DuckLake Documentation

DuckLake version 0.3
Generated on 2025-09-26 at 06:53 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

Specification

Introduction

This page contains the specification for the DuckLake format, version 0.3.

Building Blocks

DuckLake requires two main components:

- **Catalog database:** DuckLake requires a database that supports transactions and primary key constraints as defined by the SQL-92 standard.
- **Data storage:** The DuckLake specification requires a storage component for storing the data in Parquet format.

Catalog Database

DuckLake uses SQL tables and queries to define the catalog information (metadata, statistics, etc.). This specification explains the schema and semantics of these:

- Data Types
- Queries
- Tables

If you are reading this specification for the first time, we recommend starting with the "Queries" page, which introduces the queries used by DuckLake.

Data Storage

DuckLake works uses Parquet files to represent its tables. These files can be stored in object storage (blob storage), block storage or file storage.

Data Types

DuckLake specifies multiple different data types for field values, and also supports nested types. The types of columns are defined in the `column_type` field of the `ducklake_column` table.

Primitive Types

Type	Description
`boolean`	True or false
`int8`	8-bit signed integer
`int16`	16-bit signed integer
`int32`	32-bit signed integer
`int64`	64-bit signed integer
`uint8`	8-bit unsigned integer
`uint16`	16-bit unsigned integer
`uint32`	32-bit unsigned integer
`uint64`	64-bit unsigned integer
`float32`	32-bit IEEE 754 floating-point value
`float64`	64-bit IEEE 754 floating-point value
`decimal(P, S)`	Fixed-point decimal with precision P and scale S
`time`	Time of day, microsecond precision
`timetz`	Time of day, microsecond precision, with time zone
`date`	Calendar date
`timestamp`	Timestamp, microsecond precision
`timestamptz`	Timestamp, microsecond precision, with time zone
`timestamp_s`	Timestamp, second precision
`timestamp_ms`	Timestamp, millisecond precision
`timestamp_ns`	Timestamp, nanosecond precision
`interval`	Time interval in three different granularities: months, days, and milliseconds
`varchar`	Text
`blob`	Binary data
`json`	JSON
`uuid`	Universally unique identifier

Nested Types

DuckLake supports nested types and primitive types. Nested types are defined recursively, i.e., in order to define a column of type `INT[]` two columns are defined. The top-level column is of type `list`, which has a child column of type `int32`.

The following nested types are supported:

Type	Description
`list`	Collection of values with a single child type
`struct`	A tuple of typed values
`map`	A collection of key-value pairs

Geometry Types

DuckLake supports geometry types via the `SPATIAL` extension and the Parquet `geometry` type. The `geometry` type can store different types of spatial representations called geometry primitives, of which DuckLake supports the following:

Geometry primitive	Description
`POINT`	A single location in coordinate space.
`LINESTRING`	A sequence of points connected by straight line segments.
`POLYGON`	A planar surface defined by one exterior boundary and zero or more interior boundaries (holes).
`MULTIPOINT`	A collection of `POINT` geometries.
`MULTILINESTRING`	A collection of `LINESTRING` geometries.
`MULTIPOLYGON`	A collection of `POLYGON` geometries.
`LINESTRING Z`	A `LINESTRING` geometry with an additional Z (elevation) coordinate for each point.
`GEOMETRYCOLLECTION`	A heterogeneous collection of geometry primitives (e.g., points, lines, polygons, etc.).

Queries

This page explains the queries issues to the DuckLake catalog database for reading and writing data.

Reading Data

DuckLake specifies *tables* and *update transactions* to modify them. DuckLake is not a black box: all metadata is stored as SQL tables under the user's control. Of course, they can be queried in whichever way is best for a client. Below we describe a small working example to retrieve table data.

> The information below is to provide transparency to users and to aid developers making their own implementation of DuckLake. The `ducklake` DuckDB extension is running these operations in the background.

Get Current Snapshot

Before anything else we need to find a snapshot ID to be queries. There can be many snapshots in the `ducklake_snapshot` table. A snapshot ID is a continuously increasing number that identifies a snapshot. In most cases, you would query the most recent one like so:

```
SELECT snapshot_id
FROM ducklake_snapshot
WHERE snapshot_id =
    (SELECT max(snapshot_id) FROM ducklake_snapshot);
```

List Schemas

A DuckLake catalog can contain many SQL-style schemas, which each can contain many tables. These are listed in the `ducklake_schema` table. Here's how we get the list of valid schemas for a given snapshot:

```
SELECT schema_id, schema_name
FROM ducklake_schema
WHERE
    <SNAPSHOT_ID> >= begin_snapshot AND
    (<SNAPSHOT_ID> < end_snapshot OR end_snapshot IS NULL);
```

where

- `<SNAPSHOT_ID>{:.language-sql .highlight}` is a `BIGINT` referring to the `snapshot_id` column in the `ducklake_snapshot` table.

List Tables

We can list the tables available in a schema for a specific snapshot using the `ducklake_table` table:

```
SELECT table_id, table_name
FROM ducklake_table
WHERE
    schema_id = <SCHEMA_ID> AND
    <SNAPSHOT_ID> >= begin_snapshot AND
    (<SNAPSHOT_ID> < end_snapshot OR end_snapshot IS NULL);
```

where

- <SCHEMA_ID>{:.language-sql .highlight} is a BIGINT referring to the schema_id column in the ducklake_schema table.
- <SNAPSHOT_ID>{:.language-sql .highlight} is a BIGINT referring to the snapshot_id column in the ducklake_snapshot table.

Show the Structure of a Table

For each given table, we can list the available top-level columns using the ducklake_column table:

```
SELECT column_id, column_name, column_type
FROM ducklake_column
WHERE
    table_id = <TABLE_ID> AND
    parent_column IS NULL AND
    <SNAPSHOT_ID> >= begin_snapshot AND
    (<SNAPSHOT_ID> < end_snapshot OR end_snapshot IS NULL)
ORDER BY column_order;
```

where

- <TABLE_ID>{:.language-sql .highlight} is a BIGINT referring to the table_id column in the ducklake_table table.
- <SNAPSHOT_ID>{:.language-sql .highlight} is a BIGINT referring to the snapshot_id column in the ducklake_snapshot table.

> DuckLake supports nested columns – the filter for parent_column IS NULL only shows the top-level columns.

For the list of supported data types, please refer to the "Data Types" page.

SELECT

Now that we know the table structure we can query actual data from the Parquet files that store table data. We need to join the list of data files with the list of delete files (if any). There can be at most one delete file per file in a single snapshot.

```
SELECT data.path AS data_file_path, del.path AS delete_file_path
FROM ducklake_data_file AS data
LEFT JOIN (
    SELECT *
    FROM ducklake_delete_file
    WHERE
        <SNAPSHOT_ID> >= begin_snapshot AND
        (<SNAPSHOT_ID> < end_snapshot OR end_snapshot IS NULL)
    ) AS del
USING (data_file_id)
WHERE
    data.table_id = <TABLE_ID> AND
    <SNAPSHOT_ID> >= data.begin_snapshot AND
    (<SNAPSHOT_ID> < data.end_snapshot OR data.end_snapshot IS NULL)
ORDER BY file_order;
```

where

- <TABLE_ID>{:.language-sql .highlight} is a BIGINT referring to the table_id column in the ducklake_table table.
- <SNAPSHOT_ID>{:.language-sql .highlight} is a BIGINT referring to the snapshot_id column in the ducklake_snapshot table.

Now we have a list of files. In order to reconstruct actual table rows, we need to read all rows from the data_file_path files and remove the rows labeled as deleted in the delete_file_path.

Not all files have to contain all the columns currently defined in the table, some files may also have columns that existed previously but have been removed.

> DuckLake also supports changing the schema, see schema evolution.

Note on Paths

In DuckLake, paths can be relative to the initially specified data path. Whether path is relative or not is stored in the `ducklake_data_file` and `ducklake_delete_file` entries (`path_is_relative`) to the `data_path` prefix from `ducklake_metadata`.

SELECT with File Pruning

One of the main strengths of Lakehouse formats is the ability to *prune* files that cannot contain data relevant to the query. The `ducklake_file_column_stats` table contains the file-level statistics. We can use the information there to prune the list of files to be read if a filter predicate is given.

We can get a list of all files that are part of a given table like described above. We can then reduce that list to only relevant files by querying the per-file column statistics. For example, for scalar equality we can find the relevant files using the query below:

```
SELECT data_file_id
FROM ducklake_file_column_stats
WHERE
    table_id = <TABLE_ID> AND
    column_id = <COLUMN_ID> AND
    (<SCALAR> >= min_value OR min_value IS NULL) AND
    (<SCALAR> <= max_value OR max_value IS NULL);
```

where

- `<TABLE_ID>`{:.language-sql .highlight} is a `BIGINT` referring to the `table_id` column in the `ducklake_table` table.
- `<COLUMN_ID>`{:.language-sql .highlight} is a `BIGINT` referring to the `column_id` column in the `ducklake_column` table.
- `<SCALAR>`{:.language-sql .highlight} is the scalar comparison value for the pruning.

Of course, other filter predicates like “greater than” will require slightly different filtering here.

> The minimum and maximum values for each column are stored as strings and need to be cast for correct range filters on numeric columns.

Writing Data

Snapshot Creation

Any changes to data stored in DuckLake require the creation of a new snapshot. We need to:

- create a new snapshot in `ducklake_snapshot` and
- log the changes a snapshot made in `ducklake_snapshot_changes`

```
INSERT INTO ducklake_snapshot (
    snapshot_id,
    snapshot_timestamp,
    schema_version,
    next_catalog_id,
    next_file_id
)
VALUES (
    <SNAPSHOT_ID>,
    now(),
    <SCHEMA_VERSION>,
    <NEXT_CATALOG_ID>,
    <NEXT_FILE_ID>
```

```
);

INSERT INTO ducklake_snapshot_changes (
    snapshot_id,
    snapshot_changes,
    author,
    commit_message,
    commit_extra_info
)
VALUES (
    <SNAPSHOT_ID>,
    <CHANGES>,
    <AUTHOR>,
    <COMMIT_MESSAGE>,
    <COMMIT_EXTRA_INFO>
);
```

where

- <SNAPSHOT_ID>{:.language-sql .highlight} is the new snapshot identifier. This should be `max(snapshot_id) + 1`.
- <SCHEMA_VERSION>{:.language-sql .highlight} is the schema version for the new snapshot. If any schema changes are made, this needs to be incremented. Otherwise the previous snapshot's `schema_version` can be re-used.
- <NEXT_CATALOG_ID>{:.language-sql .highlight} gives the next unused identifier for tables, schemas, or views. This only has to be incremented if new catalog entries are created.
- <NEXT_FILE_ID>{:.language-sql .highlight} is the same but for data or delete files.
- <CHANGES>{:.language-sql .highlight} contains a list of changes performed by the snapshot. See the list of possible values in the `ducklake_snapshot_changes` table's documentation.
- <AUTHOR>{:.language-sql .highlight} contains information about the author of the commit (optional).
- <COMMIT_MESSAGE>{:.language-sql .highlight} attaches a commit message to the transaction (optional).
- <COMMIT_EXTRA_INFO>{:.language-sql .highlight} attaches extra information to the transaction (optional).

CREATE SCHEMA

A schema is a collection of tables. In order to create a new schema, we can just insert into the `ducklake_schema` table:

```
INSERT INTO ducklake_schema (
    schema_id,
    schema_uuid,
    begin_snapshot,
    end_snapshot,
    schema_name
)
VALUES (
    <SCHEMA_ID>,
    uuid(),
    <SNAPSHOT_ID>,
    NULL,
    <SCHEMA_NAME>
);
```

where

- <SCHEMA_ID>{:.language-sql .highlight} is the new schema identifier. This should be created by incrementing `next_catalog_id` from the previous snapshot.
- <SNAPSHOT_ID>{:.language-sql .highlight} is the snapshot identifier of the new snapshot as described above.
- <SCHEMA_NAME>{:.language-sql .highlight} is just the name of the new schema.

CREATE TABLE

Creating a table in a schema is very similar to creating a schema. We insert into the `ducklake_table` table:

```
INSERT INTO ducklake_table (
    table_id,
    table_uuid,
    begin_snapshot,
    end_snapshot,
    schema_id,
    table_name
)
VALUES (
    <TABLE_ID>,
    uuid(),
    <SNAPSHOT_ID>,
    NULL,
    <SCHEMA_ID>,
    <TABLE_NAME>
);
```

where

- <TABLE_ID>{:.language-sql .highlight} is the new table identifier. This should be created by further incrementing `next_catalog_id` from the previous snapshot.
- <SNAPSHOT_ID>{:.language-sql .highlight} is the snapshot identifier of the new snapshot as described above.
- <SCHEMA_ID>{:.language-sql .highlight} is a `BIGINT` referring to the `schema_id` column in the `ducklake_schema` table table.
- <TABLE_NAME>{:.language-sql .highlight} is just the name of the new table.

A table needs some columns, we can add columns to the new table by inserting into the `ducklake_column` table table. For each column to be added, we run the following query:

```
INSERT INTO ducklake_column (
    column_id,
    begin_snapshot,
    end_snapshot,
    table_id,
    column_order,
    column_name,
    column_type,
    nulls_allowed
)
VALUES (
    <COLUMN_ID>,
    <SNAPSHOT_ID>,
    NULL,
    <TABLE_ID>,
    <COLUMN_ORDER>,
    <COLUMN_NAME>,
    <COLUMN_TYPE>,
    <NULLS_ALLOWED>
);
```

where

- <COLUMN_ID>{:.language-sql .highlight} is the new column identifier. This ID must be unique *within the table* over its entire life time.
- <SNAPSHOT_ID>{:.language-sql .highlight} is the snapshot identifier of the new snapshot as described above.
- <TABLE_ID>{:.language-sql .highlight} is a `BIGINT` referring to the `table_id` column in the `ducklake_table` table.
- <COLUMN_ORDER>{:.language-sql .highlight} is a number that defines where the column is placed in an ordered list of columns.
- <COLUMN_NAME>{:.language-sql .highlight} is just the name of the column.
- <COLUMN_TYPE>{:.language-sql .highlight} is the data type of the column. See the "Data Types" page for details.
- <NULLS_ALLOWED>{:.language-sql .highlight} is a boolean that defines if `NULL` values can be stored in the column. Typically set to `true`.

> We skipped some complexity in this example around default values and nested types and just left those fields as NULL. See the table schema definition for additional details.

DROP TABLE

Dropping a table in DuckLake requires an update in the `end_snapshot` field in all metadata entries corresponding to the dropped table ID.

```
UPDATE ducklake_table
SET
    end_snapshot = <SNAPSHOT_ID>
WHERE
    table_id = <TABLE_ID> AND
    end_snapshot IS NULL;

UPDATE ducklake_partition_info
SET
    end_snapshot = <SNAPSHOT_ID>
WHERE
    table_id = <TABLE_ID> AND
    end_snapshot IS NULL;

UPDATE ducklake_column
SET
    end_snapshot = <SNAPSHOT_ID>
WHERE
    table_id = <TABLE_ID> AND
    end_snapshot IS NULL;

UPDATE ducklake_column_tag
SET
    end_snapshot = <SNAPSHOT_ID>
WHERE
    table_id = <TABLE_ID> AND
    end_snapshot IS NULL;

UPDATE ducklake_data_file
SET
    end_snapshot = <SNAPSHOT_ID>
WHERE
    table_id = <TABLE_ID> AND
    end_snapshot IS NULL;

UPDATE ducklake_delete_file
SET
    end_snapshot = <SNAPSHOT_ID>
WHERE
    table_id = <TABLE_ID> AND
    end_snapshot IS NULL;

UPDATE ducklake_tag
SET
    end_snapshot = <SNAPSHOT_ID>
WHERE
    object_id  = <TABLE_ID> AND
    end_snapshot IS NULL;
```

where

- <SNAPSHOT_ID>{:.language-sql .highlight} is the snapshot identifier of the new snapshot as described above.
- <TABLE_ID>{:.language-sql .highlight} is the identifier of the table that will be dropped.

DROP SCHEMA

Dropping a schema in ducklake requires updating the `end_snapshot` in the `ducklake_schema` table.

```
UPDATE ducklake_schema
SET
    end_snapshot = <SNAPSHOT_ID>
WHERE
    schema_id = <SCHEMA_ID> AND
    end_snapshot IS NULL;
```

where

- <SNAPSHOT_ID>{:.language-sql .highlight} is the snapshot identifier of the new snapshot as described above.
- <SCHEMA_ID>{:.language-sql .highlight} is the identifier of the schema that will be dropped.

> DROP SCHEMA is only allowed on empty schemas. Ensure that all tables within the schema are dropped beforehand.

INSERT

Inserting data into a DuckLake table consists of two main steps: first, we need to write a Parquet file containing the actual row data to storage, and second, we need to register that file in the metadata tables and update global statistics. Let's assume the file has already been written.

```
INSERT INTO ducklake_data_file (
    data_file_id,
    table_id,
    begin_snapshot,
    end_snapshot,
    path,
    path_is_relative,
    file_format,
    record_count,
    file_size_bytes,
    footer_size,
    row_id_start
)
VALUES (
    <DATA_FILE_ID>,
    <TABLE_ID>,
    <SNAPSHOT_ID>,
    NULL,
    <PATH>,
    true,
    'parquet',
    <RECORD_COUNT>,
    <FILE_SIZE_BYTES>,
    <FOOTER_SIZE>,
    <ROW_ID_START>
);
```

where

- <DATA_FILE_ID>{:.language-sql .highlight} is the new data file identifier. This ID must be unique *within the table* over its entire life time.
- <TABLE_ID>{:.language-sql .highlight} is a `BIGINT` referring to the `table_id` column in the `ducklake_table` table.
- <SNAPSHOT_ID>{:.language-sql .highlight} is the snapshot identifier of the new snapshot as described above.
- <PATH>{:.language-sql .highlight} is the file name relative to the DuckLake data path from the top-level metadata.
- <RECORD_COUNT>{:.language-sql .highlight} is the number of rows in the file.
- <FILE_SIZE_BYTES>{:.language-sql .highlight} is the file size.

- <FOOTER_SIZE>{:.language-sql .highlight} is the position of the Parquet footer. This helps with efficiently reading the file.
- <ROW_ID_START>{:.language-sql .highlight} is the first logical row ID from the file. This number can be read from the ducklake_table_stats table via column `next_row_id`.

We have omitted some complexity around relative paths, encrypted files, partitioning and partial files in this example. Refer to the ducklake_data_file table documentation for details.

> DuckLake also supports changing the schema, see schema evolution.

We also have to update some statistics in the ducklake_table_stats table and ducklake_table_column_stats table' tables.

```
UPDATE ducklake_table_stats
SET
    record_count = record_count + <RECORD_COUNT>,
    next_row_id = next_row_id + <RECORD_COUNT>,
    file_size_bytes = file_size_bytes + <FILE_SIZE_BYTES>
WHERE table_id = <TABLE_ID>;

UPDATE ducklake_table_column_stats
SET
    contains_null = contains_null OR <NULL_COUNT> > 0,
    contains_nan = contains_nan,
    min_value = min(min_value, <MIN_VALUE>),
    max_value = max(max_value, <MAX_VALUE>)
WHERE
    table_id = <TABLE_ID> AND
    column_id = <COLUMN_ID>;

INSERT INTO ducklake_file_column_stats (
    data_file_id,
    table_id,
    column_id,
    value_count,
    null_count,
    min_value,
    max_value,
    contains_nan
)
VALUES (
    <DATA_FILE_ID>,
    <TABLE_ID>,
    <COLUMN_ID>,
    <RECORD_COUNT>,
    <NULL_COUNT>,
    <MIN_VALUE>,
    <MAX_VALUE>,
    <CONTAINS_NAN>;
);
```

where

- <TABLE_ID>{:.language-sql .highlight} is a `BIGINT` referring to the `table_id` column in the ducklake_table table.
- <COLUMN_ID>{:.language-sql .highlight} is a `BIGINT` referring to the `column_id` column in the ducklake_column table.
- <DATA_FILE_ID>{:.language-sql .highlight} is a `BIGINT` referring to the `data_file_id` column in the ducklake_data_file table.
- <RECORD_COUNT>{:.language-sql .highlight} is the number of values (including `NULL` and `NaN` values) in the file column.
- <NULL_COUNT>{:.language-sql .highlight} is the number of `NULL` values in the file column.
- <MIN_VALUE>{:.language-sql .highlight} is the *minimum* value in the file column as a string.
- <MAX_VALUE>{:.language-sql .highlight} is the *maximum* value in the file column as a string.
- <FILE_SIZE_BYTES>{:.language-sql .highlight} is the size of the new Parquet file.
- <CONTAINS_NAN>{:.language-sql .highlight} is a flag whether the column contains any `NaN` values. This is only relevant for floating-point types.

> This example assumes there are already rows in the table. If there are none, we need to use `INSERT` instead here. We also skipped the `column_size_bytes` column here, it can safely be set to `NULL`.

DELETE

Deleting data from a DuckLake table consists of two main steps: first, we need to write a Parquet delete file containing the row index to be deleted to storage, and second, we need to register that delete file in the metadata tables. Let's assume the file has already been written.

```
INSERT INTO ducklake_delete_file (
    delete_file_id,
    table_id,
    begin_snapshot,
    end_snapshot,
    data_file_id,
    path,
    path_is_relative,
    format,
    delete_count,
    file_size_bytes,
    footer_size
)
VALUES (
    <DELETE_FILE_ID>,
    <TABLE_ID>,
    <SNAPSHOT_ID>,
    NULL,
    <DATA_FILE_ID>,
    <PATH>,
    true,
    'parquet',
    <DELETE_COUNT>,
    <FILE_SIZE_BYTES>,
    <FOOTER_SIZE>
);
```

where

- <DELETE_FILE_ID>{:.language-sql .highlight} is the identifier for the new delete file.
- <TABLE_ID>{:.language-sql .highlight} is a `BIGINT` referring to the `table_id` column in the `ducklake_table` table.
- <SNAPSHOT_ID>{:.language-sql .highlight} is the snapshot identifier of the new snapshot as described above.
- <DATA_FILE_ID>{:.language-sql .highlight} is the identifier of the data file from which the rows are to be deleted.
- <PATH>{:.language-sql .highlight} is the file name relative to the DuckLake data path from the top-level metadata.
- <DELETE_COUNT>{:.language-sql .highlight} is the number of deletion records in the file.
- <FILE_SIZE_BYTES>{:.language-sql .highlight} is the file size.
- <FOOTER_SIZE>{:.language-sql .highlight} is the position of the Parquet footer. This helps with efficiently reading the file.

> We have omitted some complexity around relative paths and encrypted files in this example. Refer to the `ducklake_delete_file` table documentation for details.

> Please note that `DELETE` operations also do not require updates to table statistics, as the statistics are maintained as upper bounds, and deletions do not violate these bounds.

UPDATE

In DuckLake, `UPDATE` operations are internally implemented as a combination of a `DELETE` followed by an `INSERT`. Specifically, the outdated row is marked for deletion, and the updated version of that row is inserted. As a result, the changes to the metadata tables are equivalent to performing a `DELETE` and an `INSERT` operation sequentially within the same transaction.

Tables

Tables

DuckLake 0.3 uses 22 tables to store metadata and to stage data fragments for data inlining. Below we describe all those tables and their semantics.

The following figure shows the most important tables defined by the DuckLake schema:

DuckLake schema{: .lightmode-img } DuckLake schema{: .darkmode-img }

Snapshots

- `ducklake_snapshot`
- `ducklake_snapshot_changes`

DuckLake Schema

- `ducklake_schema`
- `ducklake_table`
- `ducklake_view`
- `ducklake_column`

Data Files and Tables

- `ducklake_data_file`
- `ducklake_delete_file`
- `ducklake_files_scheduled_for_deletion`
- `ducklake_inlined_data_tables`

Data File Mapping

- `ducklake_column_mapping`
- `ducklake_name_mapping`

Statistics

DuckLake supports statistics on the table, column and file level.

- `ducklake_table_stats`
- `ducklake_table_column_stats`
- `ducklake_file_column_stats`

Partitioning Information

DuckLake supports defining explicit partitioning.

- ducklake_partition_info
- ducklake_partition_column
- ducklake_file_partition_value

Auxiliary Tables

- ducklake_metadata
- ducklake_tag
- ducklake_column_tag
- ducklake_schema_versions

Full Schema Creation Script

Below is the full SQL script to create a DuckLake metadata database:

```
CREATE TABLE ducklake_metadata (key VARCHAR NOT NULL, value VARCHAR NOT NULL, scope VARCHAR, scope_id
BIGINT);
CREATE TABLE ducklake_snapshot (snapshot_id BIGINT PRIMARY KEY, snapshot_time TIMESTAMPTZ, schema_version
BIGINT, next_catalog_id BIGINT, next_file_id BIGINT);
CREATE TABLE ducklake_snapshot_changes (snapshot_id BIGINT PRIMARY KEY, changes_made VARCHAR, author
VARCHAR, commit_message VARCHAR, commit_extra_info VARCHAR);
CREATE TABLE ducklake_schema (schema_id BIGINT PRIMARY KEY, schema_uuid UUID, begin_snapshot BIGINT, end_
snapshot BIGINT, schema_name VARCHAR, path VARCHAR, path_is_relative BOOLEAN);
CREATE TABLE ducklake_table (table_id BIGINT, table_uuid UUID, begin_snapshot BIGINT, end_snapshot BIGINT,
schema_id BIGINT, table_name VARCHAR, path VARCHAR, path_is_relative BOOLEAN);
CREATE TABLE ducklake_view (view_id BIGINT, view_uuid UUID, begin_snapshot BIGINT, end_snapshot BIGINT,
schema_id BIGINT, view_name VARCHAR, dialect VARCHAR, sql VARCHAR, column_aliases VARCHAR);
CREATE TABLE ducklake_tag (object_id BIGINT, begin_snapshot BIGINT, end_snapshot BIGINT, key VARCHAR, value
VARCHAR);
CREATE TABLE ducklake_column_tag (table_id BIGINT, column_id BIGINT, begin_snapshot BIGINT, end_snapshot
BIGINT, key VARCHAR, value VARCHAR);
CREATE TABLE ducklake_data_file (data_file_id BIGINT PRIMARY KEY, table_id BIGINT, begin_snapshot BIGINT,
end_snapshot BIGINT, file_order BIGINT, path VARCHAR, path_is_relative BOOLEAN, file_format VARCHAR,
record_count BIGINT, file_size_bytes BIGINT, footer_size BIGINT, row_id_start BIGINT, partition_id BIGINT,
encryption_key VARCHAR, partial_file_info VARCHAR, mapping_id BIGINT);
CREATE TABLE ducklake_file_column_stats (data_file_id BIGINT, table_id BIGINT, column_id BIGINT, column_
size_bytes BIGINT, value_count BIGINT, null_count BIGINT, min_value VARCHAR, max_value VARCHAR, contains_
nan BOOLEAN, extra_stats VARCHAR);
CREATE TABLE ducklake_delete_file (delete_file_id BIGINT PRIMARY KEY, table_id BIGINT, begin_snapshot
BIGINT, end_snapshot BIGINT, data_file_id BIGINT, path VARCHAR, path_is_relative BOOLEAN, format VARCHAR,
delete_count BIGINT, file_size_bytes BIGINT, footer_size BIGINT, encryption_key VARCHAR);
CREATE TABLE ducklake_column (column_id BIGINT, begin_snapshot BIGINT, end_snapshot BIGINT, table_id
BIGINT, column_order BIGINT, column_name VARCHAR, column_type VARCHAR, initial_default VARCHAR, default_
value VARCHAR, nulls_allowed BOOLEAN, parent_column BIGINT);
CREATE TABLE ducklake_table_stats (table_id BIGINT, record_count BIGINT, next_row_id BIGINT, file_size_
bytes BIGINT);
CREATE TABLE ducklake_table_column_stats (table_id BIGINT, column_id BIGINT, contains_null BOOLEAN,
contains_nan BOOLEAN, min_value VARCHAR, max_value VARCHAR, extra_stats VARCHAR);
CREATE TABLE ducklake_partition_info (partition_id BIGINT, table_id BIGINT, begin_snapshot BIGINT, end_
snapshot BIGINT);
CREATE TABLE ducklake_partition_column (partition_id BIGINT, table_id BIGINT, partition_key_index BIGINT,
column_id BIGINT, transform VARCHAR);
CREATE TABLE ducklake_file_partition_value (data_file_id BIGINT, table_id BIGINT, partition_key_index
BIGINT, partition_value VARCHAR);
```

```
CREATE TABLE ducklake_files_scheduled_for_deletion (data_file_id BIGINT, path VARCHAR, path_is_relative
BOOLEAN, schedule_start TIMESTAMPTZ);
CREATE TABLE ducklake_inlined_data_tables (table_id BIGINT, table_name VARCHAR, schema_version BIGINT);
CREATE TABLE ducklake_column_mapping (mapping_id BIGINT, table_id BIGINT, type VARCHAR);
CREATE TABLE ducklake_name_mapping (mapping_id BIGINT, column_id BIGINT, source_name VARCHAR, target_field_
id BIGINT, parent_column BIGINT, is_partition BOOLEAN);
CREATE TABLE ducklake_schema_versions (begin_snapshot BIGINT, schema_version BIGINT);
```

ducklake_column

This table describes the columns that are part of a table, including their types, default values etc.

Column name	Column type
column_id	BIGINT
begin_snapshot	BIGINT
end_snapshot	BIGINT
table_id	BIGINT
column_order	BIGINT
column_name	VARCHAR
column_type	VARCHAR
initial_default	VARCHAR
default_value	VARCHAR
nulls_allowed	BOOLEAN
parent_column	BIGINT

- `column_id` is the numeric identifier of the column. If the parquet file includes a field identifier, it corresponds to the file's `field_id`. This identifier should remain consistent throughout all versions of the column, until it's dropped.
- `begin_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. This version of the column exists *starting with* this snapshot id.
- `end_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. This version of the column exists *up to but not including* this snapshot id. If `end_snapshot` is `NULL`, this version of the column is currently valid.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `column_order` is a number that defines the position of the column in the list of columns. it needs to be unique within a snapshot but does not have to be strictly monotonic (gaps are ok).
- `column_name` is the name of this version of the column, e.g., `my_column`.
- `column_type` is the type of this version of the column as defined in the list of data types.
- `initial_default` is the *initial* default value as the column is being created, e.g., in `ALTER TABLE`, encoded as a string. Can be `NULL`.
- `default_value` is the *operational* default value as data is being inserted and updated, e.g., in `INSERT`, encoded as a string. Can be `NULL`.
- `nulls_allowed` defines whether `NULL` values are allowed in this version of the column. Note that default values have to be set if this is set to `false`.
- `parent_column` is the `column_id` of the parent column. This is `NULL` for top-level and non-nested columns. For example, for `STRUCT` types, this would refer to the "parent" struct column.

> Every `ALTER` of the column creates a new version of the column, which will use the same `column_id`.

ducklake_column_mapping

Mappings contain the information used to map parquet fields to column ids in the absence of `field-id`s in the Parquet file.

Column name	Column type
`mapping_id`	`BIGINT`
`table_id`	`BIGINT`
`type`	`VARCHAR`

- `mapping_id` is the numeric identifier of the mapping. `mapping_id` is incremented from `next_catalog_id` in the `ducklake_snapshot` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `type` defines what method is used to perform the mapping.

The valid `type` values:

`type`	Description
`map_by_name`	Map the columns based on the names in the parquet file

ducklake_name_mapping

This table contains the information used to map a name to a `column_id` for a given `mapping_id` with the `map_by_name` type.

Column name	Column type
`mapping_id`	`BIGINT`
`column_id`	`BIGINT`
`source_name`	`VARCHAR`
`target_field_id`	`BIGINT`
`parent_column`	`BIGINT`

- `mapping_id` refers to a `mapping_id` from the `ducklake_column_mapping` table.
- `column_id` refers to a `column_id` from the `ducklake_column` table.
- `source_name` refers to the name of the field this mapping applies to.
- `target_field_id` refers to the `field-id` that a field with the `source_name` is mapped to.
- `parent_column` is the `column_id` of the parent column. This is `NULL` for top-level and non-nested columns. For example, for STRUCT types, this would refer to the ”parent” struct column.

ducklake_column_tag

Columns can also have tags, those are defined in this table.

Column name	Column type
`table_id`	`BIGINT`

Column name	Column type
column_id	BIGINT
begin_snapshot	BIGINT
end_snapshot	BIGINT
key	VARCHAR
value	VARCHAR

- table_id refers to a table_id from the ducklake_table table.
- column_id refers to a column_id from the ducklake_column table.
- begin_snapshot refers to a snapshot_id from the ducklake_snapshot table. The tag is valid *starting with* this snapshot id.
- end_snapshot refers to a snapshot_id from the ducklake_snapshot table. The tag is valid *up to but not including* this snapshot id. If end_snapshot is NULL, the tag is currently valid.
- key is an arbitrary key string. The key can't be NULL.
- value is the arbitrary value string.

ducklake_data_file

Data files contain the actual row data.

Column name	Column type	
data_file_id	BIGINT	Primary Key
table_id	BIGINT	
begin_snapshot	BIGINT	
end_snapshot	BIGINT	
file_order	BIGINT	
path	VARCHAR	
path_is_relative	BOOLEAN	
file_format	VARCHAR	
record_count	BIGINT	
file_size_bytes	BIGINT	
footer_size	BIGINT	
row_id_start	BIGINT	
partition_id	BIGINT	
encryption_key	VARCHAR	
partial_file_info	VARCHAR	
mapping_id	BIGINT	

- data_file_id is the numeric identifier of the file. It is a primary key. data_file_id is incremented from next_file_id in the ducklake_snapshot table.
- table_id refers to a table_id from the ducklake_table table.
- begin_snapshot refers to a snapshot_id from the ducklake_snapshot table. The file is part of the table *starting with* this snapshot id.

- `end_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The file is part of the table *up to but not including* this snapshot id. If `end_snapshot` is `NULL`, the file is currently part of the table.
- `file_order` is a number that defines the vertical position of the file in the table. it needs to be unique within a snapshot but does not have to be strictly monotonic (holes are ok).
- `path` is the file path of the data file, e.g., `my_file.parquet` for a relative path.
- `path_is_relative` whether the `path` is relative to the `path` of the table (true) or an absolute path (false).
- `file_format` is the storage format of the file. Currently, only `parquet` is allowed.
- `record_count` is the number of records (row) in the file.
- `file_size_bytes` is the size of the file in Bytes.
- `footer_size` is the size of the file metadata footer, in the case of Parquet the Thrift data. This is an optimization that allows for faster reading of the file.
- `row_id_start` is the first logical row id in the file. (Every row has a unique row-id that is maintained.)
- `partition_id` refers to a `partition_id` from the `ducklake_partition_info` table.
- `encryption_key` contains the encryption for the file if encryption is enabled.
- `partial_file_info` is used when snapshots refer to parts of a file.
- `mapping_id` refers to a `mapping_id` from the `ducklake_column_mapping` table.

ducklake_delete_file

Delete files contains the row ids of rows that are deleted. Each data file will have its own delete file if any deletes are present for this data file.

Column name	Column type	
`delete_file_id`	`BIGINT`	Primary Key
`table_id`	`BIGINT`	
`begin_snapshot`	`BIGINT`	
`end_snapshot`	`BIGINT`	
`data_file_id`	`BIGINT`	
`path`	`VARCHAR`	
`path_is_relative`	`BOOLEAN`	
`format`	`VARCHAR`	
`delete_count`	`BIGINT`	
`file_size_bytes`	`BIGINT`	
`footer_size`	`BIGINT`	
`encryption_key`	`VARCHAR`	

- `delete_file_id` is the numeric identifier of the delete file. It is a primary key. `delete_file_id` is incremented from `next_file_id` in the `ducklake_snapshot` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `begin_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The delete file is part of the table *starting with* this snapshot id.
- `end_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The delete file is part of the table *up to but not including* this snapshot id. If `end_snapshot` is `NULL`, the delete file is currently part of the table.
- `data_file_id` refers to a `data_file_id` from the `ducklake_data_file` table.
- `path` is the file name of the delete file, e.g., `my_file-deletes.parquet` for a relative path.
- `path_is_relative` whether the `path` is relative to the `path` of the table (true) or an absolute path (false).
- `format` is the storage format of the delete file. Currently, only `parquet` is allowed.

- `delete_count` is the number of deletion records in the file.
- `file_size_bytes` is the size of the file in Bytes.
- `footer_size` is the size of the file metadata footer, in the case of Parquet the Thrift data. This is an optimization that allows for faster reading of the file.
- `encryption_key` contains the encryption for the file if encryption is enabled.

ducklake_file_column_stats

This table contains column-level statistics for a single data file.

Column name	Column type
`data_file_id`	`BIGINT`
`table_id`	`BIGINT`
`column_id`	`BIGINT`
`column_size_bytes`	`BIGINT`
`value_count`	`BIGINT`
`null_count`	`BIGINT`
`min_value`	`VARCHAR`
`max_value`	`VARCHAR`
`contains_nan`	`BOOLEAN`
`extra_stats`	`VARCHAR`

- `data_file_id` refers to a `data_file_id` from the `ducklake_data_file` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `column_id` refers to a `column_id` from the `ducklake_column` table.
- `column_size_bytes` is the byte size of the column.
- `value_count` is the number of values in the column. This does not have to correspond to the number of records in the file for nested types.
- `null_count` is the number of values in the column that are `NULL`.
- `min_value` contains the minimum value for the column, encoded as a string. This does not have to be exact but has to be a lower bound. The value has to be cast to the actual type for accurate comparison, e.g., on integer types.
- `max_value` contains the maximum value for the column, encoded as a string. This does not have to be exact but has to be an upper bound. The value has to be cast to the actual type for accurate comparison, e.g., on integer types.
- `contains_nan` is a flag whether the column contains any `NaN` values. This is only relevant for floating-point types.
- `extra_stats` contains different statistics from specific types such as Geometry types.

ducklake_file_partition_value

This table defines which data file belongs to which partition.

Column name	Column type
`data_file_id`	`BIGINT`
`table_id`	`BIGINT`
`partition_key_index`	`BIGINT`

Column name	Column type
`partition_value`	`VARCHAR`

- `data_file_id` refers to a `data_file_id` from the `ducklake_data_file` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `partition_key_index` refers to a `partition_key_index` from the `ducklake_partition_column` table.
- `partition_value` is the value that all the rows in the data file have, encoded as a string.

ducklake_files_scheduled_for_deletion

Files that are no longer part of any snapshot are scheduled for deletion

Column name	Column type
`data_file_id`	`BIGINT`
`path`	`VARCHAR`
`path_is_relative`	`BOOLEAN`
`schedule_start`	`TIMESTAMP`

- `data_file_id` refers to a `data_file_id` from the `ducklake_data_file` table.
- `path` is the file name of the file, e.g., `my_file.parquet`. The file name is either relative to the `data_path` value in `ducklake_metadata` or absolute. If relative, the `path_is_relative` field is set to `true`.
- `path_is_relative` defines whether the path is absolute or relative, see above.
- `schedule_start` is a time stamp of when this file was scheduled for deletion.

ducklake_inlined_data_tables

This table links DuckLake snapshots with inlined data tables.

Column name	Column type
`table_id`	`BIGINT`
`table_name`	`VARCHAR`
`schema_version`	`BIGINT`

- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `table_name` is a string that names the data table for inlined data.
- `schema_version` refers to a schema version in the `ducklake_snapshot` table.

ducklake_metadata

The `ducklake_metadata` table contains key/value pairs with information about the specific setup of the DuckLake catalog.

Column name	Column type	
`key`	`VARCHAR`	Not `NULL`
`value`	`VARCHAR`	Not `NULL`
`scope`	`VARCHAR`	
`scope_id`	`BIGINT`	

- `key` is an arbitrary key string. See below for a list of pre-defined keys. The key can't be `NULL`.
- `value` is the arbitrary value string.
- `scope` defines the scope of the setting.
- `scope_id` is the id of the item that the setting is scoped to (see the table below) or `NULL` for the Global scope.

Scope	`scope`	Description
Global	`NULL`	The scope of the setting is global for the entire catalog.
Schema	`schema`	The setting is scoped to the `schema_id` referenced by `scope_id`.
Table	`table`	The setting is scoped to the `table_id` referenced by `scope_id`.

Currently, the following values for `key` are specified:

Name	Description	Notes	Scope(s)
`version`	DuckLake format version.		Global
`created_by`	Tool used to write the DuckLake.		Global
`table`	A string that identifies which program wrote the schema, e.g., `DuckDB v1.3.2`.		Global
`data_path`	Path to data files, e.g., `s3://mybucket/myprefix/`.	Has to end in `/`	Global
`encrypted`	Whether or not to encrypt Parquet files written to the data path.	`'true'` or `'false'`	Global
`data_inlining_row_limit`	Maximum amount of rows to inline in a single insert.		Global, Schema, Table
`target_file_size`	The target data file size for insertion and compaction operations.		Global, Schema, Table
`parquet_row_group_size_bytes`	Number of bytes per row group in Parquet files.		Global, Schema, Table
`parquet_row_group_size`	Number of rows per row group in Parquet files.		Global, Schema, Table

Name	Description	Notes	Scope(s)
`parquet_compression`	Compression algorithm for Parquet files, e.g., `zstd`.	`uncompressed`, `snappy`, `gzip`, `zstd`, `brotli`, `lz4`, `lz4_raw`	Global, Schema, Table
`parquet_compression_level`	Compression level for Parquet files.		Global, Schema, Table
`parquet_version`	Parquet format version.	`1` or `2`	Global, Schema, Table
`hive_file_pattern`	If partitioned data should be written in a hive-like folder structure.	`'true'` or `'false'`	Global
`require_commit_message`	If an explicit commit message is required for a snapshot commit.	`'true'` or `'false'`	Global
`rewrite_delete_threshold`	Minimum amount of data (0-1) that must be removed from a file before a rewrite is warranted.	Value between `0` and `1`	Global
`delete_older_than`	How old unused files must be to be removed by the `ducklake_delete_orphaned_files` and `ducklake_cleanup_old_files` functions.	Duration string (e.g., `7d`, `24h`)	Global
`expire_older_than`	How old snapshots must be, by default, to be expired by `ducklake_expire_snapshots`.	Duration string (e.g., `30d`)	Global
`compaction_schema`	Pre-defined schema used as a default value for compaction functions.	Used by `ducklake_flush_inlined_data`, `ducklake_merge_adjacent_files`, `ducklake_rewrite_data_files`, etc.	Global
`compaction_table`	Pre-defined table used as a default value for compaction functions.	Used by `ducklake_flush_inlined_data`, `ducklake_merge_adjacent_files`, `ducklake_rewrite_data_files`, etc.	Global
`per_thread_output`	Whether to create separate output files per thread during parallel insertion.	`'true'` or `'false'`	Global

ducklake_partition_column

Partitions can refer to one or more columns, possibly with transformations such as hashing or bucketing.

Column name	Column type
`partition_id`	`BIGINT`
`table_id`	`BIGINT`
`partition_key_index`	`BIGINT`

Column name	Column type
`column_id`	`BIGINT`
`transform`	`VARCHAR`

- `partition_id` refers to a `partition_id` from the `ducklake_partition_info` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `partition_key_index` defines where in the partition key the column is. For example, in a partitioning by (a, b, c) the `partition_key_index` of b would be `1`.
- `column_id` refers to a `column_id` from the `ducklake_column` table.
- `transform` defines the type of a transform that is applied to the column value, e.g., `year`.

The table of supported transforms is as follows.

Transform	Source type(s)	Description	Result type
`identity`	Any	Source value, unmodified	Source type
`year`	`date`, `timestamp`, `timestamptz`, `timestamp_s`, `timestamp_ms`, `timestamp_ns`	Extract a date or timestamp year, as years from 1970	`int64`
`month`	`date`, `timestamp`, `timestamptz`, `timestamp_s`, `timestamp_ms`, `timestamp_ns`	Extract a date or timestamp month, as months from 1970-01-01	`int64`
`day`	`date`, `timestamp`, `timestamptz`, `timestamp_s`, `timestamp_ms`, `timestamp_ns`	Extract a date or timestamp day, as days from 1970-01-01	`int64`
`hour`	`timestamp`, `timestamptz`, `timestamp_s`, `timestamp_ms`, `timestamp_ns`	Extract a timestamp hour, as hours from 1970-01-01 00:00:00	`int64`

ducklake_partition_info

Column name	Column type
`partition_id`	`BIGINT`
`table_id`	`BIGINT`
`begin_snapshot`	`BIGINT`
`end_snapshot`	`BIGINT`

- `partition_id` is a numeric identifier for a partition. `partition_id` is incremented from `next_catalog_id` in the `ducklake_snapshot` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `begin_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The partition is valid *starting with* this snapshot id.
- `end_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The partition is valid *up to but not including* this snapshot id. If `end_snapshot` is `NULL`, the partition is currently valid.

ducklake_schema

This table defines valid schemas.

Column name	Column type	
schema_id	BIGINT	Primary Key
schema_uuid	UUID	
begin_snapshot	BIGINT	
end_snapshot	BIGINT	
schema_name	VARCHAR	
path	VARCHAR	
path_is_relative	BOOLEAN	

- schema_id is the numeric identifier of the schema. schema_id is incremented from next_catalog_id in the ducklake_snapshot table.
- schema_uuid is a UUID that gives a persistent identifier for this schema. The UUID is stored here for compatibility with existing Lakehouse formats.
- begin_snapshot refers to a snapshot_id from the ducklake_snapshot table. The schema exists *starting with* this snapshot id.
- end_snapshot refers to a snapshot_id from the ducklake_snapshot table. The schema exists *up to but not including* this snapshot id. If end_snapshot is NULL, the schema is currently valid.
- schema_name is the name of the schema, e.g., my_schema.
- path is the data_path of the schema.
- path_is_relative whether the path is relative to the data_path of the catalog (true) or an absolute path (false).

ducklake_schema_versions

This table contains the schema versions for a range of snapshots. It is necessary to compact files with different schemas.

Column name	Column type
begin_snapshot	BIGINT
schema_version	BIGINT

- begin_snapshot refers to a snapshot_id in the ducklake_snapshot table.
- schema_version refers to the schema_version of a ducklake_snapshot.

ducklake_snapshot

This table contains the valid snapshots in a DuckLake.

Column name	Column type	
snapshot_id	BIGINT	Primary Key
snapshot_time	TIMESTAMP	
schema_version	BIGINT	
next_catalog_id	BIGINT	
next_file_id	BIGINT	

- `snapshot_id` is the continuously increasing numeric identifier of the snapshot. It is a primary key and is referred to by various other tables below.
- `snapshot_time` is the time stamp at which the snapshot was created.
- `schema_version` is a continuously increasing number that is incremented whenever the schema is changed, e.g., by creating a table. This allows for caching of schema information if only data is changed.
- `next_catalog_id` is a continuously increasing number that describes the next identifier for schemas, tables, views, partitions, and column name mappings. This is only changed if one of those entries is created, i.e., the schema is changing.
- `next_file_id` is a continuously increasing number that contains the next id for a data or deletion file to be added. It is only changed if data is being added or deleted, i.e., not for schema changes.

ducklake_snapshot_changes

This table lists changes that happened in a snapshot for easier conflict detection.

Column name	Column type	Description
`snapshot_id`	`BIGINT`	Primary Key
`changes_made`	`VARCHAR`	List of changes in snapshot
`author`	`VARCHAR`	Author of the snapshot
`commit_message`	`VARCHAR`	Commit message
`commit_extra_info`	`VARCHAR`	Extra information regarding the commit

The `ducklake_snapshot_changes` table contains a summary of changes made by a snapshot. This table is used during Conflict Resolution to quickly find out if two snapshots have conflicting changesets.

- `snapshot_id` refers to a `snapshot_id` from the `ducklake_snapshot` table.
- `changes_made` is a comma-separated list of high-level changes made by the snapshot. The values that are contained in this list have the following format:
  - `created_schema:<SCHEMA_NAME>`{:.language-sql .highlight} – the snapshot created a schema with the given name.
  - `created_table:<TABLE_NAME>`{:.language-sql .highlight} – the snapshot created a table with the given name.
  - `created_view:<VIEW_NAME>`{:.language-sql .highlight} – the snapshot created a view with the given name.
  - `inserted_into_table:<TABLE_ID>`{:.language-sql .highlight} – the snapshot inserted data into the given table.
  - `deleted_from_table:<TABLE_ID>`{:.language-sql .highlight} – the snapshot deleted data from the given table.
  - `compacted_table:<TABLE_ID>`{:.language-sql .highlight} – the snapshot run a compaction operation on the given table.
  - `dropped_schema:<SCHEMA_ID>`{:.language-sql .highlight} – the snapshot dropped the given schema.
  - `dropped_table:<TABLE_ID>`{:.language-sql .highlight} – the snapshot dropped the given table.
  - `dropped_view:<VIEW_ID>`{:.language-sql .highlight} – the snapshot dropped the given view.
  - `altered_table:<TABLE_ID>`{:.language-sql .highlight} – the snapshot altered the given table.
  - `altered_view:<VIEW_ID>`{:.language-sql .highlight} – the snapshot altered the given view.

> Names are written in quoted-format using SQL-style escapes, i.e., the name `this "table" contains quotes` is written as `"this ""table"" contains quotes"`.

ducklake_table

This table describes tables. Inception!

Column name	Column type
table_id	BIGINT
table_uuid	UUID
begin_snapshot	BIGINT
end_snapshot	BIGINT
schema_id	BIGINT
table_name	VARCHAR
path	VARCHAR
path_is_relative	BOOLEAN

- table_id is the numeric identifier of the table. table_id is incremented from next_catalog_id in the ducklake_snapshot table.
- table_uuid is a UUID that gives a persistent identifier for this table. The UUID is stored here for compatibility with existing Lakehouse formats.
- begin_snapshot refers to a snapshot_id from the ducklake_snapshot table. The table exists *starting with* this snapshot id.
- end_snapshot refers to a snapshot_id from the ducklake_snapshot table. The table exists *up to but not including* this snapshot id. If end_snapshot is NULL, the table is currently valid.
- schema_id refers to a schema_id from the ducklake_schema table.
- table_name is the name of the table, e.g., my_table.
- path is the data_path of the table.
- path_is_relative whether the path is relative to the path of the schema (true) or an absolute path (false).

ducklake_table_column_stats

This table contains column-level statistics for an entire table.

Column name	Column type
table_id	BIGINT
column_id	BIGINT
contains_null	BOOLEAN
contains_nan	BOOLEAN
min_value	VARCHAR
max_value	VARCHAR
extra_stats	VARCHAR

- table_id refers to a table_id from the ducklake_table table.
- column_id refers to a column_id from the ducklake_column table.
- contains_null is a flag whether the column contains any NULL values.
- contains_nan is a flag whether the column contains any NaN values. This is only relevant for floating-point types.
- min_value contains the minimum value for the column, encoded as a string. This does not have to be exact but has to be a lower bound. The value has to be cast to the actual type for accurate comparison, e.g., on integer types.
- max_value contains the maximum value for the column, encoded as a string. This does not have to be exact but has to be an upper bound. The value has to be cast to the actual type for accurate comparison, e.g., on integer types.
- extra_stats contains different statistics from specific types such as Geometry types.

ducklake_table_stats

This table contains table-level statistics.

Column name	Column type
table_id	BIGINT
record_count	BIGINT
next_row_id	BIGINT
file_size_bytes	BIGINT

- table_id refers to a table_id from the ducklake_table table.
- record_count is the total amount of rows in the table. This can be approximate.
- next_row_id is the row id for newly inserted rows. Used for row lineage tracking.
- file_size_bytes is the total file size of all data files in the table. This can be approximate.

ducklake_tag

Schemas, tables, and views etc can have tags, those are declared in this table.

Column name	Column type
object_id	BIGINT
begin_snapshot	BIGINT
end_snapshot	BIGINT
key	VARCHAR
value	VARCHAR

- object_id refers to a schema_id, table_id etc. from various tables above.
- begin_snapshot refers to a snapshot_id from the ducklake_snapshot table. The tag is valid *starting with* this snapshot id.
- end_snapshot refers to a snapshot_id from the ducklake_snapshot table. The tag is valid *up to but not including* this snapshot id. If end_snapshot is NULL, the tag is currently valid.
- key is an arbitrary key string. The key can't be NULL.
- value is the arbitrary value string.

ducklake_view

This table describes SQL-style VIEW definitions.

Column name	Column type
view_id	BIGINT
view_uuid	UUID
begin_snapshot	BIGINT
end_snapshot	BIGINT

Column name	Column type
schema_id	BIGINT
view_name	VARCHAR
dialect	VARCHAR
sql	VARCHAR
column_aliases	VARCHAR

- view_id is the numeric identifier of the view. view_id is incremented from next_catalog_id in the ducklake_snapshot table.
- view_uuid is a UUID that gives a persistent identifier for this view. The UUID is stored here for compatibility with existing Lakehouse formats.
- begin_snapshot refers to a snapshot_id from the ducklake_snapshot table. The view exists *starting with* this snapshot id.
- end_snapshot refers to a snapshot_id from the ducklake_snapshot table. The view exists *up to but not including* this snapshot id. If end_snapshot is NULL, the view is currently valid.
- schema_id refers to a schema_id from the ducklake_schema table.
- view_name is the name of the view, e.g., my_view.
- dialect is the SQL dialect of the view definition, e.g., duckdb.
- sql is the SQL string that defines the view, e.g., SELECT * FROM my_table.
- column_aliases contains a possible rename of the view columns. Can be NULL if no rename is set.

DuckDB Extension

Introduction

In DuckDB, DuckLake is supported through the `ducklake` extension.

Installation

Install the latest stable DuckDB. (The `ducklake` extension requires DuckDB v1.3.0 "Ossivalis" or later.)

```
INSTALL ducklake;
```

Configuration

To use DuckLake, you need to make two decisions: which metadata catalog database you want to use and where you want to store those files. In the simplest case, you use a local DuckDB file for the metadata catalog and a local folder on your computer for file storage.

Creating a New Database

DuckLake databases are created by simply starting to use them with the ATTACH statement. In the simplest case, you can create a local, DuckDB-backed DuckLake like so:

```
ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;
USE my_ducklake;
```

This will create a file `my_ducklake.ducklake`, which is a DuckDB database with the DuckLake schema.

We also use USE so we don't have to prefix all table names with `my_ducklake`. Once data is inserted, this will also create a folder `my_ducklake.ducklake.files` in the same directory, where Parquet files are stored.

If you would like to use another directory, you can specify this in the DATA_PATH parameter for ATTACH:

```
ATTACH 'ducklake:my_other_ducklake.ducklake' AS my_other_ducklake (DATA_PATH 'some/other/path/');
USE ...;
```

The path is stored in the DuckLake metadata and does not have to be specified again to attach to an existing DuckLake catalog.

> Both DATA_PATH and the database file path should be relative paths (e.g., `./some/path/` or `some/path/`). Moreover, for database creation the path needs to exist already, i.e., `ATTACH 'ducklake:db/my_ducklake.ducklake' AS my_ducklake;` where db needs to be an existing directory.

Attaching an Existing Database

Attaching to an existing database also uses the ATTACH syntax. For example, to re-connect to the example from the previous section in a new DuckDB session, we can just type:

```
ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;
USE my_ducklake;
```

Using DuckLake

DuckLake is used just like any other DuckDB database. You can create schemas and tables, insert data, update data, delete data, modify table schemas etc.

Note that – similarly to other data lake and Lakehouse formats – the DuckLake format does not support indexes, primary keys, foreign keys, and `UNIQUE` or `CHECK` constraints.

Don't forget to either specify the database name of the DuckLake explicitly or use `USE`. Otherwise you might inadvertently use the temporary, in-memory database.

Example

Let's observe what happens in DuckLake when we interact with a dataset. We will use the Netherlands train traffic dataset here.

We use the example DuckLake from above:

```
ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;
USE my_ducklake;
```

Let's now import the dataset into the a new table:

```
CREATE TABLE nl_train_stations AS
    FROM 'https://blobs.duckdb.org/nl_stations.csv';
```

Now Let's peek behind the courtains. The data was just read into a Parquet file, which we can also just query.

```
FROM glob('my_ducklake.ducklake.files/**/*');
FROM 'my_ducklake.ducklake.files/**/*.parquet' LIMIT 10;
```

But now lets change some things around. We're really unhappy with the name of the old name of the "Amsterdam Bijlmer ArenA" station now that the stadium has been renamed to "Johan Cruijff ArenA" and everyone here loves Johan. So let's change that.

```
UPDATE nl_train_stations SET name_long='Johan Cruijff ArenA' WHERE code = 'ASB';
```

Poof, its changed. We can confirm:

```
SELECT name_long FROM nl_train_stations WHERE code = 'ASB';
```

In the background, more files have appeared:

```
FROM glob('my_ducklake.ducklake.files/**/*');
```

We now see three files. The original data file, the rows that were deleted, and the rows that were inserted. Like most systems, DuckLake models updates as deletes followed by inserts. The deletes are just a Parquet file, we can query it:

```
FROM 'my_ducklake.ducklake.files/**/ducklake-*-delete.parquet';
```

The file should contain a single row that marks row 29 as deleted. A new file has appeared that contains the new values for this row.

There are now three snapshots, the table creation, data insertion, and the update. We can query that using the `snapshots()` function:

```
FROM my_ducklake.snapshots();
```

And we can query this table at each point:

```
SELECT name_long FROM nl_train_stations AT (VERSION => 1) WHERE code = 'ASB';
SELECT name_long FROM nl_train_stations AT (VERSION => 2) WHERE code = 'ASB';
```

Time travel finally achieved!

Detaching from a DuckLake

To detach from a DuckLake, make sure that your DuckLake is not your default database, then use the DETACH statement:

```
USE memory;
DETACH my_ducklake;
```

Using DuckLake from a Client

DuckDB works with any DuckDB client that supports DuckDB version 1.3.0.

Usage

Connecting

To use DuckLake, you must first either connect to an existing DuckLake, or create a new DuckLake. The ATTACH command can be used to select the DuckLake instance to connect to. In the ATTACH command, you must specify the catalog database and the data storage location. When attaching, a new DuckLake is automatically created if none exists in the specified catalog database.

Note that the data storage location only has to be specified when creating a new DuckLake. When connecting to an existing DuckLake, the data storage location is loaded from the catalog database.

```
ATTACH 'ducklake:<metadata_storage_location>' (DATA_PATH '<data_storage_location>');
```

In addition, DuckLake connection parameters can also be stored in secrets.

```
ATTACH 'ducklake:<secret_name>';
```

Examples

Connect to DuckLake, reading the configuration from the default (unnamed) secret:

```
ATTACH 'ducklake:';
```

Connect to DuckLake, reading the configuration from the secret named `my_secret`:

```
ATTACH 'ducklake:my_secret';
```

Use a DuckDB database `duckdb_database.ducklake` as the catalog database with the data path defaulting to `duckdb_database.ducklake.files`:

```
ATTACH 'ducklake:duckdb_database.ducklake';
```

Use a DuckDB database `duckdb_database.ducklake` as the catalog database with the data path explicitly specified as the `my_files` directory:

```
ATTACH 'ducklake:duckdb_database.ducklake' (DATA_PATH 'my_files/');
```

Use a PostgreSQL database as the catalog database and an S3 path as the data path:

```
ATTACH 'ducklake:postgres:dbname=postgres' (DATA_PATH 's3://my-bucket/my-data/');
```

Connect to DuckLake in read-only mode:

```
ATTACH 'ducklake:postgres:dbname=postgres' (READ_ONLY);
```

It is also possible to override the data path for a particular connection. This will not change the value of the `data_path` stored in the DuckLake metadata, but it will override it for the current connection allowing data to be stored in a different path.

```
ATTACH 'ducklake:duckdb_database.ducklake' (DATA_PATH 'other_data_path/', OVERRIDE_DATA_PATH true);
```

> If `OVERRIDE_DATA_PATH` is used, data under the original `DATA_PATH` will not be able to be queried in the current connection. This behavior may be changed in the future to allow to query data in a catalog regardless of the current write `DATA_PATH`.

Parameters

The following parameters are supported for ATTACH:

Name	Description	Default
CREATE_IF_NOT_EXISTS	Creates a new DuckLake if the specified one does not already exist	true
DATA_INLINING_ROW_LIMIT	The number of rows for which data inlining is used	0
DATA_PATH	The storage location of the data files	<metadata_file>.files{:.language-sql .highlight} for DuckDB files, required otherwise
ENCRYPTED	Whether or not data is stored encrypted	false
META_<PARAMETER_NAME>{:.language-sql .highlight}	Pass <PARAMETER_NAME>{:.language-sql .highlight} to the catalog server	
METADATA_CATALOG	The name of the attached catalog database	__ducklake_metadata_<ducklake_name>{:.language-sql .highlight}
METADATA_PARAMETERS	Map of parameters to pass to the catalog server	{}
METADATA_PATH	The connection string for connecting to the metadata catalog	
METADATA_SCHEMA	The schema in the catalog server in which to store the DuckLake tables	main
MIGRATE_IF_REQUIRED	Migrates the DuckLake schema if required	true
OVERRIDE_DATA_PATH	If the path provided in data_path differs from the stored path and this option is set to true, the path is overridden	true
SNAPSHOT_TIME	If provided, connect to DuckLake at a snapshot at a specified point in time	
SNAPSHOT_VERSION	If provided, connect to DuckLake at a specified snapshot id	

In addition, any parameters that are prefixed with META_ are passed to the catalog used to store the metadata. The supported parameters depend on the metadata catalog that is used. For example, postgres supports the SECRET parameter. By using the META_SECRET parameter we can pass this parameter to the PostgreSQL instance.

Secrets

Instead of configuring the connection using ATTACH, secrets can be created that contain all required information for setting up a connection. Secrets support the same list of parameters as ATTACH, in addition to the METADATA_PATH and METADATA_PARAMETERS parameters.

Name	Description	Default
METADATA_PATH	The connection string for connecting to the metadata	
METADATA_PARAMETERS	Map of parameters to pass to the catalog server	{}

```
-- default (unnamed) secret
CREATE SECRET (
    TYPE ducklake,
    METADATA_PATH '<metadata.db>',
    DATA_PATH '<metadata_files/>'
);

ATTACH 'ducklake:' AS my_ducklake;

-- named secrets
CREATE SECRET <my_secret> (
    TYPE ducklake,
    METADATA_PATH '',
    DATA_PATH 's3://<my-s3-bucket>/',
    METADATA_PARAMETERS MAP {'TYPE': 'postgres', 'SECRET': 'postgres_secret'}
);
ATTACH 'ducklake:<my_secret>' AS my_ducklake;
```

To persist secrets, use the `CREATE PERSISTENT SECRET` statement.

Choosing a Catalog Database

You may choose different *catalog databases* for your DuckLake. The choice depends on several factors, including whether you need to use multiple clients, which database systems are available in your organization, etc.

On the technical side, consider the following:

- If you would like to perform **local data warehousing with a single client**, use DuckDB as the catalog database.
- If you would like to perform **local data warehousing using multiple local clients**, use SQLite as the catalog database.
- If you would like to operate a **multi-user Lakehouse** with potentially remote clients, choose a transactional client-server database system as the catalog database: MySQL or PostgreSQL.

DuckDB

DuckDB can, of course, natively connect to DuckDB database files. So, to get started, you only need to install the `ducklake` extension and attach to your DuckLake:

```
INSTALL ducklake;

ATTACH 'ducklake:metadata.ducklake' AS my_ducklake;
USE my_ducklake;
```

Note that if you are using DuckDB as your catalog database, you're limited to a single client.

PostgreSQL

DuckDB can interact with a PostgreSQL database using the `postgres` extension. Install the `ducklake` and the `postgres` extension, and attach to your DuckLake as follows:

```
INSTALL ducklake;
INSTALL postgres;

-- Make sure that the database `ducklake_catalog` exists in PostgreSQL.
ATTACH 'ducklake:postgres:dbname=ducklake_catalog host=localhost' AS my_ducklake
    (DATA_PATH 'data_files/');
USE my_ducklake;
```

For details on how to configure the connection, see the `postgres` extension's documentation.

The `ducklake` and `postgresql` extensions require PostgreSQL 12 or newer.

SQLite

DuckDB can read and write a SQLite database file using the `sqlite` extension. Install the `ducklake` and the `sqlite` extension, and attach to your DuckLake as follows:

```
INSTALL ducklake;
INSTALL sqlite;

ATTACH 'ducklake:sqlite:metadata.sqlite' AS my_ducklake
    (DATA_PATH 'data_files/');
USE my_ducklake;
```

While SQLite doesn't allow concurrent reads and writes, its default mode is to ATTACH and DETACH for every query, together with providing a "retry time-out" for queries when a write-lock is encountered. This allows a reasonable amount of multi-processing support (effectively hiding the single-writer model).

MySQL

> **Warning.** There are a number of known issues with MySQL as a catalog for DuckLake. This is due to some limitations regarding the DuckDB MySQL connector. We therefore do not recommend to use MySQL as a catalog for DuckLake.

DuckDB can interact with a MySQL database using the `mysql` extension. Install the `ducklake` and the `mysql` extension, and attach to your DuckLake as follows:

```
INSTALL ducklake;
INSTALL mysql;

-- Make sure that the database `ducklake_catalog` exists in MySQL
ATTACH 'ducklake:mysql:db=ducklake_catalog host=localhost' AS my_ducklake
    (DATA_PATH 'data_files/');
USE my_ducklake;
```

For details on how to configure the connection, see the `mysql` extension's documentation.

Using the `ducklake` and `mysql` extensions require MySQL 8 or newer.

Choosing Storage

DuckLake as a concept will *never* change existing files, neither by changing existing content nor by appending to existing files. This greatly reduces the consistency requirements of file systems and greatly simplifies caching.

The DuckDB `ducklake` extension can work with any file system backend that DuckDB supports. This currently includes:

- local files and folders
- cloud object store like
  - AWS S3 and compatible (e.g., Cloudflare R2, Hetzner Object Storage, etc.)
  - Google Cloud Storage
  - Azure Blob Store
- virtual network attached file systems
  - NFS
  - SMB

  - FUSE
  - Python fsspec file systems
  - ...

When choosing storage, its important to consider the following factors

- *access latency and data transfer throughput*, a cloud further away will be accessible to everyone but have a higher latency. local files are very fast, but not accessible to anyone else. A compromise might be a site-local storage server.
- *scalability and cost*, an object store is quite* scalable, but potentially charges for data transfer. A local server might not incur significant operating expenses, but might struggle serving thousands of clients.

It might also be interesting to use DuckLake encryption when choosing external cloud storage.

Snapshots

Snapshots represent commits made to DuckLake. Every snapshot performs a set of changes that alter the state of the database. Snapshots can create tables, insert or delete data, and alter schemas.

Changes can only be made to DuckLake using snapshots. Every set of changes must be accompanied by a snapshot.

Listing Snapshots

The set of snapshots can be queried using the `snapshots` function. This returns a list of all snapshots and their changesets.

```
ATTACH 'ducklake:snapshot_test.db' AS snapshot_test;
SELECT * FROM snapshot_test.snapshots();
```

snapshot_id	snapshot_time	schema_version	changes	author	commit_message	commit_extra_info
0	2025-05-26 17:03:37.746+00	0	{schemas_created=[main]}	NULL	NULL	NULL
1	2025-05-26 17:03:38.66+00	1	{tables_created=[main.tbl]}	NULL	NULL	NULL
2	2025-05-26 17:03:38.748+00	1	{tables_inserted_into=[1]}	NULL	NULL	NULL
3	2025-05-26 17:03:39.788+00	1	{tables_deleted_from=[1]}	NULL	NULL	NULL

It is also possible to retrieve the latest snapshot id directly with a function.

```
FROM snapshot_test.current_snapshot();
```

id
3

The DuckLake extension also provides a function to get the latest committed snapshot for an existing open connection. This may be useful when multiple connections are updating the same target.

```
FROM snapshot_test.last_committed_snapshot();
```

Which would return the following for the current connection:

id
3

But if a new connection is open, it will return:

id
NULL

Adding a Commit Message to a Snapshot

An author and commit message can also be added in the context of a transaction. Optionally, you can also add some extra information.

```
CREATE TABLE ducklake.people (a INTEGER, b VARCHAR);

-- Begin Transaction
BEGIN;
INSERT INTO ducklake.people VALUES (1, 'pedro');
CALL ducklake.set_commit_message('Pedro', 'Inserting myself', extra_info => '{''foo'': 7, ''bar'': 10}');
COMMIT;
-- End transaction
```

snapshot_id	snapshot_time	schema_version	changes	author	commit_message	commit_extra_info
0	2025-08-18 13:10:49.636+02	0	{schemas_created=[main]}	NULL	NULL	NULL
1	2025-08-18 13:24:15.472+02	1	{tables_created=[main.t1]}	NULL	NULL	NULL
2	2025-08-18 13:25:24.423+02	2	{tables_created=[main.people]}	NULL	NULL	NULL
3	2025-08-18 13:26:06.38+02	2	{tables_inserted_into=[2]}	Pedro	Inserting myself	{'foo':7, 'bar':10}

Schema Evolution

DuckLake supports the evolution of the schemas of tables without requiring any data files to be rewritten. The schema of a table can be changed using the `ALTER TABLE` statement. The following statements are supported:

Adding Columns / Fields

```
-- add a new column of type INTEGER, with default value NULL
ALTER TABLE tbl ADD COLUMN new_column INTEGER;
-- add a new column with an explicit default value
ALTER TABLE tbl ADD COLUMN new_column VARCHAR DEFAULT 'my_default';
```

Fields can be added to columns of type `struct`. The path to the `struct` column must be specified, followed by the name of the new field and the type of the new field.

```
-- add a new field of type INTEGER, with default value NULL
ALTER TABLE tbl ADD COLUMN nested_column.new_field INTEGER;
```

Dropping Columns / Fields

```
-- drop the top-level column `new_column` from the table
ALTER TABLE tbl DROP COLUMN new_column;
```

Fields can be dropped by specifying the full path to the field.

```
-- drop the field `new_field` from the struct column `nested_column`
ALTER TABLE tbl DROP COLUMN nested_column.new_field;
```

Renaming Columns / Fields

```
-- rename the top-level column "new_column" to "new_name"
ALTER TABLE tbl RENAME new_column TO new_name;
```

Field scan be renamed by specifying the full path to the field.

```
-- rename the field "new_field" within the struct column "nested_column" to "new_name"
ALTER TABLE tbl RENAME nested_column.new_field TO new_name;
```

Renaming Tables

```
-- rename the table "tbl" to "tbl_new_name"
ALTER TABLE tbl RENAME TO tbl_new_name;
```

Type Promotion

The types of columns can be changed.

```
-- change the type of col1 to BIGINT
ALTER TABLE tbl ALTER col1 SET TYPE BIGINT;
-- change the type of field "new_field" within the struct column "nested_column" to BIGINT
ALTER TABLE tbl ALTER nested_column.new_field SET TYPE BIGINT;
```

Note that not all type changes are valid. Only *type promotions* are supported. Type promotions must be lossless. As such, valid type promotions are promoting from a narrower type (`int32`) to a wider type (`int64`).

The full set of valid type promotions is as follows:

Source	Target
`int8`	`int16`, `int32`, `int64`
`int16`	`int32`, `int64`
`int32`	`int64`
`uint8`	`uint16`, `uint32`, `uint64`
`uint16`	`uint32`, `uint64`
`uint32`	`uint64`
`float32`	`float64`

Field Identifiers

Columns are tracked using **field identifiers**. These identifiers are stored in the `column_id` field of the `ducklake_column` table. The identifiers are also written to each of the data files. For Parquet files, these are written in the `field_id` field. These identifiers are used to reconstruct the data of a table for a given snapshot.

When reading the data for a table, the schema together with the correct field identifiers is read from the `ducklake_column` table. Data files can contain any number of columns that exist in that schema, and can also contain columns that do not exist in that schema.

- If we drop a column, previously written data files still contain the dropped column.
- If we add a column, previously written data files do not contain the new column.
- If we change the type of a column, previously written data files contain data for the column in the old type.

To reconstruct the correct table data for a given snapshot, we must perform *field id remapping*. This is done as follows:

- Data for a column is read from the column with the corresponding `field_id`. The data types might not match in case of type promotion. In this case, the values must be cast to the correct type of the column.
- Any column that has a `field_id` that exists in the data file but not in the table schema must be ignored
- Any column that has a `field_id` that does not exist in the data file must be replaced with the `initial_default` value in the `ducklake_column` table

Time Travel

In DuckLake, every snapshot represents a consistent state of the database. DuckLake keeps a record of all historic snapshots and their changesets, unless compaction is triggered and historic snapshots are explicitly deleted.

Using time travel, it is possible to query the state of the database as of any recorded snapshot. The snapshot to query can be specified either (1) using a timestamp, or (2) explicitly using a snapshot identifier. The `snapshots` function can be used to obtain a list of valid snapshots for a given DuckLake database.

Examples

Query the table at a specific snapshot version.

```
SELECT * FROM tbl AT (VERSION => 3);
```

Query the table as it was last week.

```
SELECT * FROM tbl AT (TIMESTAMP => now() - INTERVAL '1 week');
```

Attach a DuckLake database at a specific snapshot version.

```
ATTACH 'ducklake:file.db' (SNAPSHOT_VERSION 3);
```

Attach a DuckLake database as it was at a specific time.

```
ATTACH 'ducklake:file.db' (SNAPSHOT_TIME '2025-05-26 00:00:00');
```

Upserting

Upserting is the combination of updating and inserting. In database operations this usually means *do something to a record if it already exists* and *do something else if it doesn't*. Many databases support primary keys to assist with this behavior. This is also the case with DuckDB, which allows for the syntax `INSERT INTO ... ON CONFLICT`.

DuckLake, on the other hand, does not support primary keys. However, the `MERGE INTO` syntax provides the same upserting functionality.

Syntax

```
MERGE INTO target_table [target_alias]
    USING source_table [source_alias]
    ON (target_table.field = source_table.field) -- USING (field)
    WHEN MATCHED THEN UPDATE [SET] | DELETE
    WHEN NOT MATCHED THEN INSERT;
```

Usage

First, let's create a simple table.

```
CREATE TABLE people(id INTEGER, name VARCHAR, salary FLOAT);
INSERT INTO people VALUES (1, 'John', 92_000.0), (2, 'Anna', 100_000.0);
```

The simplest upsert would be updating or inserting a whole row.

```
MERGE INTO people
    USING (
        SELECT
            unnest([3, 1]) AS id,
            unnest(['Sarah', 'Jhon']) AS name,
            unnest([95_000.0, 105_000.0]) AS salary
    ) AS upserts
    ON (upserts.id = people.id)
    WHEN MATCHED THEN UPDATE
    WHEN NOT MATCHED THEN INSERT;

FROM people;
```

id	name	salary
1	Jhon	92000.0
3	Sarah	95000.0
2	Anna	105000.0

In the previous example we are updating the whole row if `id` matches. However, it is also a common pattern to receive a change set with some keys and the changed value. This is a good use for SET.

```
MERGE INTO people
    USING (
        SELECT
            1 AS id,
            98_000.0 AS salary
    ) AS salary_updates
    ON (salary_updates.id = people.id)
    WHEN MATCHED THEN UPDATE SET salary = salary_updates.salary;

FROM people;
```

id	name	salary
3	Sarah	95000.0
2	Anna	105000.0
1	Jhon	98000.0

Another common pattern is to receive a delete set of rows, which may only contain ids of rows to be deleted.

```
MERGE INTO people
    USING (
        SELECT
            1 AS id,
    ) AS deletes
    ON (deletes.id = people.id)
    WHEN MATCHED THEN DELETE;

FROM people;
```

id	name	salary
3	Sarah	95000.0
2	Anna	105000.0

`MERGE INTO` also supports more complex conditions, for example for a given delete set we can decide to only remove rows that contain a `salary` bigger than a certain amount.

```
MERGE INTO people
    USING (
        SELECT
            unnest([3, 2]) AS id,
    ) AS deletes
    ON (deletes.id = people.id)
    WHEN MATCHED AND people.salary > 100_000.0 THEN DELETE;

FROM people;
```

id	name	salary
3	Sarah	95000.0

Unsupported Behavior

Multiple `UPDATE` or `DELETE` operators are not currently supported. The following query **would not work**:

```
MERGE INTO people
    USING (
        SELECT
            unnest([3, 1]) AS id,
            unnest(['Sarah', 'Jhon']) AS name,
            unnest([95_000.0, 105_000.0]) AS salary
    ) AS upserts
    ON (upserts.id = people.id)
    WHEN MATCHED AND people.salary < 100_000.0 THEN UPDATE
    -- Second update or delete condition
    WHEN MATCHED AND people.salary > 100_000.0 THEN DELETE
    WHEN NOT MATCHED THEN INSERT;

Not implemented Error:
MERGE INTO with DuckLake only supports a single UPDATE/DELETE action currently
```

Configuration

`ducklake` Extension Configuration

The `ducklake` extension also allows for some configuration regarding retry mechanism for transaction conflicts.

Option List

Name	Description	Default
`ducklake_max_retry_count`	The maximum amount of retry attempts for a DuckLake transaction	10
`ducklake_retry_wait_ms`	Time between retries in ms	100
`ducklake_retry_backoff`	Backoff factor for exponentially increasing retry wait time	1.5

Setting Config Values

```
SET ducklake_max_retry_count = 100;
SET ducklake_retry_wait_ms = 100;
SET ducklake_retry_backoff = 2;
```

DuckLake Specific Configuration

DuckLake supports persistent and scoped configuration operations. These options can be set using the `set_option` function. The options that have been set can be queried using the `options` function. Configuration is persisted in the `ducklake_metadata` table.

Option List

Name	Description	Default
`data_inlining_row_limit`	Maximum amount of rows to inline in a single insert	0
`parquet_compression`	Compression algorithm for Parquet files (uncompressed, snappy, gzip, zstd, brotli, lz4, lz4_raw)	snappy
`parquet_version`	Parquet format version (1 or 2)	1
`parquet_compression_level`	Compression level for Parquet files	3
`parquet_row_group_size`	Number of rows per row group in Parquet files	122880
`parquet_row_group_size_bytes`	Number of bytes per row group in Parquet files	
`hive_file_pattern`	If partitioned data should be written in a hive-like folder structure	true
`target_file_size`	The target data file size for insertion and compaction operations	512MB
`version`	DuckLake format version	
`created_by`	Tool used to write the DuckLake	
`data_path`	Path to data files	

Name	Description	Default
`require_commit_message`	If an explicit commit message is required for a snapshot commit.	false
`rewrite_delete_threshold`	Minimum fraction of data removed from a file before a rewrite is warranted (0...1)	0.95
`delete_older_than`	How old unused files must be to be removed by cleanup functions	
`expire_older_than`	How old snapshots must be to be expired by default	
`compaction_schema`	Pre-defined schema used as a default value for compaction functions	
`compaction_table`	Pre-defined table used as a default value for compaction functions	
`encrypted`	Whether or not to encrypt Parquet files written to the data path	false
`per_thread_output`	Whether to create separate output files per thread during parallel insertion	false

Setting Config Values

```
-- set the global parquet compression algorithm used when writing Parquet files
CALL my_ducklake.set_option('parquet_compression', 'zstd');
-- set the parquet compression algorithm used for tables in a specific schema
CALL my_ducklake.set_option('parquet_compression', 'zstd', schema => 'my_schema');
-- set the parquet compression algorithm used for a specific table
CALL my_ducklake.set_option('parquet_compression', 'zstd', table_name => 'my_table');

-- see all options for the given attached DuckLake
FROM my_ducklake.options();
```

Scoping

Options can be set either globally, per-schema or per-table. The most specific scope that is set is always used for any given setting, i.e., settings are used in the following priority:

Priority	Setting Scope
1	Table
2	Schema
3	Global
4	Default

Paths

DuckLake manages files stored in a separate storage location. The paths to the files are stored in the catalog server. Paths can be either absolute or relative to their parent path. Whether or not a path is relative is stored in the `path_is_relative` column, alongside the `path`. By default, all paths written by DuckLake are relative paths.

Path type	Path location	Parent path
File path	`ducklake_data_file` / `ducklake_delete_file`	Table path
Table path	`ducklake_table`	Schema path

Path type	Path location	Parent path
Schema path	`ducklake_schema`	Data path
Data path	`ducklake_metadata`	

Default Path Structure

The root `data_path` is specified through the `data_path` parameter when creating a new DuckLake. When loading an existing DuckLake, the `data_path` is loaded from the `ducklake_metadata` if not provided.

Schemas. When creating a schema, a schema path is set. By default, this path is the name of the schema for alphanumeric names (`{schema_name}/`) – or `{schema_uuid}/` otherwise. This path is set as relative to the root `data_path`.

Tables. When creating a table, a table path is set. By default, this path is the name of the schema for alphanumeric names (`{table_name}/`) – or `{table_uuid}/` otherwise. This path is set as relative to the path of the parent schema.

Files. When writing a new data or delete file to the table, a new file path is generated. For unpartitioned tables, this path is `ducklake-{uuid}.parquet` – relative to the table path.

Partitioned Files. When writing data to a partitioned table, the files are by default written to directories in the hive partitioning style. Writing data in this manner is not required as the partition values are tracked in the catalog server itself. For encrypted tables, the partitioned paths are omitted, and the files are all writte

This results in the following path structure:

```
main
├── unpartitioned_table
│   └── ducklake-{uuid}.parquet
└── partitioned_table
    └── year=2025
        └── ducklake-{uuid}.parquet
```

Maintenance

Recommended Maintenance

Metadata Maintenance

Most operations performed by DuckLake happen in the catalog database. As such, the maintenance of the metadata server are handled by the chosen catalog database. For example, when running PostgreSQL, it is likely sufficient to occasionally run `VACUUM` in order to ensure the system stays performant.

Data File Maintenance

The data files that DuckLake writes to the data directory may require maintenance depending on how the insertions take place. When snapshots write small batches of data at a time and data inlining is not used small Parquet files will be written to storage. It is recommended to merge these Parquet files using the `merge_adjacent_files` function.

DuckLake also never deletes old data files. As old data remains accessible through time travel. Even when a table is dropped, the data files associated with that table are not deleted. In order to trigger a delete of these files, the snapshots that refer to that table must be expired and files should be cleaned up

If you have tables that are heavily deleted, it can be the case that you have a lot of delete files that will slow read performance. In this case, we recommend you run a function to rewrite the deleted files.

If you need to run all of this operations periodically, then we recommend you use the `CHECKPOINT` statement.

Merge Adjacent Files

Unless data inlining is used, each insert to DuckLake writes data to a new Parquet file. If small insertions are performed, the Parquet files that are written are small. These only hold a few rows. For performance reasons, it is generally recommended that Parquet files are at least a few megabytes each.

DuckLake supports merging of files **without needing to expire snapshots**. This is supported due to the *lightweight snapshots* that can refer to a part of a Parquet file. Effectively, we can merge multiple Parquet files into a single Parquet file that holds data inserted by multiple snapshots. The data file is then setup so that the snapshots refer to only part of that Parquet file.

This preserves all of the original behavior – including time travel and data change feeds – for these snapshots. In effect, this manner of compaction is completely transparent from a user perspective.

This compaction technique can be triggered using the `merge_adjacent_files` function. For example:

```
CALL ducklake_merge_adjacent_files('my_ducklake');
```

Or if you want to target a specific table within a schema:

```
CALL ducklake_merge_adjacent_files('my_ducklake', 't', schema => 'some_schema');
```

> Calling this function does not immediately delete the old files. See the cleanup old files section on how to trigger a clean-up of these files.

Expire Snapshots

DuckLake in normal operation never removes any data, even when tables are dropped or data is deleted. Due to time travel, the removed data is still accessible.

Data can only be physically removed from DuckLake by expiring snapshots that refer to the old data. This can be done using the `ducklake_expire_snapshots` function.

Usage

The below command expires a snapshot with a specific snapshot id.

```
CALL ducklake_expire_snapshots('ducklake', versions => [2]);
```

The below command expires snapshots older than a week.

```
CALL ducklake_expire_snapshots('ducklake', older_than => now() - INTERVAL '1 week');
```

The below command performs a *dry run*, which only lists the snapshots that will be deleted, instead of actually deleting them.

```
CALL ducklake_expire_snapshots('ducklake', dry_run => true, older_than => now() - INTERVAL '1 week');
```

It is also possible to set a DuckLake option to expire snapshots that applies to the whole catalog.

```
CALL ducklake.set_option('expire_older_than', '1 month');
```

Cleaning Up Files

Note that expiring snapshots does not immediately delete files that are no longer referenced. See the cleanup old files section on how to trigger a clean-up of these files.

Cleanup of Files

In DuckLake you may want to delete data files that are either a part of an expired snapshot or are simply not tracked by DuckLake anymore (i.e., orphaned files).

Cleanup of Files from Expired Snapshots

When files are no longer required in DuckLake, due to e.g. snapshots being expired or files being merged, they are not immediately deleted. The reason for this is that there might still be active queries that are scanning these files.

The files are instead added to the `ducklake_files_scheduled_for_deletion` table. These files can then be deleted at a later point. It is generally safe to delete files that have been scheduled for deletion more than a few days ago, provided there are no read transactions that last that long. The files can be deleted using the `ducklake_cleanup_old_files` function.

Usage

The below command deletes all files scheduled for deletion.

```
CALL ducklake_cleanup_old_files('ducklake', cleanup_all => true);
```

The below command deletes all files that were scheduled for deletion more than a week ago.

```
CALL ducklake_cleanup_old_files('ducklake', older_than => now() - INTERVAL '1 week');
```

The below command performs a *dry run*, which only lists the files that will be deleted, instead of actually deleting them.

```
CALL ducklake_cleanup_old_files('ducklake', dry_run => true, older_than => now() - INTERVAL '1 week');
```

Cleanup of Orphaned Files

Orphan files are files that are untracked by the DuckLake metadata catalog. They may appear due to, for example, a systems failure. DuckLake provides the `ducklake_delete_orphaned_files` function to delete these files.

Usage

The below command deletes all orphaned files.

```
CALL ducklake_delete_orphaned_files('ducklake', cleanup_all => true);
```

The below command deletes all files that are older than a specified time.

```
CALL ducklake_delete_orphaned_files('ducklake', older_than => now() - INTERVAL '1 week');
```

The below command performs a *dry run*, which only lists the files that will be deleted, instead of actually deleting them.

```
CALL ducklake_delete_orphaned_files('ducklake', dry_run => true, older_than => now() - INTERVAL '1 week');
```

There is also a catalog-level option available.

```
CALL ducklake.set_option('delete_older_than', '1 week');
```

Rewrite Heavily Deleted Files

DuckLake uses a merge-on-read strategy when data is deleted from a table. In short, this means that DuckLake uses a delete file which contains a pointer to the deleted records on the original file. This makes deletes very efficient. However, for heavily deleted tables, reading performance will be hindered by this approach. To solve this problem, DuckLake exposes a function called `ducklake_rewrite_data_files` that rewrites files that contain an amount of deletes bigger than a given threshold to a new file that contains non-deleted records. This files can then be further compacted with a `ducklake_merge_adjacent_files` operation. The default value for the delete threshold is 0.95.

Usage

Apply to all tables in a catalog:

```
CALL ducklake_rewrite_data_files('my_ducklake');
```

Apply only to a specific table:

```
CALL ducklake_rewrite_data_files('my_ducklake', 't');
```

Provide a specific value for the delete threshold:

```
CALL ducklake_rewrite_data_files('my_ducklake', 't', delete_threshold => 0.5);
```

Set a specific threshold for the whole catalog:

```
CALL my_ducklake.set_option('rewrite_delete_threshold', 0.5);
```

Checkpoint

DuckLake provides the option to implement all the maintenance functions bundled in the `CHECKPOINT` statement. This statement will run in order the following DuckLake functions:

- `ducklake_flush_inlined_data`

- `ducklake_expire_snapshots`
- `ducklake_merge_adjacent_files`
- `ducklake_rewrite_data_files`
- `ducklake_cleanup_old_files`
- `ducklake_delete_orphaned_files`

Usage

The `CHECKPOINT` statement takes the following global DuckLake options:

- `rewrite_delete_threshold`: A threshold that determines the minimum amount of data that must be removed from a file before a rewrite is warranted (0...1). Used by `ducklake_rewrite_data_files`.
- `delete_older_than`: How old unused files must be to be removed by the `ducklake_delete_orphaned_files` and `ducklake_cleanup_old_files` cleanup functions.
- `expire_older_than`: How old snapshots must be, by default, to be expired by `ducklake_expire_snapshots`.
- `compaction_schema`: Pre-defined schema used as a default value for the following compaction functions `ducklake_flush_inlined_data`,`ducklake_merge_adjacent_files`, `ducklake_rewrite_data_files`, `ducklake_delete_orphaned_files`.
- `compaction_table`: Pre-defined table used as a default value for the following compaction functions `ducklake_flush_inlined_data`,`ducklake_merge_adjacent_files`, `ducklake_rewrite_data_files`, `ducklake_delete_orphaned_files`.

If this options are not provided via the `ducklake.set_option` function, `CHECKPOINT` will use the default values when applicable and will run a `CHECKPOINT` of the whole DuckLake.

```
CHECKPOINT;
```

Advanced Features

Constraints

DuckLake has limited support for constraints. The only constraint type that is currently supported is `NOT NULL`. It does not support `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE` or `CHECK` constraints.

Examples

Define a column as not accepting `NULL` values using the `NOT NULL` constraint.

```
CREATE TABLE tbl (col INTEGER NOT NULL);
```

Add a `NOT NULL` constraint to an existing column of an existing table.

```
ALTER TABLE tbl ALTER col SET NOT NULL;
```

Drop a `NOT NULL` constraint from a table.

```
ALTER TABLE tbl ALTER col DROP NOT NULL;
```

Conflict Resolution

In DuckLake, snapshot identifiers are written in a sequential order. The first snapshot has `snapshot_id` 0, and subsequent snapshot ids are monotonically increasing such that the second snapshot has id 1, etc. The sequential nature of snapshot identifiers is used to **detect conflicts** between snapshots. The `ducklake_snapshot` table has a `PRIMARY KEY` constraint defined over the `snapshot_id` column.

When two connections try to write to a `ducklake` table, they will try to write a snapshot with the same identifier and one of the transactions will trigger a `PRIMARY KEY` constraint violation and fail to commit. When such a conflict occurs – we try to resolve the conflict. In many cases, such as when both transactions are inserting data into a table, we can retry the commit without having to rewrite any actual files. During conflict resolution, we query the `ducklake_snapshot_changes` table to figure out the high-level set of changes that other snapshots have made in the meantime.

- If there are no logical conflicts between the changes that the snapshots have made – we automatically retry the transaction in the metadata catalog without rewriting any data files.
- If there are logical conflicts, we abort the transaction and instruct the user that conflicting changes have been made.

Logical Conflicts

Below is a list of logical conflicts based on the snapshot's changeset. Snapshots conflict when any of the following conflicts occur:

Schemas

- Both snapshots create a schema with the same name
- Both snapshots drop the same schema
- A snapshot tries to drop a schema in which another transaction created an entry

Tables / Views

- Both snapshots create a table or view with the same name in the same schema
- A snapshot tries to create a table or view in a schema that was dropped by another snapshot
- Both snapshots drop the same table or view
- A snapshot tries to alter a table or view that was dropped or altered by another snapshot

Data

- A snapshot tries to insert data into a table that was dropped or altered by another snapshot
- A snapshot tries to delete data from a table that was dropped, altered, deleted from or compacted by another snapshot

Compaction

- A snapshot tries to compact a table that was deleted from by another snapshot
- A snapshot tries to compact a table that was dropped by another snapshot

Data Change Feed

In addition to allowing you to query the state of the database at any point in time, DuckLake allows you to query the *changes that were made between any two snapshots*. This can be done using the `table_changes` function.

Examples

Consider the following DuckLake instance:

```
ATTACH 'ducklake:changes.db' AS db (DATA_PATH 'change_files/');
-- snapshot 1
CREATE TABLE db.tbl(id INTEGER, val VARCHAR);
-- snapshot 2
INSERT INTO db.tbl VALUES (1, 'Hello'), (2, 'DuckLake');
-- snapshot 3
DELETE FROM db.tbl WHERE id = 1;
-- snapshot 4
UPDATE db.tbl SET val = concat(val, val, val);
```

Changes Made by a Specific Snapshot

```
FROM db.table_changes('tbl', 2, 2);
```

snapshot_id	rowid	change_type	id	val
2	0	insert	1	Hello
2	1	insert	2	DuckLake

Changes Made between Multiple Snapshots

```
FROM db.table_changes('tbl', 3, 4);
```

|------------:|------:|-----------------|---:|-------------------------| | 3 | 0 | delete | 1 | Hello | | 4 | 1 | update_postimage | 2 | DuckLakeDuckLakeDuckLake | | 4 | 1 | update_preimage | 2 | DuckLake |

Changes Made in the Last Week

```
FROM changes.table_changes('tbl', now() - INTERVAL '1 week', now());
```

`table_changes`

The `table_changes` function takes as input the table for which changes should be returned, and two bounds: the start snapshot and the end snapshot (inclusive). The bounds can be given either as a snapshot id, or as a timestamp.

The result of the function is the set of changes, read using the table schema as of the end snapshot provided, and three extra columns: `snapshot_id`, `rowid` and `change_type`.

Column	Description
snapshot_id	The snapshot which made the change
rowid	The row identifier of the row which was changed
change_type	insert, update_preimage, update_postimage or delete

Updates are split into two rows: the `update_preimage` and `update_postimage`. `update_preimage` is the row as it was prior to the update operation. `update_postimage` is the row as it is after the update operation.

When the schema of a table is altered, changes are read as of the schema of the table as of the end snapshot. As such, if a column is dropped in between the provided bounds, the dropped column is omitted from the entire result. If a column is added, any changes made to the table prior to the addition of the column will have the column substituted with its default value.

Compaction

Compaction operations that expire snapshots can limit the change feed that can be read. For example, if deleted rows are removed as part of compaction, these cannot be returned by the change feed anymore.

Data Inlining

> Data Inlining is currently experimental. It needs to be enabled explicitly and is only supported for DuckDB databases. We are planning to improve support for this feature in the future.

When writing small changes to DuckLake, it can be wasteful to write each changeset to an individual Parquet file. DuckLake supports directly writing small changes to the metadata using Data Inlining. Instead of writing a Parquet file to the data storage and then writing a reference to that file in the metadata catalog, we directly write the rows to inlined data tables within the metadata catalog.

Data inlining must be enabled explicitly using the `DATA_INLINING_ROW_LIMIT` attach parameter. When enabled, any inserts that write fewer than the given amount of rows are automatically written to inlined tables instead.

```
ATTACH 'ducklake:inlining.db' (DATA_INLINING_ROW_LIMIT 10);
```

Inlined data behaves exactly the same as data written to Parquet files. The inlined data can be queried, updated and deleted, and the schema of inlined data can be modified. The only difference is that the inlined data lives in the metadata catalog, instead of in Parquet files in the data path.

For example, when inserting a low number of rows, data is automatically inlined:

```
CREATE TABLE inlining.tbl(col INTEGER);
-- inserting 3 rows, data is inlined
INSERT INTO inlining.tbl VALUES (1), (2), (3);
-- no Parquet files exist
SELECT COUNT(*) FROM glob('inlining.db.files/**');
```

count_star()
int64
0

When inserting more data than the `DATA_INLINING_ROW_LIMIT`, inserts are automatically written to Parquet:

```
INSERT INTO inlining.tbl FROM range(100);
SELECT COUNT(*) FROM glob('inlining.db.files/**');
```

count_star()
int64
1

Flushing Inlined Data

Inlined data can be manually flushed to parquet files by calling the `ducklake_flush_inlined_data` function. For example:

```
-- flush all inlined data in all schemas and tables
CALL ducklake_flush_inlined_data('my_ducklake');
-- flush inlined data only within a specific schema
CALL ducklake_flush_inlined_data('my_ducklake', schema_name => 'my_schema');
-- flush inlined data for a specific table in the default 'main' schema
CALL ducklake_flush_inlined_data('my_ducklake', table_name => 'my_table');
-- flush inlined data for a specific table in a specific schema
CALL ducklake_flush_inlined_data('my_ducklake', schema_name => 'my_schema', table_name => 'my_table');
```

Encryption

DuckLake supports an encrypted mode. In this mode, all files that are written to the data directory are encrypted using Parquet encryption. In order to use this mode, the ENCRYPTED flag must be passed when initializing the DuckLake catalog:

```
ATTACH 'ducklake:encrypted.ducklake'
    (DATA_PATH 'untrusted_location/', ENCRYPTED);
```

When enabled, all Parquet files that are written as part of DuckLake operations are automatically encrypted. The encryption keys for each file are automatically generated by the system when the files are written. New encryption keys are automatically generated for each write operation – such that each file is encrypted using their own encryption key. The generated keys are stored in the catalog, in the `encryption_key` field of the `ducklake_data_file` table.

When data is read from the encrypted files, the keys are read from the catalog server and automatically used to decrypt the files. This allows encrypted DuckLake databases to be interacted with in exactly the same manner as unencrypted databases.

Partitioning

DuckLake tables can be partitioned by a user-defined set of partition keys. When a DuckLake table has partitioning keys defined, any new data is split up into separate data files along the partitioning keys. During query planning, the partitioning keys are used to prune which files are scanned.

The partitioning keys defined on a table only affect new data written to the table. Previously written data will be kept partitioned by the keys the table had when that data was written. This allows the partition layout of a table to evolve over-time as needed.

The partitioning keys for a file are stored in DuckLake. These keys do not need to be necessarily stored within the files, or in the paths to the files.

Examples

> By default, DuckLake supports Hive style partitioning. If you want to avoid this style of partitions, you can opt out via using `CALL my_ducklake.set_option('hive_file_pattern', false)`

Set the partitioning keys of a table, such that new data added to the table is partitioned by these keys.

```
-- partition on a column
ALTER TABLE tbl SET PARTITIONED BY (part_key);
-- partition by the year/month of a timestamp
ALTER TABLE tbl SET PARTITIONED BY (year(ts), month(ts));
```

Remove the partitioning keys of a table, such that new data added to the table is no longer partitioned.

```
ALTER TABLE tbl RESET PARTITIONED BY;
```

DuckLake supports the following partition clauses:

Transform	Expression
identity	col_name
year	year(ts)
month	month(ts)
day	day(ts)
hour	hour(ts)

Transactions

DuckLake has full support for ACID and offers snapshot isolation for all interactions with the database. All operations, including DDL statements such as `CREATE TABLE` or `ALTER TABLE`, have full transactional support. Transactions have all-or-nothing semantics and can be composed of multiple changes that are made to the database.

The extension also provides some syntax to be able to manage transactions. This is explained in the DuckDB documentation. Basically it comes down to this:

```
BEGIN TRANSACTION;
-- Some operation
-- Some other operation
COMMIT;
-- or
ROLLBACK; -- ABORT will have the same behavior
```

In the context of DuckLake, one committed transaction (i.e., a `BEGIN-COMMIT` block) represents one snapshot.

If multiple transactions are being performed concurrently in one table, the `ducklake` extension has some default configurations for a retry mechanism. This default configurations can be overridden.

Row Lineage

Every row created in DuckLake has a unique row identifier, which can be queried as the `rowid` virtual column. This identifier is assigned when a row is first inserted into the system. The identifier is preserved when the row is moved between files – for example as part of `UPDATE` and compaction operations.

The `rowid` column can be used to track whether the addition of files actually introduces new rows into DuckLake, or whether the operation is simply moving files around. This is used internally in the data change feed to differentiate between update operations and delete + insert operations.

Views

Views can be created using the standard `CREATE VIEW` syntax. The views are stored in the metadata, in the `ducklake_view` table.

Examples

Create a view.

```
CREATE VIEW v1 AS SELECT * FROM tbl;
```

Comments

Comments can be added to tables, views and columns using the `COMMENT ON` syntax. The comments are stored in the metadata, and can be modified in a transactional manner.

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';
```

Metadata

List Files

The `ducklake_list_files` function can be used to list the data files and corresponding delete files that belong to a given table, optionally for a given snapshot.

Usage

```
-- list all files
FROM ducklake_list_files('catalog', 'table_name');
-- get list of files at a specific snapshot_version
FROM ducklake_list_files('catalog', 'table_name', snapshot_version => 2);
-- get list of files at a specific point in time
FROM ducklake_list_files('catalog', 'table_name', snapshot_time => '2025-06-16 15:24:30');
-- get list of files of a table in a specific schema
FROM ducklake_list_files('catalog', 'table_name', schema => 'main');
```

Parameter	Description	Default
catalog	Name of attached DuckLake catalog	
table_name	Name of table to fetch files from	
schema	Schema for the table	main
snapshot_version	If provided, fetch files for a given snapshot id	
snapshot_time	If provided, fetch files for a given timestamp	

Result

The function returns the following result set.

column_name	column_type
data_file	VARCHAR
data_file_size_bytes	UBIGINT
data_file_footer_size	UBIGINT
data_file_encryption_key	BLOB
delete_file	VARCHAR
delete_file_size_bytes	UBIGINT
delete_file_footer_size	UBIGINT
delete_file_encryption_key	BLOB

- If the file has delete files, the corresponding delete file is returned, otherwise these fields are `NULL`.
- If the database is encrypted, the encryption key must be used to read the file.
- The `footer_size` refers to the Parquet footer size – this is optionally provided.

Adding Files

The `ducklake_add_data_files` function can be used to register existing data files as new files in DuckLake. The files are not copied over – DuckLake is merely made aware of their existence, allowing them to be queried through DuckLake. Adding files in this manner supports regular transactional semantics.

> **Note.** that ownership of the Parquet file is transferred to DuckLake. As such, compaction operations (such as those triggered through `merge_adjacent_files` or `expire_snapshots` followed by `cleanup_old_files`) can cause the files to be deleted by DuckLake.

Usage

```
-- add the file "people.parquet" to the "people" table in "my_ducklake"
CALL ducklake_add_data_files('my_ducklake', 'people', 'people.parquet');
-- target a specific schema rather than the default main
CALL ducklake_add_data_files('my_ducklake', 'people', 'people.parquet', schema => 'some_schema');
-- add the file - any columns that are present in the table but not in the file will have their default
values used when reading
CALL ducklake_add_data_files('my_ducklake', 'people', 'people.parquet', allow_missing => true);
-- add the file - if the file has extra columns in the table they will be ignored (they will not be
queryable through DuckLake)
CALL ducklake_add_data_files('my_ducklake', 'people', 'people.parquet', ignore_extra_columns => true);
```

Missing Columns

When adding files to a table, all columns that are present in the table must be present in the Parquet file, otherwise an error is thrown. The `allow_missing` option can be used to add the file anyway – in which case any missing columns will be substituted with the `initial_default` value of the column.

Extra Columns

When adding files to a table, if there are any columns present that are not present in the table, an error is thrown by default. The `ignore_extra_columns` option can be used to add the file anyway – any extra columns will be ignored and unreachable.

Type Mapping

In general, types of columns in the source Parquet file must match the type as defined in the table, otherwise an error is thrown. Types in the Parquet file can be narrower than the type defined in the table. Below is a supported mapping type:

Table type	Supported Parquet types
`bool`	`bool`
`int8`	`int8`
`int16`	`int[8/16]`,`uint8`
`int32`	`int[8/16/32]`,`uint[8/16]`
`int64`	`int[8/16/32/64]`,`uint[8/16/32]`
`uint8`	`uint8`
`uint16`	`uint[8/16]`
`uint32`	`uint[8/16/32]`
`uint64`	`uint[8/16/32/64]`

Table type	Supported Parquet types
`float`	`float`
`double`	`float/double`
`decimal(P, S)`	`decimal(P',S')`, where `P' <= P` and `S' <= S`
`blob`	`blob`
`varchar`	`varchar`
`date`	`date`
`time`	`time`
`timestamp`	`timestamp`, `timestamp_ns`
`timestamp_ns`	`timestamp`, `timestamp_ns`
`timestamptz`	`timestamptz`

Migrations

DuckDB to DuckLake

Migrating from DuckDB to DuckLake is very simple to do with the DuckDB `ducklake` extension. However, if you are currently using some DuckDB features that are unsupported in DuckLake, this guide will definitely help you.

First Scenario: Everything is Supported

If you are not using any of the unsupported features, migrating from DuckDB to DuckLake will be as simple as running the following commands:

```
ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;
ATTACH 'duckdb.db' AS my_duckdb;

COPY FROM DATABASE my_duckdb TO my_ducklake;
```

Note that it doesn't matter what catalog you are using as a metadata backend for DuckLake.

Second Scenario: Not Everything is Supported

If you have been using DuckDB for a while, there is a chance you are using some very specific types, macros, default values that are not literals or even things like generated columns. If this is your case, then migrating will have some tradeoffs.

- Specific types need to be cast to a supported DuckLake type. User defined types that are created as a `STRUCT` can be interpreted as such and `ENUM` and `UNION` will be cast to `VARCHAR` and `VARINT` will be cast to `INT`.
- Macros can be migrated to a DuckDB persisted database. If you are using DuckDB as your catalog for DuckLake, then this will be the destination. If you are using other catalogs like PostgreSQL, SQLite or MySQL, DuckDB macros are not supported and therefore can't be migrated.
- Default values that are not literals require you to change the logic of your insertion. See the following example:

  ```
  -- Works in DuckDB, doesn't work in DuckLake
  CREATE TABLE t1 (id INTEGER, d DATE DEFAULT now());
  INSERT INTO t1 VALUES (2);

  -- Works in DuckLake and simulates the same behavior
  CREATE TABLE t1 (id INTEGER, d DATE);
  INSERT INTO t1 VALUES(2, now());
  ```

- Generated columns are the same as defaults that are not literals and therefore they need to be specified when inserting the data into the destination table. This means that the values will always be persisted (no `VIRTUAL` option).

Migration Script

The following Python script can be used to migrate from a DuckDB persisted database to DuckLake bypassing the unsupported features.

> Currently, only local migrations are supported by this script. The script will be adapted in the future to account for migrations to remote object storage such as S3 or GCS.

```
import duckdb
import argparse
import re
import os
from collections import deque

TYPE_MAPPING = {
    "VARINT": "::VARCHAR::INT",
    "UNION/ENUM": "::VARCHAR",
    "BIT": "::VARCHAR",
}


def get_postgres_secret():
    return f"""
        CREATE SECRET postgres_secret(
            TYPE postgres,
            HOST '{os.getenv("POSTGRES_HOST", "localhost")}',
            PORT {os.getenv("POSTGRES_PORT", "5432")},
            DATABASE {os.getenv("POSTGRES_DB", "migration_test")},
            USER '{os.getenv("POSTGRES_USER", "user")}',
            PASSWORD '{os.getenv("POSTGRES_PASSWORD", "simple")}'
        );"""


def _resolve_data_types(
    table: str, schema: str, catalog: str, conn: duckdb.DuckDBPyConnection
):
    excepts = []
    casts = []
    for col in conn.execute(
        f"SELECT column_name, data_type FROM information_schema.columns WHERE table_name = '{table}' AND
        table_schema = '{schema}' AND table_catalog = '{catalog}'"
    ).fetchall():
        col_name, col_type = col[0], col[1]
        # Handle mapped types
        if col_type in TYPE_MAPPING or re.match(r"(ENUM|UNION)\b", col_type):
            cast = TYPE_MAPPING.get(col_type) or TYPE_MAPPING["UNION/ENUM"]
            casts.append(f"{col_name}{cast} AS {col_name}")
            excepts.append(col_name)
        # Handle array types
        elif re.fullmatch(r"(INTEGER|VARCHAR|FLOAT)\[\d+\]", col_type):
            base_type = re.match(r"(INTEGER|VARCHAR|FLOAT)", col_type).group(1)
            cast = f"::{base_type}[]"
            casts.append(f"{col_name}{cast} AS {col_name}")
            excepts.append(col_name)
    return excepts, casts


def migrate_tables_and_views(duckdb_catalog: str, con: duckdb.DuckDBPyConnection):
    """
    Migrate tables and views from the DuckDB catalog to DuckLake using a queue system.
    If migration of a table or view fails, it will be re-added to the back of the queue.
    """
    rows = con.execute(
        f"SELECT table_catalog, table_schema, table_name, table_type "
        f"FROM information_schema.tables WHERE table_catalog = '{duckdb_catalog}'"
    ).fetchall()

    # The idea behind this queue is to retry failed migration of views due to missing dependencies.
```

```
    # The failed item is re-added to the back of the queue and waits for the rest of the dependencies to be
    migrated.
    # This avoids the need to generate a full dependency graph, which would make this script very complex.
    queue = deque(rows)
    failed_last_round = set()

    while queue:
        catalog, schema, table, table_type = queue.popleft()
        con.execute(f"CREATE SCHEMA IF NOT EXISTS {schema}")
        try:
            if table_type == "VIEW":
                view_definition = con.execute(
                    f"SELECT view_definition FROM information_schema.views "
                    f"WHERE table_name = '{table}' AND table_schema = '{schema}' AND table_catalog =
                    '{catalog}'"
                ).fetchone()[0]
                con.execute(
                    f"CREATE VIEW IF NOT EXISTS {view_definition.removeprefix('CREATE VIEW ')}"
                )
                print(f"Migrating Catalog: {catalog}, Schema: {schema}, View: {table}")
            else:
                excepts, casts = _resolve_data_types(table, schema, catalog, con)
                if casts:
                    select_clause = (
                        "* EXCLUDE(" + ", ".join(excepts) + "),\n" + ",\n".join(casts)
                    )
                    con.execute(
                        f"CREATE TABLE IF NOT EXISTS {schema}.{table} AS "
                        f"SELECT {select_clause} FROM {catalog}.{schema}.{table}"
                    )
                else:
                    con.execute(
                        f"CREATE TABLE IF NOT EXISTS {schema}.{table} AS "
                        f"SELECT * FROM {catalog}.{schema}.{table}"
                    )
                print(f"Migrating Catalog: {catalog}, Schema: {schema}, Table: {table}")
        except Exception as e:
            print(f"WARNING: Requeuing {table_type} {table}")
            # Prevent infinite loop if no progress is possible
            if (catalog, schema, table, table_type) in failed_last_round:
                print(
                    f"Skipping {table_type} {table} permanently due to repeated failure. {e}"
                )
                continue
            else:
                queue.append((catalog, schema, table, table_type))
                failed_last_round.add((catalog, schema, table, table_type))
        else:
            # Success — ensure we clear from failure set
            failed_last_round.discard((catalog, schema, table, table_type))


def migrate_macros(con: duckdb.DuckDBPyConnection, duckdb_catalog: str):
    """
    Migrate macros from the DuckDB catalog to DuckLake metadata database.
    """
    for row in con.execute(
        f"SELECT function_name, parameters, macro_definition FROM duckdb_functions() "
        f"WHERE database_name='{duckdb_catalog}'"
    ).fetchall():
        name, parameters, definition = row[0], row[1], row[2]
        print(f"Migrating Macro: {name}")
        con.execute(
            f"CREATE OR REPLACE MACRO {name}({','.join(parameters)}) AS {definition}"
```

```
        )


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Migrate DuckDB catalog to DuckLake.")
    parser.add_argument("--duckdb-catalog", required=True, help="DuckDB catalog name")
    parser.add_argument("--duckdb-file", required=True, help="Path to DuckDB file")
    parser.add_argument(
        "--ducklake-catalog", required=True, help="DuckLake catalog name"
    )
    parser.add_argument(
        "--catalog-type",
        choices=["duckdb", "postgresql", "sqlite"],
        required=True,
        help="Choose one of: duckdb, postgresql, sqlite",
    )
    parser.add_argument("--ducklake-file", required=False, help="Path to DuckLake file")
    parser.add_argument(
        "--ducklake-data-path", required=True, help="Data path for DuckLake"
    )

    args = parser.parse_args()

    con = duckdb.connect(database=args.duckdb_file)

    if args.catalog_type == "postgresql":
        con.execute(get_postgres_secret())

    secret = (
        "CREATE SECRET ducklake_secret (TYPE ducklake"
        + (
            f"\n,METADATA_PATH '{args.ducklake_file if args.catalog_type == 'duckdb' else
            f'sqlite:{args.ducklake_file}'}'"
            if args.catalog_type in ("duckdb", "sqlite")
            else "\n,METADATA_PATH ''"
        )
        + f"\n,DATA_PATH '{args.ducklake_data_path}'"
        + (
            "\n,METADATA_PARAMETERS MAP {'TYPE': 'postgres', 'SECRET': 'postgres_secret'});"
            if args.catalog_type == "postgresql"
            else ");"
        )
    )
    con.execute(secret)

    con.execute(
        f"ATTACH '{args.duckdb_file}' AS {args.duckdb_catalog};"
        f"ATTACH 'ducklake:ducklake_secret' AS {args.ducklake_catalog}; USE {args.ducklake_catalog};"
    )

    migrate_tables_and_views(
        duckdb_catalog=args.duckdb_catalog,
        con=con,
    )

    if args.catalog_type == "duckdb":
        # DETACH ducklake to be able to attach to the metadata database in migrate_macros
        con.execute(f"USE {args.duckdb_catalog}; DETACH {args.ducklake_catalog};")
        con.execute(
            f"ATTACH '{args.ducklake_file}' AS ducklake_metadata; USE ducklake_metadata;"
        )
        migrate_macros(
            con=con,
            duckdb_catalog=args.duckdb_catalog,
```

```
        )
    con.close()
```

The script can be run in any Python environment with DuckDB installed. The usage is the following:

```
usage: migration.py [-h]
    --duckdb-catalog DUCKDB_CATALOG
    --duckdb-file DUCKDB_FILE
    --ducklake-catalog DUCKLAKE_CATALOG
    --catalog-type{duckdb,postgresql,sqlite}
    [--ducklake-file DUCKLAKE_FILE]
    --ducklake-data-path DUCKLAKE_DATA_PATH
```

If you are migrating to PostgreSQL, make sure that you provide the following environment variables for the PostgreSQL secret connection:

- POSTGRES_HOST
- POSTGRES_PORT
- POSTGRES_DB
- POSTGRES_USER
- POSTGRES_PASSWORD

Guides

Backup and Recovery

DuckLake has two components: catalog and storage. The catalog contains all of DuckLake's metadata, while the storage contains all of the data files in Parquet format. The catalog is a database, while the storage layer can be any filesystem backend supported by DuckDB. These two components have different backup strategies, so this document will address them separately.

> In this document, we will focus on disasters caused by human errors or application failures/malfunctions that result in data corruption or loss.

Catalog Backup and Recovery

Backup and recovery strategies depend on the SQL database you are using as a DuckLake catalog.

> Compaction and cleanup jobs should only be done before manual backups. These operations can re-write and remove data files, effectively changing the file layout for a specific snapshot.

DuckDB Catalog

For DuckDB, the best approach is to perform regular backups of the metadata database. If the original database is corrupted, tampered with, or even deleted, you can recover from this backup.

```
-- Backup
ATTACH 'db.db' AS db (READ_ONLY);
ATTACH 'backup.db' AS backup;
COPY FROM DATABASE db TO backup;

-- Recover
ATTACH 'db.db' AS db;
ATTACH 'backup.db' AS backup (READ_ONLY);
COPY FROM DATABASE backup TO db;
ATTACH 'ducklake:db.db' AS my_ducklake;
```

It is very important to note that transactions committed to DuckLake after the metadata backup will not be tracked when recovering. The data from the transactions will exist in the data files, but the backup will point to a previous snapshot. If you are running batch jobs, make sure to always back up after the batch job. If you are regularly micro-batching or streaming data, then schedule periodic jobs to back up your metadata.

> **Tip.** If you want to make a backup with the current timestamp, you need to do this with a specific client. Right now ATTACH does not support functions, only strings. This is how it would look in Python:
>
> ```
> import duckdb
> import datetime
> con = duckdb.connection(f"backup_{datetime.datetime.now().strftime('%Y-%m-%d__%I_%M_%S')}.db")
> ```

SQLite Catalog

For SQLite, the process is exactly the same as with DuckDB and has the same implications.

```
-- Backup
ATTACH 'sqlite:db.db' AS db (READ_ONLY);
ATTACH 'sqlite:backup.db' AS backup;
COPY FROM DATABASE db TO backup;

-- Recover
ATTACH 'sqlite:db.db' AS db;
ATTACH 'sqlite:backup.db' AS backup (READ_ONLY);
COPY FROM DATABASE backup TO db;
ATTACH 'ducklake:sqlite:db.db' AS my_ducklake;
```

PostgreSQL Catalog

For PostgreSQL, there are two main approaches to backup and recovery:

- SQL dump: This approach is similar to the one mentioned for SQLite and DuckDB. This process can happen periodically and can only recover to a particular point in time (i.e., the time of the dump). For DuckLake, this will be a specific snapshot, and transactions after this snapshot will not be recorded.
- Continuous Archiving and Point-in-Time Recovery (PITR): This is a more complex approach but allows recovery to a specific point in time. For DuckLake, this means you can recover to a specific snapshot without losing any transactions.

Note that the SQL dump approach can also be managed by DuckDB using the `postgres` extension. In fact, the backup can be a DuckDB file.

```
-- Backup
ATTACH 'postgres:connection_string' AS db (READ_ONLY);
ATTACH 'duckdb:backup.db' AS backup;
COPY FROM DATABASE db TO backup;

-- Recover
ATTACH 'postgres:connection_string' AS db;
ATTACH 'duckdb:backup.db' AS backup (READ_ONLY);
COPY FROM DATABASE backup TO db;
ATTACH 'ducklake:postgres:connection_string' AS my_ducklake;
```

> Cloud-hosted PostgreSQL solutions may offer different mechanisms. We encourage you to check what your specific vendor recommends as a strategy for backup and recovery.

Storage Backup and Recovery

Backup and recovery of the data files also depend on the storage you are using. In this document, we will only focus on cloud-based object storage since it is the most common for Lakehouse architectures.

S3

In S3, there are three main mechanisms that AWS offers to back up and/or restore data:

- Cross-bucket replication
- S3 backup service
- Enable S3 versioning

Both the S3 backup service and S3 object versioning will restore data files in the same bucket. On the other hand, cross-bucket replication will copy the files to a different bucket, and therefore your DuckLake initialization should change:

```
-- Before
ATTACH 'ducklake:some.db' AS my_ducklake (DATA_PATH 's3://<og-bucket>/');

-- After
ATTACH 'ducklake:some.db' AS my_ducklake (DATA_PATH 's3://<replication-bucket>/');
```

GCS

GCS has similar mechanisms to back up and/or restore data:

- Cross-bucket replication
- Backup and DR service
- Object versioning with soft deletes enabled

Regarding cross-bucket replication, repointing to the new bucket will be necessary.

```
-- Before
ATTACH 'ducklake:some.db' AS my_ducklake (DATA_PATH 'gs://<og-bucket>/');

-- After
ATTACH 'ducklake:some.db' AS my_ducklake (DATA_PATH 'gs://<replication-bucket>/');
```

Access Control

While access control is not per se a feature of DuckLake, we can leverage the tools that DuckLake uses and their permission systems to implement schema- and table-level permissions in DuckLake.

Basic Principles

In this guide, we focus on three different roles regarding access control in DuckLake:

- The **DuckLake Superuser** can perform any DuckLake operation, most notably:
  - Initializing DuckLake (done the first time we run the `ATTACH` command).
  - Creating schemas.
  - `CREATE`, `INSERT`, `UPDATE`, `DELETE`, and `SELECT` on any DuckLake table.
- The **DuckLake Writer** can perform the following operations:
  - `ATTACH` to an existing DuckLake.
  - `CREATE`, `INSERT`, `UPDATE`, `DELETE`, and `SELECT` on any or a subset of DuckLake tables.
  - Optionally, `SELECT` on any or a subset of DuckLake tables.
  - Optionally, `CREATE` schema.
- The **DuckLake Reader** can perform the following operations:
  - `ATTACH` to an existing DuckLake. Both `READ_ONLY` and regular attaching modes will work.
  - `SELECT` on any or a subset of DuckLake tables.

> These roles are not actually implemented in DuckLake; they are constructs used in this guide, as they represent the most common types of roles present in data management systems.

DuckLake has two components: the metadata catalog, which resides in a SQL database, and the storage, which can be any filesystem backend. The roles mentioned above require different specific permissions at the **catalog level**:

- The DuckLake Superuser needs all permissions under the specified schema (`public` by default). Since this user initializes all tables, they also become the owner. Subsequent migrations between different version of the DuckLake specification must be carried out by this user.
- The DuckLake Writer only needs permissions to `INSERT`, `UPDATE`, `DELETE`, and `SELECT` at the catalog level. This is sufficient for any operation in DuckLake, including operations that expire snapshots.
- The DuckLake Reader only needs `SELECT` permissions at the catalog level.

At the storage level, we can leverage the way DuckLake structures data paths for different tables, which uses the following convention:

`/<schema>/<table>/<partition>/<data_file>.parquet`

Using this convention and the policy mechanisms of certain filesystems (such as cloud-based object storage), we can establish access to certain paths at the schema, table, or even partition level.

> This will not work if we use `ducklake_add_data_files` and the added files do not follow the path convention; permissions at the path level will not apply to these files.

The following diagram shows how these roles and their necessary permissions work in DuckLake:

DuckLake schema

Access Control with S3 and PostgreSQL

The following is an example implementation of the basic principles described above, focusing on PostgreSQL as a DuckLake catalog and S3 as the storage backend.

PostgreSQL Requirements

In this section, we create the three roles described above in PostgreSQL. We create them as users for simplicity, but you may also create them as groups if you expect a specific role to be used by multiple users.

```
-- Setup initialization user, migrations, and writing, assuming the database is already created
CREATE USER ducklake_superuser WITH PASSWORD 'simple';
GRANT CREATE ON DATABASE access_control TO ducklake_superuser;
GRANT CREATE, USAGE ON SCHEMA public TO ducklake_superuser;
GRANT CREATE, SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO ducklake_superuser;

-- Writer/reader
CREATE USER ducklake_writer WITH PASSWORD 'simple';
GRANT USAGE ON SCHEMA public TO ducklake_writer;
GRANT USAGE, SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO ducklake_writer;

-- Reader only
CREATE USER ducklake_reader WITH PASSWORD 'simple';
GRANT SELECT ON ALL TABLES IN SCHEMA public TO ducklake_reader;
```

S3 Requirements

In AWS, we create three users. The writer user will only have access to a specific schema, and the reader will only have access to a specific table. The policies needed for these users are as follows:

DuckLake Test

In this section, we connect to DuckLake using these different roles to demonstrate how the implementation works in practice using the DuckLake extension of DuckDB.

Let's initialize DuckLake and perform some basic operations with the **DuckLake Superuser**.

```
-- Using the credentials for the AWS DuckLake Superuser (other providers such as STS or SSO can also be
used)
CREATE OR REPLACE SECRET s3_ducklake_superuser (
  TYPE s3,
  PROVIDER config,
  KEY_ID '<key>',
  SECRET '<secret>',
  REGION 'eu-north-1'
```

```
);

-- Using the DuckLake Superuser credentials for Postgres
CREATE OR REPLACE SECRET postgres_secret_superuser (
  TYPE postgres,
  HOST 'localhost',
  DATABASE 'access_control',
  USER 'ducklake_superuser',
  PASSWORD 'simple'
);

-- DuckLake config secret
CREATE OR REPLACE SECRET ducklake_superuser_secret (
  TYPE ducklake,
  METADATA_PATH '',
  DATA_PATH 's3://ducklake-access-control/',
  METADATA_PARAMETERS MAP {'TYPE': 'postgres','SECRET': 'postgres_secret_superuser'}
);

-- This initializes DuckLake
ATTACH 'ducklake:ducklake_superuser_secret' AS ducklake_superuser;
USE ducklake_superuser;

-- Perform operations in DuckLake
CREATE SCHEMA IF NOT EXISTS some_schema;
CREATE TABLE IF NOT EXISTS some_schema.some_table (id INTEGER, name VARCHAR);
INSERT INTO some_schema.some_table VALUES (1, 'test');
```

Now let's use the **DuckLake Writer**:

```
-- Drop this to avoid the extension defaulting to this secret
DROP SECRET s3_ducklake_superuser;

-- Using the DuckLake Writer credentials for Postgres
CREATE OR REPLACE SECRET postgres_secret_writer (
  TYPE postgres,
  HOST 'localhost',
  DATABASE 'access_control',
  USER 'ducklake_writer',
  PASSWORD 'simple'
);

-- Using the credentials for the AWS DuckLake Writer
CREATE OR REPLACE SECRET s3_ducklake_schema_reader_writer (
  TYPE s3,
  PROVIDER config,
  KEY_ID '<key>',
  SECRET '<secret>',
  REGION 'eu-north-1'
);

-- DuckLake config secret
CREATE OR REPLACE SECRET ducklake_writer_secret (
  TYPE ducklake,
  METADATA_PATH '',
  DATA_PATH 's3://ducklake-access-control/',
  METADATA_PARAMETERS MAP {'TYPE': 'postgres','SECRET': 'postgres_secret_writer'}
);

ATTACH 'ducklake:ducklake_writer_secret' AS ducklake_writer;
USE ducklake_writer;

-- Perform operations
CREATE TABLE IF NOT EXISTS some_schema.another_table (id INTEGER, name VARCHAR);
```

```
INSERT INTO some_schema.another_table VALUES (1, 'test'); -- Works
INSERT INTO some_schema.some_table VALUES (2, 'test2'); -- Also works

-- Try to perform an unauthorized operation
CREATE TABLE other_table_in_main (id INTEGER, name VARCHAR); -- This unfortunately works
INSERT INTO other_table_in_main VALUES (1, 'test'); -- This doesn't work
```

In the last example, there are limitations to this approach. We can create an empty table, as this only inserts a new record in the metadata catalog—something the DuckLake Writer is allowed to do. The solution is to wrap table initializations in a transaction to ensure the table can't be created if there is no permission to insert data.

```
BEGIN TRANSACTION;
CREATE TABLE other_table_in_main (id INTEGER, name VARCHAR);
INSERT INTO other_table_in_main VALUES (1, 'test');
COMMIT;
```

This will throw the following error:

```
HTTP Error:
Unable to connect to URL "https://ducklake-access-control.s3.amazonaws.com/main/other_table_in_
main/ducklake-01992ec2-d9f7-745e-88e8-708e659a70be.parquet": 403 (Forbidden).

Authentication Failure - this is usually caused by invalid or missing credentials.
* No credentials are provided.
* See https://duckdb.org/docs/stable/extensions/httpfs/s3api.html
```

> The error message is the generic one used when DuckDB cannot access an object in S3; nothing specific to DuckLake.

The **DuckLake Reader** is the simplest role.

```
DROP SECRET s3_ducklake_schema_reader_writer;
CREATE OR REPLACE SECRET s3_ducklake_table_reader (
  TYPE s3,
  PROVIDER config,
  KEY_ID '<key_id>',
  SECRET '<secret_key>',
  REGION 'eu-north-1'
);
CREATE OR REPLACE SECRET postgres_secret_reader (
  TYPE postgres,
  HOST 'localhost',
  DATABASE 'access_control',
  USER 'ducklake_reader',
  PASSWORD 'simple'
);
CREATE OR REPLACE SECRET ducklake_reader_secret (
  TYPE ducklake,
  METADATA_PATH '',
  DATA_PATH 's3://ducklake-access-control/',
  METADATA_PARAMETERS MAP {'TYPE': 'postgres','SECRET': 'postgres_secret_reader'}
);
ATTACH 'ducklake:ducklake_reader_secret' AS ducklake_reader;
USE ducklake_reader;

SELECT * FROM some_schema.some_table; -- Works
SELECT * FROM some_schema.another_table; -- Fails
```

The last query will print the following error:

```
HTTP Error:
HTTP GET error on 'https://ducklake-access-control.s3.amazonaws.com/some_schema/another_
table/ducklake-019929c8-c9c9-77d7-91e6-bc3c6dc87605.parquet' (HTTP 403)
```

If we try to create a table, which is just a metadata operation, the error will be different, as it is imposed by a lack of permissions on the PostgreSQL side:

```
CREATE TABLE yet_another_table (a INT);
```

```
TransactionContext Error:
Failed to commit: Failed to commit DuckLake transaction: Failed to write new table to DuckLake: Failed to
prepare COPY "COPY "public"."ducklake_table" FROM STDIN (FORMAT BINARY)": ERROR:  permission denied for
table ducklake_table
```

Unsupported Features

This page describes what is supported in DuckDB and DuckLake in relation to DuckDB standalone (i.e., `:memory:` or DuckDB file modes). We can make a distinction between:

- What is **currently** not supported by the DuckLake specification. These are features that you are supported by DuckDB when using DuckDB's native database format but will not work with a DuckLake backend since the specification does not support them.
- What is **currently** not supported by the `ducklake` DuckDB extension. These are features that are supported by the DuckLake specification but are not (yet) implemented in the DuckDB extension.

Unsupported by the DuckLake Specification

Within this group, we are going to make a distinction between what is not supported now but is likely to be supported in the future and what is not supported and is unlikely to be supported.

Likely to be Supported in the Future

- User defined types.
- Geometry/Geospatial types
- Fixed-size arrays, i.e., ARRAY type
- ENUM type
- Variant types
- CHECK constraints. Not to be confused with Primary or Foreign Key constraint.
- Scalar and table macros (functions). However, if the catalog database supports it, there is a workaround.

```
-- Using DuckDB as a catalog, create the macro in the catalog
USE __ducklake_metadata_my_ducklake;
CREATE MACRO add_and_multiply(a, b, c) AS (a + b) * c;

-- Use the macro to create a table in DuckLake
CREATE TABLE my_ducklake.table_w_macro AS
    SELECT add_and_multiply(1, 2, 3) AS col;
```

- Default values that are not literals. See the following example:

```
-- This is allowed
CREATE TABLE t1 (id INTEGER, d DATE DEFAULT '2025-08-08');

-- This is not allowed
CREATE TABLE t1 (id INTEGER, d DATE DEFAULT now());
```

- Dropping dependencies, such as views, when calling DROP ... CASCADE. Note that this is also a DuckDB limitation.
- Generated columns

Unlikely to be Supported in the Future

- Indexes
- Primary key or enforced unique constraints and foreign key constraints are unlikely to be supported as these are constraints are prohibitively expensive to enforce in data lake setups. We may consider supporting unenforced primary keys, similar to BigQuery's implementation.
- Upserting is only supported via the `MERGE INTO` syntax since primary keys are not supported in DuckLake.
- Sequences
- `VARINT` type
- `BITSTRING` type
- `UNION` type

Unsupported by the `ducklake` DuckDB Extension

The following features are currently unsupported by the `ducklake` DuckDB extension:

- Data inlining is limited to DuckDB catalogs
- MySQL catalogs are not fully supported in the DuckDB extension
- Updates that target the same row multiple times

DuckLake Blog

Announcing DuckLake 0.1

Publication date: 2025-05-27

Authors: Mark Raasveldt and Hannes Mühleisen

TL;DR: We are announcing the DuckLake lakehouse format.

For the announcement of DuckLake 0.1, see the DuckLake Manifesto.

Updates in the DuckLake 0.2 Standard

Publication date: 2025-07-04

Author: Mark Raasveldt

TL;DR: We are releasing the updated DuckLake 0.2 standard with several new features.

We released DuckLake a little over a month ago, and we were overwhelmed by the positive response from the community. Naturally, we also received a bunch of feature requests. With DuckLake version 0.2, we are happy to present the improvements made since!

We would like to start with a quick reminder: the term DuckLake refers to both the DuckLake open standard and the `ducklake` DuckDB extension. For the updates in the latter, see the blog post on duckdb.org, which also contains a description of the changes in the standard.

New Features

Relative Schema/Table Paths

In the old DuckLake standard, paths were always only relative to the global data path. In the DuckLake v0.2 standard, the location to which data and delete files are written now have three layers:

- Data paths are relative to the **table path**
- Table paths are relative to the **schema path**
- Schema paths are relative to the **global data path**

This allows data files to be written in a more structured manner. By default, the schema and table name are set as the path to which the files are written. For example:

```
main/
    my_table/
        ducklake-<uuid1>.parquet
        ducklake-<uuid2>.parquet
```

The paths are stored in the `ducklake_table` and `ducklake_schema` tables.

By writing all files for a given table in a given subdirectory, it is now possible to use prefix-based access control at the object store level to grant users access to only specific schemas or tables in the database.

Name Mapping, and Adding Existing Parquet Files

DuckLake by default uses field ids to perform column mapping. When writing Parquet files through DuckLake, each column contains a field id that indicates to which top-level column it belongs. This allows DuckLake to perform metadata-only alter operations such as renaming and dropping fields.

When registering existing Parquet files written through other means or by other writers, these files generally do not have field identifiers written to them. In order to facilitate using these Parquet files in DuckLake, v0.2 adds a new way of mapping columns in the form of name mapping. This allows registering Parquet files as follows:

```
ATTACH 'ducklake:my_ducklake.db' AS my_ducklake;
CREATE TABLE my_ducklake.people (id BIGINT, name VARCHAR);
COPY (SELECT 42 AS id, 'Mark' AS name) TO 'people.parquet';
CALL ducklake_add_data_files('my_ducklake', 'people', 'people.parquet');

FROM my_ducklake.people;
```

id int64	name varchar
42	Mark

Every file that is added to DuckLake has an optional `mapping_id`, that tells the system the mapping of column name to field id. As this is done on a per-file basis, Parquet files can always be added to DuckLake without restriction. All DuckLake operations are supported on added files, including schema evolution and data change feeds.

Settings

DuckLake v0.2 adds support for scoped settings. The settings stored in the `ducklake_metadata` table now have an optional scope field, together with a `scope_id`. This allows settings to be scoped at either the schema or table level – instead of requiring all settings to be scoped globally.

Partition Transforms

This release adds support for the year/month/day/hour partition transforms, allowing these functions to be used to partition data directly instead of having to create separate columns containing these fields.

Migration Guide

You can migrate from a DuckLake that runs v0.1 to v0.2 using the following SQL queries. The DuckDB `ducklake` extension automatically runs the following SQL queries to perform the migration:

```
ALTER TABLE ducklake_schema ADD COLUMN path VARCHAR DEFAULT '';
ALTER TABLE ducklake_schema ADD COLUMN path_is_relative BOOLEAN DEFAULT true;
ALTER TABLE ducklake_table ADD COLUMN path VARCHAR DEFAULT '';
ALTER TABLE ducklake_table ADD COLUMN path_is_relative BOOLEAN DEFAULT true;
ALTER TABLE ducklake_metadata ADD COLUMN scope VARCHAR;
ALTER TABLE ducklake_metadata ADD COLUMN scope_id BIGINT;
ALTER TABLE ducklake_data_file ADD COLUMN mapping_id BIGINT;
CREATE TABLE ducklake_column_mapping (mapping_id BIGINT, table_id BIGINT, type VARCHAR);
CREATE TABLE ducklake_name_mapping (mapping_id BIGINT, column_id BIGINT, source_name VARCHAR, target_field_
id BIGINT, parent_column BIGINT);
UPDATE ducklake_partition_column
SET column_id = (SELECT list(column_id ORDER BY column_order)
FROM ducklake_column
WHERE table_id = ducklake_partition_column.table_id AND parent_column IS NULL AND end_snapshot IS
NULL)[ducklake_partition_column.column_id + 1];
UPDATE ducklake_metadata
SET value = '0.2'
WHERE key = 'version';
```

DuckLake 0.3 with Iceberg Interoperability and Geometry Support

Publication date: 2025-09-17

Authors: Guillermo Sanchez, Gabor Szarnyas

TL;DR: We are releasing version 0.3 of the DuckLake Specification and the `ducklake` DuckDB extension.

The focus of the team in the last two months has been on making DuckLake as robust as possible. Besides this, we have implemented a bunch of features that will make the experience of running DuckLake even smoother. In this post, we will discuss both the updates in the DuckDB `ducklake` extension and in the DuckLake v0.3 specification.

New Features in the `ducklake` Extension

The `ducklake` extension in DuckDB v1.4.0 ships several new features. To try out the features described in this blog post, make sure that you have DuckDB v1.4.0 installed and all of your extensions are up-to-date. This can be achieved by running:

```
UPDATE EXTENSIONS;
```

Interoperability with Iceberg

Powered by the DuckDB `iceberg` extension, it is possible to copy data between DuckLake to Iceberg.

Now you can **copy from Iceberg to DuckLake.**

```
CREATE SCHEMA iceberg_datalake.default;
CREATE TABLE iceberg_datalake.default.iceberg_table AS
    SELECT a FROM range(4) t(a);
COPY FROM DATABASE iceberg_datalake TO my_ducklake;
```

Copying from DuckLake to Iceberg also works, given that the schemas are already created in Iceberg.

```
-- Assuming Iceberg catalog is empty since the COPY command does
-- not replace tables
CREATE SCHEMA iceberg_datalake.main;
CREATE TABLE my_ducklake.default.ducklake_table AS
    SELECT a FROM range(4) t(a);
COPY FROM DATABASE my_ducklake TO iceberg_datalake;
```

These examples are data copies (i.e., deep copies) of the latest snapshot, which means that only data is ported from Iceberg to DuckLake and vice versa. Metadata-only copies are also supported from Iceberg to DuckLake. The main difference is that metadata only copies do not copy over the underlying data, only the metadata, including all the snapshot history. This means that you can query previous snapshots of an Iceberg table as if it was a DuckLake table.

```
CREATE SCHEMA iceberg_datalake.default;
CREATE TABLE iceberg_datalake.default.iceberg_table AS
    SELECT a FROM range(4) t(a); -- version 0
INSERT INTO iceberg_datalake.default.iceberg_table
    SELECT a FROM range(4, 10) t(a); -- version 1
CALL iceberg_to_ducklake('iceberg_datalake', 'ducklake');
```

Now you can query any version of the iceberg table as if it was a DuckLake table.

```
FROM my_ducklake.default.iceberg_table AT (VERSION => 0);
```

a
int64
0
1
2
3

MERGE INTO Statement

Since DuckDB 1.4 supports `MERGE INTO` this functionality can also be used in the `ducklake` extension since this release. This is a very common statement in OLAP systems that do not support primary keys but still want to support upserting (i.e., `UPDATE` plus `INSERT`) functionality.

```
CREATE TABLE Stock(item_id INTEGER, balance INTEGER);
INSERT INTO Stock VALUES (10, 2200), (20, 1900);

WITH new_stocks(item_id, volume) AS (VALUES (20, 2200), (30, 1900))
    MERGE INTO Stock USING new_stocks USING (item_id)
    WHEN MATCHED THEN UPDATE SET balance = balance + volume
    WHEN NOT MATCHED THEN INSERT VALUES (new_stocks.item_id, new_stocks.volume);
FROM Stock;
```

item_id	balance
int32	int32
10	2200
20	4100
30	1900

`MERGE INTO` also supports more complex conditions and `DELETE` statements.

```
WITH deletes(item_id, delete_threshold) AS (VALUES (10, 3000))
    MERGE INTO Stock USING deletes USING (item_id)
    WHEN MATCHED AND balance < delete_threshold THEN DELETE;
FROM Stock;
```

item_id	balance
int32	int32
20	4100
30	1900

CHECKPOINT Statement

DuckLake now also supports the `CHECKPOINT` statement. In DuckLake, this statement runs a series of maintenance functions in a sequential order. This includes flushing inlined data, expiring snapshots, compacting files, and rewriting heavily deleted files as well as cleaning up old or orphaned files. The `CHECKPOINT` statement can be configured via some global options that can be set via the `my_ducklake.set_option` function.

```
ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;
USE my_ducklake;
CHECKPOINT;
```

Faster Inserts

In cases where the output of each thread in a certain query is of a desired size (i.e., not too small), setting the option `per_thread_output` will most likely speed up the insertion process in DuckLake. If you have a large number of threads and the output of each thread is not very significant, using `per_thread_output` will likely not improve the performance that much and will generate an undesirable amount of small files that will hinder read performance. You can benefit a lot from this option if you are running a setup where DuckDB is in an EC2 instance and has a very high network bandwidth (up to 100 Gbps) to write to S3.

In the following benchmark we did find ~25% improvement when enabling `per_thread_output`:

```
.timer on
ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;
CREATE TABLE sample_table AS SELECT * FROM range(1_000_000_000);
-- 4.5 seconds
CREATE TABLE slow_copy AS SELECT * FROM sample_table;

-- enable the option
CALL my_ducklake.set_option('per_thread_output', true);

-- 3.4 seconds
CREATE TABLE fast_copy AS SELECT * FROM sample_table;
```

This feature was contributed by community member Julian Meyers.

Updates in the DuckLake Specification

Support for Geometry Types

Geometry types are now supported in DuckLake. This means that you can now use most of the functionality in the `spatial` extension of DuckDB with DuckLake as a backend. See the documentation on supported geometry primitives.

```
INSTALL spatial;
LOAD spatial;

ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;
CREATE TABLE geometry_table (polygons GEOMETRY);
INSERT INTO geometry_table VALUES ('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))');
SELECT
    polygons,
    ST_Area(polygons) AS area
FROM geometry_table;
```

polygons geometry	area double
POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))	1.0

Please note that the geometry support is currently experimental, and it's missing features such as filter pushdown, data inlining and coordinate system tracking.

Author Commits

We have also enabled some functionality to be able to author commits whenever you execute a new transaction in DuckLake. This functionality helps greatly with audit needs.

Migration from v0.2 to v0.3

You don't have to worry about manually migrating between different versions of the DuckLake specification. The moment you attach to your existing DuckLake with the new `ducklake` extension installed, the extension will update the metadata catalog to the 0.3 version using the following SQL script.

Guides and Roadmap

In the last month, we have also been preparing some guides to help you adopt DuckLake and cover some topics that are not features of DuckLake but will help you run your DuckLake smoothly.

- Migrations between different formats
- Backups and recovery
- Access control

Finally, we have also released a roadmap showing the features planned in upcoming DuckLake versions.

Acknowledgments

This document is built with Pandoc using the Eisvogel template. The scripts to build the document are available in the DuckDB-Web repository.

The emojis used in this document are provided by Twemoji under the CC-BY 4.0 license.

The syntax highlighter uses the Bluloco Light theme by Umut Topuzoğlu.