sets from the TPC-H benchmark and the LDBC Social Network Benchmark's BI workload.

Data Sets

We use the LDBC BI SF300 data set's Comment table (20GB `.tar.zst` archive, 21GB when decompressed into `.csv.gz` files), while others use the same table's `creationDate` column (4GB `.parquet` file).

The TPC data sets used in the benchmark are generated with the DuckDB tpch extension.

A Note on Benchmarks

Running fair benchmarks is difficult, especially when performing system-to-system comparison. When running benchmarks on DuckDB, please make sure you are using the latest version (preferably the nightly build). If in doubt about your benchmark results, feel free to contact us at `gabor@duckdb.org`.

Disclaimer on Benchmarks

Note that the benchmark results presented in this guide do not constitute official TPC or LDBC benchmark results. Instead, they merely use the data sets of and some queries provided by the TPC-H and the LDBC BI benchmark frameworks, and omit other parts of the workloads such as updates.

Python

Installing the Python Client

Installing via Pip

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
```

Installing from Source

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
```

For detailed instructions on how to compile DuckDB from source, see the Building guide.

Executing SQL in Python

SQL queries can be executed using the `duckdb.sql` function.

```
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 Colab link above for an in-notebook example:

```
pip install duckdb
```

Install Jupyter Notebook

```
pip install notebook
```

Or JupyterLab:

```
pip install jupyterlab
```

Install supporting libraries:

```
pip install jupysql pandas matplotlib 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 database:

```
%sql duckdb:///:memory:
```

```
%sql duckdb:///:default:
```

```
%sql duckdb:///path/to/file.db
```

> 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 DataFrame.

```
%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 DataFrame.

```
%%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

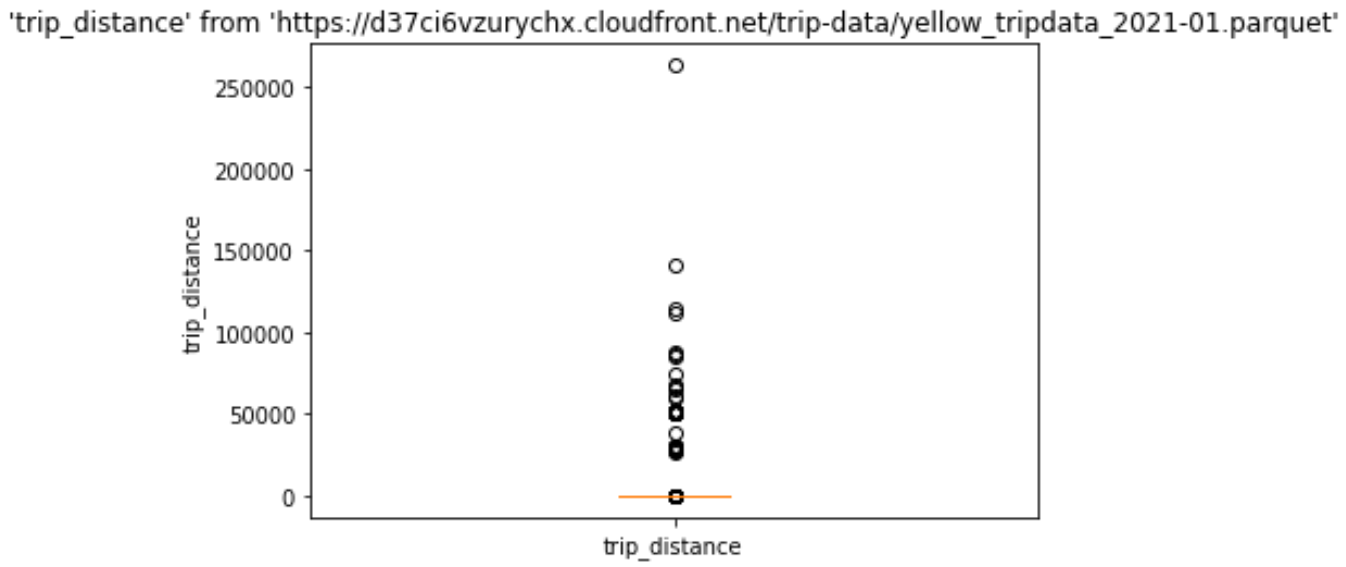
DuckDB's httpfs extension allows Parquet and CSV files to be queried remotely over http. These examples query a Parquet file that contains historical taxi data from NYC. Using the Parquet format allows DuckDB to only pull the rows and columns into memory that are needed rather than downloading the entire 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
```

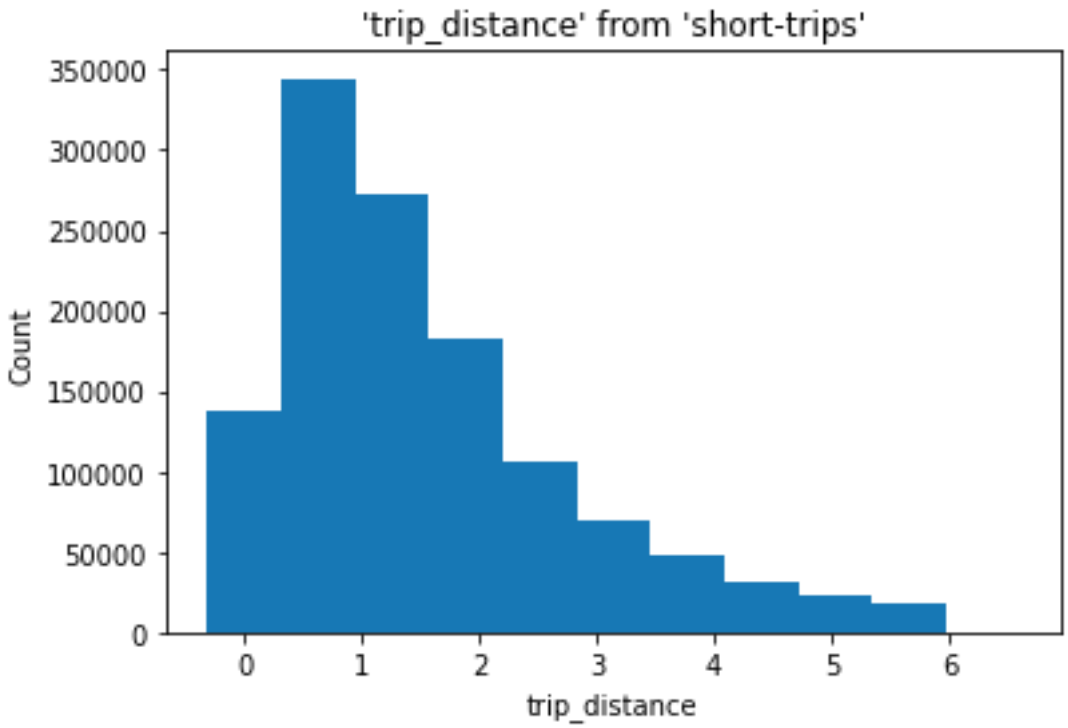


Now, create a query that filters by the 90th percentile. Note the use of the `--save`, and `--no-execute` functions. This tells JupySQL to store the query, but skips execution. It will be referenced in the next plotting call.

```
%%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()
```

The seamless integration of Pandas DataFrames to DuckDB SQL queries is allowed by replacement scans, which replace instances of accessing the `my_df` table (which does not exist in DuckDB) with a table function that reads the `my_df` dataframe.

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. There is no need to register the DataFrames manually – DuckDB can find them in the Python process by name thanks to replacement scans.

```
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")
```

If the order of columns is different or not all columns are present in the DataFrame, use `INSERT INTO ... BY NAME`:

```
duckdb.sql("INSERT INTO my_table BY NAME SELECT * FROM my_df")
```

See Also

DuckDB also supports exporting to Pandas.

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()
results
```

```
   42
0  42
```

See Also

DuckDB also supports importing from Pandas.

Import from Numpy

It is possible to query Numpy arrays from DuckDB. There is no need to register the arrays manually – DuckDB can find them in the Python process by name thanks to replacement scans. For example:

```
import duckdb
import numpy as np

my_arr = np.array([(1, 9.0), (2, 8.0), (3, 7.0)])

duckdb.sql("SELECT * FROM my_arr")
```

```
┌─────────┬─────────┬─────────┐
│ column0 │ column1 │ column2 │
│ double  │ double  │ double  │
├─────────┼─────────┼─────────┤
```

```
┌─────────┬─────────┬─────────┐
│   1.0   │   2.0   │   3.0   │
│   9.0   │   8.0   │   7.0   │
└─────────┴─────────┴─────────┘
```

See Also

DuckDB also supports exporting to Numpy.

Export to Numpy

The result of a query can be converted to a Numpy array using the `fetchnumpy()` function. For example:

```
import duckdb
import numpy as np

my_arr = duckdb.sql("SELECT unnest([1, 2, 3]) AS x, 5.0 AS y").fetchnumpy()
my_arr
```

```
{'x': array([1, 2, 3], dtype=int32), 'y': masked_array(data=[5.0, 5.0, 5.0],
             mask=[False, False, False],
       fill_value=1e+20)}
```

Then, the array can be processed using Numpy functions, e.g.:

```
np.sum(my_arr['x'])
```

```
6
```

See Also

DuckDB also supports importing from Numpy.

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 on 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.

```python
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 WHERE i >= 2 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.

```python
import duckdb
from threading import Thread, current_thread
import random

duckdb_con = duckdb.connect('my_peristent_db.duckdb')
# Use connect without parameters for an in-memory database
# duckdb_con = duckdb.connect()
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())
```

Integration 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 Gentoo penguins that have a body mass larger than 6 kg.

```
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
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 Adelie Adelie Gentoo Chinstrap	Torgersen Biscoe Dream Biscoe 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.

Integration 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)
```

```
shape: (3, 2)
┌─────┬────────┐
│ id  ┆ fruit  │
│ --- ┆ ---    │
│ i32 ┆ str    │
╞═════╪════════╡
│ 1   ┆ banana │
│ 2   ┆ apple  │
│ 3   ┆ mango  │
└─────┴────────┘
```

To learn more about Polars, feel free to explore their Python API Reference.

Using fsspec 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-compatible 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
```

```python
# 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('gcs:///bucket/file.csv')")
```

> 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 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).

Installing DBeaver

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.

SQL Features

Friendly SQL

DuckDB offers several advanced SQL features as well as extensions to the SQL syntax. We call these colloquially as ”friendly SQL”.

> Several of these features are also supported in other systems while some are (currently) exclusive to DuckDB.

Clauses

- `CREATE OR REPLACE TABLE`
- `CREATE TABLE ... AS SELECT` (CTAS)
- `DESCRIBE`
- `FROM`-first syntax with an optional `SELECT` clause
- `GROUP BY ALL`
- `INSERT INTO ... BY NAME`
- `ORDER BY ALL`
- `PIVOT` `UNPIVOT`
- `SELECT * EXCLUDE`
- `SELECT * REPLACE`
- `SUMMARIZE`
- `UNION BY NAME`

Query Features

- Column aliases in `WHERE`, `GROUP BY`, and `HAVING`
- `COLUMNS()` expression
  - with regular expressions
  - with `EXCLUDE` and `REPLACE`
  - with lambda functions
- Reusable column aliases, e.g.: `SELECT i + 1 AS j, j + 2 AS k FROM range(0, 3) t(i)`

Literals and Identifiers

- Case-insensitivity while maintaining case of entities in the catalog
- Deduplicating identifiers
- Underscores as digit separators in numeric literals

Data Types

- `MAP` data type
- `UNION` data type

Data Import

- Auto-detecting the headers and schema of CSV files
- Directly querying CSV files and Parquet files
- Loading from files using the syntax `FROM 'my.csv'`, `FROM 'my.csv.gz'`, `FROM 'my.parquet'`, etc.
- Filename expansion (globbing), e.g.: `FROM 'my-data/part-*.parquet'`

Functions and Expressions

- Dot operator for function chaining: `SELECT ('hello').upper()`
- String formatters: `format()` function with the `fmt` syntax and the `printf() function`
- List comprehensions
- List slicing
- String slicing
- `STRUCT.*` notation
- Simple `LIST` and `STRUCT` creation

Join Types

- ASOF joins
- `LATERAL` joins
- `POSITIONAL` joins

Trailing Commas

DuckDB allows trailing commas, both when listing entities (e.g., column and table names) and when constructing `LIST` items. For example, the following query works:

```
SELECT
    42 AS x,
    ['a', 'b', 'c',] AS y,
    'hello world' AS z,
;
```

See Also

- Friendlier SQL with DuckDB blog post
- Even Friendlier SQL with DuckDB blog post
- SQL Gymnastics: Bending SQL into flexible new shapes blog post

AsOf Join

What is an AsOf Join?

Time series data is not always perfectly aligned. Clocks may be slightly off, or there may be a delay between cause and effect. This can make connecting two sets of ordered data challenging. AsOf joins are a tool for solving this and other similar problems.

One of the problems that AsOf joins are used to solve is finding the value of a varying property at a specific point in time. This use case is so common that it is where the name came from:

*Give me the value of the property **as of this time**.*

More generally, however, AsOf joins embody some common temporal analytic semantics, which can be cumbersome and slow to implement in standard SQL.

Portfolio Example Data Set

Let's start with a concrete example. Suppose we have a table of stock `prices` with timestamps:

ticker	when	price
APPL	2001-01-01 00:00:00	1
APPL	2001-01-01 00:01:00	2
APPL	2001-01-01 00:02:00	3
MSFT	2001-01-01 00:00:00	1
MSFT	2001-01-01 00:01:00	2
MSFT	2001-01-01 00:02:00	3
GOOG	2001-01-01 00:00:00	1
GOOG	2001-01-01 00:01:00	2
GOOG	2001-01-01 00:02:00	3

We have another table containing portfolio `holdings` at various points in time:

ticker	when	shares
APPL	2000-12-31 23:59:30	5.16
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	24.13
GOOG	2000-12-31 23:59:30	9.33
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	10.58
DATA	2000-12-31 23:59:30	6.65
DATA	2001-01-01 00:00:30	17.95
DATA	2001-01-01 00:01:30	18.37

To load these tables to DuckDB, run:

```
CREATE TABLE prices AS FROM 'https://duckdb.org/data/prices.csv';
CREATE TABLE holdings AS FROM 'https://duckdb.org/data/holdings.csv';
```

Inner AsOf Joins

We can compute the value of each holding at that point in time by finding the most recent price before the holding's timestamp by using an AsOf Join:

```
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h
ASOF JOIN prices p
       ON h.ticker = p.ticker
      AND h.when >= p.when;
```

This attaches the value of the holding at that time to each row:

ticker	when	value
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	48.26
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	21.16

It essentially executes a function defined by looking up nearby values in the `prices` table. Note also that missing `ticker` values do not have a match and don't appear in the output.

Outer AsOf Joins

Because AsOf produces at most one match from the right hand side, the left side table will not grow as a result of the join, but it could shrink if there are missing times on the right. To handle this situation, you can use an *outer* AsOf Join:

```
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h
ASOF LEFT JOIN prices p
           ON h.ticker = p.ticker
          AND h.when >= p.when
ORDER BY ALL;
```

As you might expect, this will produce `NULL` prices and values instead of dropping left side rows when there is no ticker or the time is before the prices begin.

ticker	when	value
APPL	2000-12-31 23:59:30	
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	48.26
GOOG	2000-12-31 23:59:30	
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	21.16
DATA	2000-12-31 23:59:30	
DATA	2001-01-01 00:00:30	
DATA	2001-01-01 00:01:30	

AsOf Joins with the `USING` Keyword

So far we have been explicit about specifying the conditions for AsOf, but SQL also has a simplified join condition syntax for the common case where the column names are the same in both tables. This syntax uses the `USING` keyword to list the fields that should be compared for equality. AsOf also supports this syntax, but with two restrictions:

- The last field is the inequality
- The inequality is >= (the most common case)

Our first query can then be written as:

```
SELECT ticker, h.when, price * shares AS value
FROM holdings h
ASOF JOIN prices p USING (ticker, when);
```

Be aware that if you don't explicitly list the columns in the SELECT, the ordering field value will be the probe value, not the build value. For a natural join, this is not an issue because all the conditions are equalities, but for AsOf, one side has to be chosen. Since AsOf can be viewed as a lookup function, it is more natural to return the "function arguments" than the function internals.

See Also

For implementation details, see the blog post "DuckDB's AsOf joins: Fuzzy Temporal Lookups".

Full-Text Search

DuckDB supports full-text search via the `fts` extension. A full-text index allows for a query to quickly search for all occurrences of individual words within longer text strings.

Example: Shakespeare Corpus

Here's an example of building a full-text index of Shakespeare's plays.

```
CREATE TABLE corpus AS
    SELECT * FROM 'https://blobs.duckdb.org/data/shakespeare.parquet';
```

```
DESCRIBE corpus;
```

column_name	column_type	null	key	default	extra
line_id	VARCHAR	YES	NULL	NULL	NULL
play_name	VARCHAR	YES	NULL	NULL	NULL
line_number	VARCHAR	YES	NULL	NULL	NULL
speaker	VARCHAR	YES	NULL	NULL	NULL
text_entry	VARCHAR	YES	NULL	NULL	NULL

The text of each line is in `text_entry`, and a unique key for each line is in `line_id`.

Creating a Full-Text Search Index

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.

```
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?

```sql
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 DESC;
```

score	line_id	play_name	speaker	text_entry
4.427313429798464	H4/2.4.494	Henry IV	Carrier	As fat as butter.
3.836270302568675	H4/1.2.21	Henry IV	FALSTAFF	prologue to an egg and butter.
3.836270302568675	H4/2.1.55	Henry IV	Chamberlain	They are up already, and call for eggs and butter;
3.3844488405497115	H4/4.2.21	Henry IV	FALSTAFF	toasts-and-butter, with hearts in their bellies no
3.3844488405497115	H4/4.2.62	Henry IV	PRINCE HENRY	already made thee butter. But tell me, Jack, whose
3.3844488405497115	AWW/4.1.40	Alls well that ends well	PAROLLES	butter-womans mouth and buy myself another of
3.3844488405497115	AYLI/3.2.93	As you like it	TOUCHSTONE	right butter-womens rank to market.
3.3844488405497115	KL/2.4.132	King Lear	Fool	kindness to his horse, buttered his hay.
3.0278411214953107	AWW/5.2.9	Alls well that ends well	Clown	henceforth eat no fish of fortunes buttering.
3.0278411214953107	MWW/2.2.260	Merry Wives of Windsor	FALSTAFF	Hang him, mechanical salt-butter rogue! I will
3.0278411214953107	MWW/2.2.284	Merry Wives of Windsor	FORD	rather trust a Fleming with my butter, Parson Hugh
3.0278411214953107	MWW/3.5.7	Merry Wives of Windsor	FALSTAFF	Ill have my brains taen out and buttered, and give
3.0278411214953107	MWW/3.5.102	Merry Wives of Windsor	FALSTAFF	to heat as butter; a man of continual dissolution
2.739219044070792	H4/2.4.115	Henry IV	PRINCE HENRY	Didst thou never see Titan kiss a dish of butter?

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

For more details, see the "Generating a Shakespeare corpus for full-text searching from JSON" blog post

- 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.

Snippets

Create Synthetic Data

DuckDB allows you to quickly generate synthetic data sets. To do so, you may use:

- range functions
- hash functions, e.g., `hash`, `md5`, `sha256`
- the Faker Python package via the Python function API
- using cross products (Cartesian products)

For example:

```
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", side_effects=True)
res = duckdb.sql("""
                 SELECT hash(i * 10 + j) AS id, random_date() AS creationDate, IF (j % 2, true, false)
                 FROM generate_series(1, 5) s(i)
                 CROSS JOIN generate_series(1, 2) t(j)
                 """)
res.show()
```

This generates the following:

id uint64	creationDate date	flag boolean
6770051751173734325	2019-11-05	true
16510940941872865459	2002-08-03	true
13285076694688170502	1998-11-27	true
11757770452869451863	1998-07-03	true
2064835973596856015	2010-09-06	true
17776805813723356275	2020-12-26	false
13540103502347468651	1998-03-21	false
4800297459639118879	2015-06-12	false
7199933130570745587	2005-04-13	false
18103378254596719331	2014-09-15	false
10 rows		3 columns

Development

DuckDB Repositories

Several components of DuckDB are maintained in separate repositories.

Main repositories

- `duckdb`: core DuckDB project
- `duckdb-wasm`: WebAssembly version of DuckDB
- `duckdb-web`: documentation and blog

Clients

- `duckdb-java`: Java (JDBC) client
- `duckdb-node`: Node.js client
- `duckdb-r`: R client
- `duckdb-rs`: Rust client
- `duckdb-swift`: Swift client
- `go-duckdb`: Go client

Connectors

- `dbt-duckdb`: dbt
- `duckdb_mysql`: MySQL connector
- `postgres_scanner`: PostgresSQL connector
- `sqlite_scanner`: SQLite connector

Extensions

Extension repositories are linked in the Official Extensions page.

Testing

Overview

How is DuckDB Tested?

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.
- We use a fuzzer, which automatically reports of issues found through fuzzing DuckDB.

sqllogictest Introduction

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. The 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.

Writing Tests

Development and Testing

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).

Philosophy

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 Catch Tests. However, in most cases you should write your tests in plain SQL.

Frameworks

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.

Functions for Generating Test Data

DuckDB has built-in functions for generating test data.

`test_all_types` Function

The `test_all_types` table function generates a table whose columns correspond to types (`BOOL`, `TINYINT`, etc.). The table has three rows encoding the minimum value, the maximum value, and the null value for each type.

```
FROM test_all_types();
```

```
┌─────────┬─────────┬──────────┬─────────────┬─────────────────────┬──────────────────────┬───┬─────────┐
│  bool   │ tinyint │ smallint │     int     │       bigint        │       hugeint        │ … │ struct  │
│ boolean │  int8   │  int16   │    int32    │        int64        │       int128         │   │ struct(a integer, … │
├─────────┼─────────┼──────────┼─────────────┼─────────────────────┼──────────────────────┼───┼─────────┤
│ false   │    -128 │   -32768 │ -2147483648 │ -9223372036854775808 │ -170141183460469231… │ … │ {'a': NULL, 'b': N… │
│ true    │     127 │    32767 │  2147483647 │  9223372036854775807 │ 170141183460469231… │ … │ {'a': 42, 'b': 🦆… │
│ NULL    │    NULL │     NULL │        NULL │                NULL │                 NULL │ … │ NULL    │
└─────────┴─────────┴──────────┴─────────────┴─────────────────────┴──────────────────────┴───┴─────────┘
```

```
│ 3 rows
44 columns (11 shown) │
└──────────────────────┘
```

`test_vector_types` Function

The `test_vector_types` table function takes *n* arguments `col1`, ..., `coln` and an optional BOOLEAN argument `all_flat`. The function generates a table with *n* columns `test_vector`, `test_vector2`, ..., `test_vectorn`. In each row, each field contains values conforming to the type of their respective column.

```
FROM test_vector_types(NULL::BIGINT);
```

```
┌─────────────────────┐
│     test_vector     │
│        int64        │
├─────────────────────┤
│ -9223372036854775808 │
│  9223372036854775807 │
│                 NULL │
│         ·           │
└─────────────────────┘
```

```
FROM test_vector_types(NULL::ROW(i INTEGER, j VARCHAR, k DOUBLE), NULL::TIMESTAMP);
```

```
┌──────────────────────────────────────────────────────────────────────────────┬──────────────────────────┐
│                                 test_vector                                  │       test_vector2       │
│                  struct(i integer, j varchar, k double)                      │        timestamp         │
├──────────────────────────────────────────────────────────────────────────────┼──────────────────────────┤
│ {'i': -2147483648, 'j': 🦆🦆🦆🦆🦆, 'k': -1.7976931348623157e+308}            │ 290309-12-22 (BC) 00:00:00 │
│ {'i': 2147483647, 'j': goo\0se, 'k': 1.7976931348623157e+308}                │ 294247-01-10 04:00:54.775806 │
│ {'i': NULL, 'j': NULL, 'k': NULL}                                            │ NULL                     │
│                                          ·                                   │          ·               │
└──────────────────────────────────────────────────────────────────────────────┴──────────────────────────┘
```

`test_vector_types` has an optional argument called `all_flat` of type BOOL. This only affects the internal representation of the vector.

```
FROM test_vector_types(NULL::ROW(i INTEGER, j VARCHAR, k DOUBLE), NULL::TIMESTAMP, all_flat = true);
-- the output is the same as above but with a different internal representation
```

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. To 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. To 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. To 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. To run the tests 200..250:

```
build/debug/test/unittest --start-offset=200 --end-offset=250
```

These are also available in percentages. To 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
```

To run only the tests labeled in the file:

```
build/debug/test/unittest -f test.list
```

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.

> **Deprecated.** DuckDB deprecated the usage of types in the sqllogictest. The DuckDB test runner does not use or need them internally – therefore, only `I` should be used 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;
----
```

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'
```

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 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`
`<signed>`	`tinyint smallint integer bigint hugeint`
`<unsigned>`	`utinyint usmallint uinteger ubigint uhugeint`
`<integral>`	`<signed> <unsigned>`
`<numeric>`	`<integral> float double`
`<alltypes>`	`<numeric> bool interval varchar json`

> 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)`.

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`.

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);
```

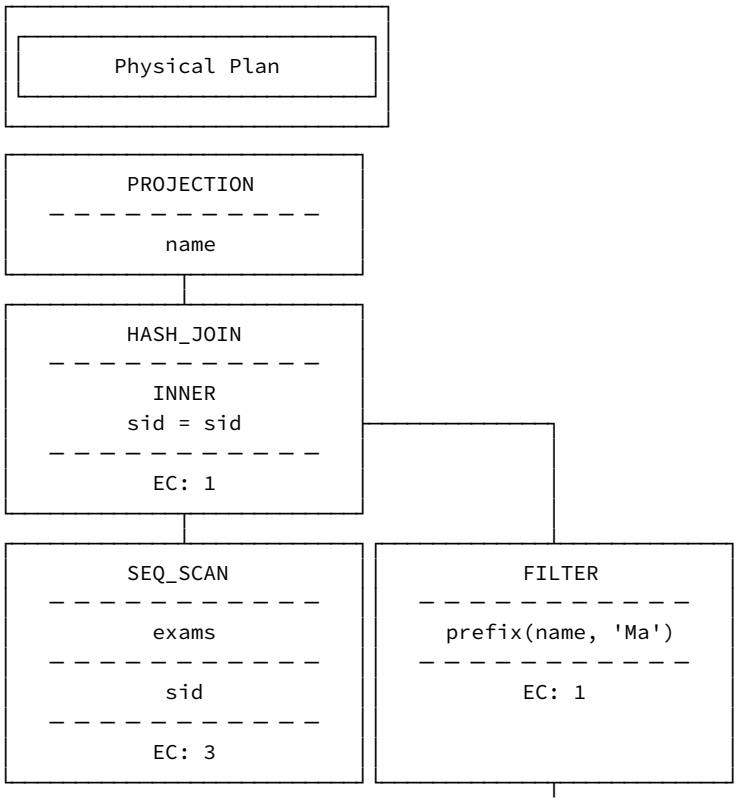
`EXPLAIN` Statement

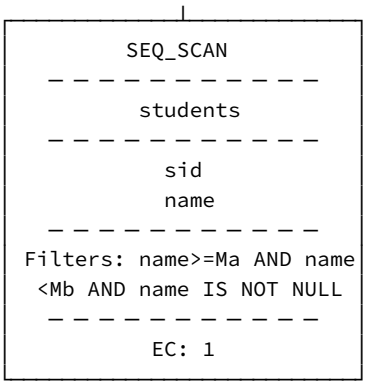
The first step to profiling a database engine is figuring out what execution plan the engine is using. The `EXPLAIN` statement allows you to peek into the query plan and see what is going on under the hood.

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 INTEGER);
CREATE TABLE exams (eid INTEGER, subject VARCHAR, sid INTEGER);
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%';
```





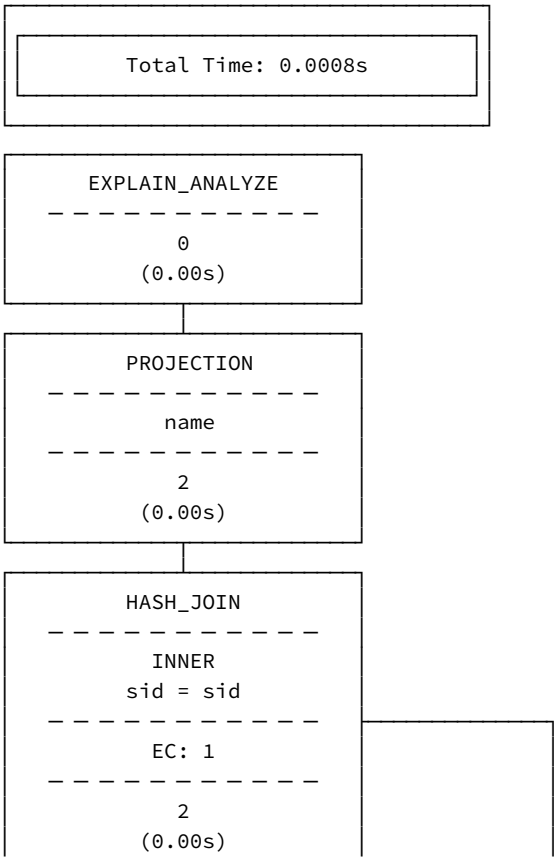
Note that the query is not actually executed – therefore, we can only see the estimated cardinality (EC) for each operator, which is calculated by using the statistics of the base tables and applying heuristics for each operator.

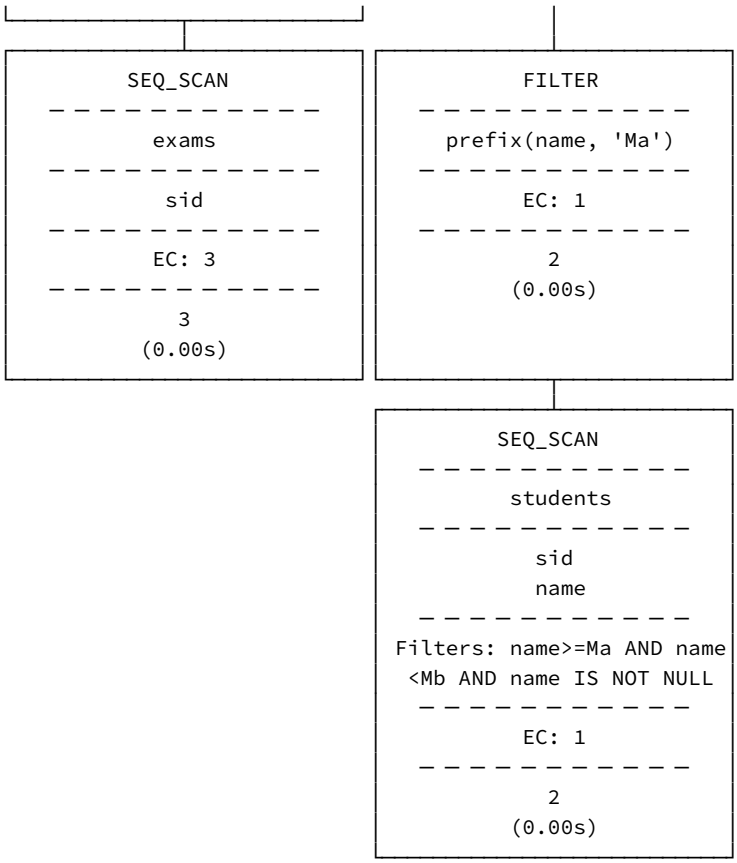
Run-Time Profiling with the `EXPLAIN ANALYZE` Statement

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';
```

> 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 -m duckdb.query_graph /path/to/file.json
```

Notation in Query Plans

In query plans, the hash join operators adhere to the following convention: the *probe side* of the join is the left operand, while the *build side* is the right operand.

Join operators in the query plan show the join type used:

- Inner joins are denoted as `INNER`.
- Left outer joins and right outer joins are denoted as `LEFT` and `RIGHT`, respectively.
- Full outer joins are denoted as `FULL`.

Release Calendar

DuckDB follows semantic versioning. Patch versions only ship bugfixes, while minor versions also introduce new features. The first major version, v1.0, is expected to be released in summer 2024.

Upcoming Releases

The planned dates of upcoming DuckDB releases are shown below. **Please note that these dates are tentative** and DuckDB maintainers may decide to push back release dates to ensure the stability and quality of releases.

Past Releases

In the following, we list DuckDB's past releases, including their codename (where applicable). Prior to version 0.4.0, all releases, including patch versions, received a codename. Since version 0.4.0, only major and minor versions get a codename.

Date	Version	Codename	Named after
2024-05-22	0.10.3	–	–
2024-04-17	0.10.2	–	–
2024-03-18	0.10.1	–	–
2024-02-13	0.10.0	Fusca	Velvet scoter *(Melanitta fusca)*
2023-11-14	0.9.2	–	–
2023-10-11	0.9.1	–	–
2023-09-26	0.9.0	Undulata	Yellow-billed duck *(Anas undulata)*
2023-06-13	0.8.1	–	–
2023-05-17	0.8.0	Fulvigula	Mottled duck *(Anas fulvigula)*
2023-02-27	0.7.1	–	–
2023-02-13	0.7.0	Labradorius	Labrador duck *(Camptorhynchus labradorius)*
2022-12-06	0.6.1	–	–
2022-11-14	0.6.0	Oxyura	White-headed duck *(Oxyura leucocephala)*
2022-09-19	0.5.1	–	–
2022-09-05	0.5.0	Pulchellus	Green pygmy goose *(Nettapus pulchellus)*

Date	Version	Codename	Named after
2022-06-20	0.4.0	Ferruginea	Andean duck *(Oxyura ferruginea)*
2022-04-11	0.3.3	Sansaniensis	*Chenoanas sansaniensis*
2022-02-07	0.3.2	Gibberifrons	Sunda teal *(Anas gibberifrons)*
2021-11-16	0.3.1	Spectabilis	King eider *(Somateria spectabilis)*
2021-10-06	0.3.0	Gracilis	Grey teal *(Anas gracilis)*
2021-09-06	0.2.9	Platyrhynchos	Mallard *(Anas platyrhynchos)*
2021-08-02	0.2.8	Ceruttii	*Histrionicus ceruttii*
2021-06-14	0.2.7	Mollissima	Common eider *(Somateria mollissima)*
2021-05-08	0.2.6	Jamaicensis	Blue-billed ruddy duck *(Oxyura jamaicensis)*
2021-03-10	0.2.5	Falcata	Falcated duck *(Mareca falcata)*
2021-02-02	0.2.4	Jubata	Australian wood duck *(Chenonetta jubata)*
2020-12-03	0.2.3	Serrator	Red-breasted merganser *(Mergus serrator)*
2020-11-01	0.2.2	Clypeata	Northern shoveler *(Spatula clypeata)*
2020-08-29	0.2.1	–	–
2020-07-23	0.2.0	–	–
2020-06-19	0.1.9	–	–
2020-05-29	0.1.8	–	–
2020-05-04	0.1.7	–	–
2020-04-05	0.1.6	–	–
2020-03-02	0.1.5	–	–
2020-02-03	0.1.3	–	–
2020-01-06	0.1.2	–	–
2019-09-24	0.1.1	–	–
2019-06-27	0.1.0	–	–

Building

Building DuckDB from Source

> 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.

Building Instructions

Prerequisites

DuckDB needs CMake and a C++11-compliant compiler (e.g., GCC, Apple-Clang, MSVC). Additionally, we recommend using the Ninja build system, which automatically parallelizes the build process.

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 openssl-devel
```

Ubuntu and Debian:

```
sudo apt-get update && sudo apt-get install -y git g++ cmake ninja-build libssl-dev
```

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.

On Windows, the DuckDB Python package requires the Microsoft Visual C++ Redistributable package to be built and to run.

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
```

> **Best practice.** It is not advised to directly call CMake, as the Makefile sets certain variables that are crucial to properly building the package.

Building Configuration

Build Types

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 `#ifdef DEBUG` blocks. Due to these, binaries of this build are expected to be slow. Note: the special debug defines are not automatically set for this build.

`relassert`

This build does not trigger the `#ifdef DEBUG` code blocks but it 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. Binaries of this build mode are significantly faster than those of the `debug` mode.

`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, DuckDB's in-house benchmark 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.

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.

Overriding Git Hash and Version

It is possible to override the Git hash and version when building from source using the `OVERRIDE_GIT_DESCRIBE` environment variable. This is useful when building from sources that are not part of a complete Git repository (e.g., an archive file with no information on commit hashes and tags). For example:

```
OVERRIDE_GIT_DESCRIBE=v0.10.0-843-g09ea97d0a9 GEN=ninja make
```

Will result in the following output when running `./build/release/duckdb`:

```
v0.10.1-dev843 09ea97d0a9
...
```

Building Extensions

Extensions can be built from source and installed from the resulting local binary.

Building Extensions using Build Flags

To build using extension flags, set the corresponding `BUILD_[EXTENSION_NAME]` extension flag when running the build, then use the `INSTALL` command.

For example, to install the `httpfs` extension, run the following script:

```
GEN=ninja BUILD_HTTPFS=1 make
```

For release builds:

```
build/release/duckdb -c "INSTALL 'build/release/extension/httpfs/httpfs.duckdb_extension';"
```

For debug builds:

```
build/debug/duckdb -c "INSTALL 'build/debug/extension/httpfs/httpfs.duckdb_extension';"
```

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. Instead, it enables tests for TPC-E through our test suite.

BUILD_FTS

When this flag is set, the `fts` (full text search) extension is built.

BUILD_HTTPFS

When this flag is set, the `httpfs` extension is built.

BUILD_JEMALLOC

When this flag is set, the `jemalloc` 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.

Using a CMake Configuration File

To build using a CMake configuration file, create an extension configuration file named `extension_config.cmake` with e.g., the following content:

```
duckdb_extension_load(autocomplete)
duckdb_extension_load(fts)
duckdb_extension_load(httpfs)
duckdb_extension_load(inet)
duckdb_extension_load(icu)
duckdb_extension_load(json)
duckdb_extension_load(parquet)
```

Build DuckDB as follows:

```
GEN=ninja EXTENSION_CONFIGS="extension_config.cmake" make
```

Then, to install the extensions in one go, run:

```
# for release builds
cd build/release/extension/
# for debug builds
cd build/debug/extension/
# install extensions
for EXTENSION in *; do
    ../duckdb -c "INSTALL '${EXTENSION}/${EXTENSION}.duckdb_extension';"
done
```

Supported Platforms

DuckDB officially supports the following platforms:

Platform name	Description
`linux_amd64`	Linux AMD64
`linux_arm64`	Linux ARM64
`osx_amd64`	macOS 12+ (Intel CPUs)
`osx_arm64`	macOS 12+ (Apple Silicon: M1, M2, M3 CPUs)
`windows_amd64`	Windows 10+ on Intel and AMD CPUs (x86_64)

DuckDB can be built from source for several other platforms such as Windows with ARM64 CPUs (Qualcomm, Snapdragon, etc.) and macOS 11.

For details on free and commercial support, see the support policy blog post.

Troubleshooting

Building the R Package is Slow

By default, R compiles packages using a single thread. To parallelize the compilation, create or edit the `~/.R/Makevars` file, and add a line like the following:

```
MAKEFLAGS = -j8
```

The above will parallelize the compilation using 8 threads. In a Linux system, in order to use all of the machine's threads, one can add the following instead:

```
MAKEFLAGS = -j$(nproc)
```

But note that, the more threads that are used, the higher the RAM consumption. If the system runs out of RAM while compiling, then the R session will crash.

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. This example also contains the flag to parallelize the build:

```
ALL_CXXFLAGS = $(PKG_CXXFLAGS) -fPIC $(SHLIB_CXXFLAGS) $(CXXFLAGS)
MAKEFLAGS = -j$(nproc)
```

When making this change, also consider making the build parallel.

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 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 (if required), then build and install the Python package separately using pip:

```
GEN=ninja BUILD_HTTPFS=1 make
```

If the next line complains about pybind11 being missing, or `--use-pep517` not being supported, make sure you're using a modern version of pip and setuptools. `python3-pip` on your OS may not be modern, so you may need to run `python3 -m pip install pip -U` first.

```
python3 -m pip install tools/pythonpkg --use-pep517 --user
```

Building the httpfs Extension on Linux

Problem: When building the `httpfs` extension on Linux, the build may fail with the following error.

```
CMake Error at /usr/share/cmake-3.22/Modules/FindPackageHandleStandardArgs.cmake:230 (message):
  Could NOT find OpenSSL, try to set the path to OpenSSL root folder in the
  system variable OPENSSL_ROOT_DIR (missing: OPENSSL_CRYPTO_LIBRARY
  OPENSSL_INCLUDE_DIR)
```

Solution: Install the `libssl-dev` library.

```
sudo apt-get install -y libssl-dev
```

Then, build with:

```
GEN=ninja BUILD_HTTPFS=1 make
```

Benchmark Suite

DuckDB has an extensive benchmark suite. When making changes that have potential performance implications, it is important to run these benchmarks to detect potential performance regressions.

Getting Started

To build the benchmark suite, run the following command in the DuckDB repository:

```
BUILD_BENCHMARK=1 BUILD_TPCH=1 make
```

Listing Benchmarks

To list all available benchmarks, run:

```
build/release/benchmark/benchmark_runner --list
```

Running Benchmarks

Running a Single Benchmark

To run a single benchmark, issue the following command:

```
build/release/benchmark/benchmark_runner benchmark/micro/nulls/no_nulls_addition.benchmark
```

The output will be printed to `stdout` in CSV format, in the following format:

```
name    run timing
benchmark/micro/nulls/no_nulls_addition.benchmark   1   0.121234
benchmark/micro/nulls/no_nulls_addition.benchmark   2   0.121702
benchmark/micro/nulls/no_nulls_addition.benchmark   3   0.122948
benchmark/micro/nulls/no_nulls_addition.benchmark   4   0.122534
benchmark/micro/nulls/no_nulls_addition.benchmark   5   0.124102
```

You can also specify an output file using the `--out` flag. This will write only the timings (delimited by newlines) to that file.

```
build/release/benchmark/benchmark_runner benchmark/micro/nulls/no_nulls_addition.benchmark
--out=timings.out
```

The output will contain the following:

```
0.182472
0.185027
0.184163
0.185281
0.182948
```

Running Multiple Benchmark Using a Regular Expression

You can also use a regular expression to specify which benchmarks to run. Be careful of shell expansion of certain regex characters (e.g., `*` will likely be expanded by your shell, hence this requires proper quoting or escaping).

```
build/release/benchmark/benchmark_runner "benchmark/micro/nulls/.*"
```

Running All Benchmarks

Not specifying any argument will run all benchmarks.

```
build/release/benchmark/benchmark_runner
```

Other Options

The `--info` flag gives you some other information about the benchmark.

```
build/release/benchmark/benchmark_runner benchmark/micro/nulls/no_nulls_addition.benchmark --info
```

```
display_name:NULL Addition (no nulls)
group:micro
subgroup:nulls
```

The `--query` flag will print the query that is run by the benchmark.

```
SELECT min(i + 1) FROM integers;
```

The `--profile` flag will output a query tree.

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. DuckDB uses a push-based vectorized model, where `DataChunks` are pushed through the operator tree. For more information, see the talk Push-Based Execution in DuckDB.

Storage

Compatibility

Backward Compatibility

Backward compatibility refers to the ability of a newer DuckDB version to read storage files created by an older DuckDB version. Version 0.10 is the first release of DuckDB that supports backward compatibility in the storage format. DuckDB v0.10 can read and operate on files created by the previous DuckDB version – DuckDB v0.9.

For future DuckDB versions, our goal is to ensure that any DuckDB version released **after** can read files created by previous versions, starting from this release. We want to ensure that the file format is fully backward compatible. This allows you to keep data stored in DuckDB files around and guarantees that you will be able to read the files without having to worry about which version the file was written with or having to convert files between versions.

Forward Compatibility

Forward compatibility refers to the ability of an older DuckDB version to read storage files produced by a newer DuckDB version. DuckDB v0.9 is **partially** forward compatible with DuckDB v0.10. Certain files created by DuckDB v0.10 can be read by DuckDB v0.9.

Forward compatibility is provided on a **best effort** basis. While stability of the storage format is important – there are still many improvements and innovations that we want to make to the storage format in the future. As such, forward compatibility may be (partially) broken on occasion.

How to Move Between Storage Formats

When you update DuckDB and open an old 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 one-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.x, v0.10.x
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
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

Compression

DuckDB uses lightweight compression. Note that compression is only applied to persistent databases and is **not applied to in-memory instances**.

Compression Algorithms

The compression algorithms supported by DuckDB include the following:

- Constant Encoding
- Run-Length Encoding (RLE)
- Bit Packing
- Frame of Reference (FOR)
- Dictionary Encoding
- Fast Static Symbol Table (FSST) – VLDB 2020 paper
- Adaptive Lossless Floating-Point Compression (ALP) – SIGMOD 2024 paper
- Chimp – VLDB 2022 paper
- Patas

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.

Row Groups

DuckDB's storage format stores the data in *row groups,* i.e., horizontal partitions of the data. This concept is equivalent to Parquet's row groups. Several features in DuckDB, including parallelism and compression are based on row groups.

Troubleshooting

Error Message When Opening an Incompatible Database File

When opening a database file that has been written by a different DuckDB version from the one you are using, the following error message may occur:

```
Error: unable to open database "...": Serialization Error: Failed to deserialize: ...
```

The message implies that the database file was created with a newer DuckDB version and uses features that are backward incompatible with the DuckDB version used to read the file.

There are two potential workarounds:

1. Update your DuckDB version to the latest stable version.
2. Open the database with the latest version of DuckDB, export it to a standard format (e.g., Parquet), then import it using to any version of DuckDB. See the `EXPORT/IMPORT DATABASE` statements for details.

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 indexes 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.

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.