

Managing Data with Delta Lake

Data lakehouses uses specialized storage frameworks to enhance the functionality of traditional Data Lakes. **Delta Lake** is one such framework of Databricks that powers the **Databricks Lakehouse Platform**.

Introducing Delta Lake

Traditional data lakes often suffer from inefficiencies and encounter various challenges in processing big data. Delta Lake technology is an innovative solution designed to operate on top of data lakes to overcome these issues.

What is Delta Lake?

Delta Lake is an Op-Src storage layer that brings reliability to data lakes by adding a transactional storage layer on top of data stored in cloud storage.

In the context of data lakehouses, a storage layer refers to the framework responsible for managing and organizing data stored within the data lake. It serves as an intermediary platform through which data is ingested, queried, and processed. Delta Lake is not a storage medium or format like Parquet or JSON but Delta Lake runs on top of such data formats to provide a robust solution that overcomes the challenges of data lakes.

While data lakes are excellent solutions for storing massive volumes of diverse data, they often encounter several challenges related to data inconsistency and performance issues. The primary factor behind these limitations is the absence of ACID transaction support in a data lake. ACID stands for atomicity, consistency, isolation, and durability, and represents fundamental rules that ensure operations on data are reliably executed, as in traditional databases. This absence led to issues such as partially committed data and corrupted files, ultimately affecting the overall reliability of the data stored in the lake.

What makes Delta Lake an innovative solution is its ability to overcome such challenges posed by traditional data lakes. Delta Lake provides ACID transaction guarantees for data manipulation operations in the lake. It offers transactional capabilities that enable performing data operations in an atomic and consistent manner. This ensures that there is no partially committed data; either all operations within a transaction are completed successfully or none of them is. These capabilities allow you to build reliable data lakes that ensure data integrity, consistency, and durability.

Delta Lake is optimized for cloud object storage.

Delta Lake Transaction Log

The Delta Lake library is deployed on the cluster as part of the Databricks Runtime. When you create a Delta Lake table within this ecosystem, the data is stored on the cloud storage

in one or more data files in Parquet format. However, alongside these data files, Delta Lake creates a transaction log in JSON format.

The Delta Lake transaction log, often referred to as the Delta Log, is an ordered record of every transaction performed on the table since its creation. As a result, it functions as the source of truth for the table's state and history. So every time you query the table, Spark checks this transaction log to determine the most recent version of the data.

Each committed transaction is recorded in a JSON file. This file contains essential details about the operations performed, such as its type (insert, update, ..., etc.) and any predicate used during these operations, including conditions and filters. Beyond simply tracking the operations executed, the log captures the names of all data files affected by these operations.

Understanding Delta Lake Functionality

Let's see the scenarios of two users, Alice (a data producer) and Bob (a data consumer) where they interact through delta lake table. Their interaction on table can be described in 4 key scenarios: data writing and reading, data updating, concurrent writes and reads, and, lastly, failed write attempts.

Writing and Reading scenario

Alice starts this scenario by creating the Delta Table and populating the data. The delta module stores the table, for ex: in two data files (part1.parquet, part2.parquet) and saves them in Parquet format within the table directory on the storage. Upon the completion of writing the data files, the Delta module adds a transaction log, labeled as 000.json, into the *delta_log* subdirectory. This log captures metadata information about the changes made to the delta table. Includes operation type, the name of the newly created data files, transaction timestamp & other relevant info.

Bob reads the Delta table through SQL. However, before directly accessing the data files, the delta module always begins by consulting the transaction log associated with the table. (*delta_log/000.json* first!). The delta module proceeds by reading these two data files and returning the results to Bob.

Delta Lake follows a structured approach for managing and processing the data in the lake. It always uses the transaction log as a point of reference to interact with the data files of Delta Lake tables.

Updating Scenario

In this, Alice makes an update to a record in *part_1.parquet* of the Delta table. However, since parquet files are immutable, delta lake takes a different approach to updates. Instead of directly modifying the record within the existing file, delta module makes a copy of the data from the original file and applies the updates in the new data file, *part_3.parquet*. It then

updates the log by writing a new transaction record (`001.json`). The new log file is now aware that the data file `part1.parquet` is no longer relevant to the current state of the table.

When Bob attempts to read data from the table, the Delta module first consults the transaction log to determine the valid files for the current table version. In this instance, the log indicates that only the parquet files *part 2* and *part 3* are included in the latest version of the table. As a result, the Delta module confidently reads data from these two files and ignores the outdated file `part 1.parquet`.

So, Delta Lake follows the principle of immutability; once a file is written to the storage layer, it remains unchanged. The approach of handling updates through file copying and transaction log management ensures that the historical versions of data are preserved. This offers a comprehensive record of all modifications performed on the table.

Concurrent writes and reads scenario

In this scenario, Alice and Bob are both interacting with the table simultaneously. Alice is inserting new data, initiating the creation of a new data file, `part 4.parquet`. Meanwhile, Bob is querying the table, where the Delta module starts by reading the transaction log to determine which Parquet files contain the relevant data.

At the time Bob executes the query, the transaction log includes information about the Parquet files *part 2* and *part 3* only, as the file `part 4.parquet` is not fully written yet. So, Bob's query reads the two latest files available that represent the current table state at that moment. Using this methodology, Delta Lake guarantees that you will always get the most recent version of the data. Your read operations will never have a deadlock state or conflicts with any ongoing operation on the table. Finally, once Alice's query finishes writing the new data, the Delta module adds a new JSON file to the transaction log, named `002.json`.

In summary, Delta Lake's transaction log helps avoid conflicts between write and read operations on the table. So, even when write and read operations are occurring simultaneously, read operations can proceed without waiting for the writes to complete. This capability helps maintain the reliability and performance of data operations on Delta Lake tables.

Failed writes scenario

Imagine that Alice attempts again to insert new data into the Delta table. The Delta module begins writing the new data to the lake in a new file, `part 5.parquet`. However, an unexpected error occurs during this operation, resulting in the creation of an incomplete file. This failure prevents the Delta module from recording any information related to this incomplete file in the transaction log.

Now, when Bob queries the table, the Delta module starts, as usual, by reading the transaction log. Since there is no information about the incomplete file `part 5.parquet` in the log, only the parquet files *part 2*, *part 3*, and *part 4* will be considered for the query output.

Consequently, Bob's query is protected from accessing the incomplete or dirty data created by Alice's unsuccessful write operation.

In essence, Delta Lake guarantees the prevention of reading incomplete or inconsistent data. The transaction log serves as a reliable record of committed operations on the table. And in the event of a failed write, the absence of corresponding information in the log ensures that subsequent queries won't be affected by incomplete data.

Delta Lake Advantages

Delta Lake's strength arises from its robust transaction log, which serves as the backbone of this innovative solution. This log empowers Delta Lake to deliver a range of features and advantages that can be summarized by the following key points:

Enabling ACID transactions

The main advantage of the transaction log is that it enables Delta Lake to execute ACID transactions on traditional data lakes. This feature helps maintain data integrity and consistency when performing data operations, ensuring that they are processed reliably and efficiently.

Scalable metadata handling

Another primary benefit of Delta Lake is the ability to handle table metadata efficiently. The table metadata, which represents information about the structure, organization, and properties of the table, is stored in the transaction log instead of a centralized metastore. This strategy enhances query performance when it comes to listing large directories and reading vast amounts of data. It also includes table statistics to accelerate operations.

Full audit logging

Additionally, the transaction log serves as a comprehensive audit trail that captures every change occurring on the table. It tracks all modifications, additions, and deletions made to the data, along with the timestamps and user information associated with each operation. This allows you to trace the evolution of the data over time, which facilitates troubleshooting issues and ensures data governance.

Working with Delta Lake Tables

Exploring Delta Time Travel

Time travel is a feature in Delta Lake that allows you to retrieve previous versions of data in Delta Lake tables. The key aspect of Delta Lake time travel is the automatic versioning of the table. This versioning provides an audit trail of all the changes that have happened on the table. Whenever a change is made to the data, Delta Lake captures and stores this change as a new version. Each version represents the state of the table at a specific point in time.

To explore the historical versions of a Delta table, you can leverage the `DESCRIBE HISTORY` command in SQL. This command provides a detailed log of all the operations performed on the table, including information such as the timestamp of the operation, the type of operation (insert, update, delete, etc.), and any additional metadata associated with the change.

Querying by timestamp

The first method allows you to retrieve the table's state as it existed at a specific point in time. This involves specifying the desired timestamp in the `SELECT` statement using the `TIMESTAMP AS OF` keyword:

```
SELECT * FROM <table_name> TIMESTAMP AS OF <timestamp>
```

Querying by version number

The second method involves using the version number associated with each operation on the table

```
SELECT * FROM <table_name> VERSION AS OF <version>
```

Alternatively, you can use its short syntax represented by `@v` followed by the version number:

```
SELECT * FROM <table_name> @v<version>
```

So, Delta Lake's time travel enables you to independently investigate different versions of the data without impacting the current state of the table. This feature is possible thanks to those extra data files that had been marked as removed in our transaction log.

Rolling Back to Previous Versions

Delta Lake time travel is particularly useful in scenarios where undesired data changes need to be rolled back to a previous state.

Delta Lake offers the `RESTORE TABLE` command that allows you to roll back the table to a specific timestamp or version number:

```
RESTORE TABLE <table_name> TO TIMESTAMP AS OF <time_stamp>
```

```
RESTORE TABLE <table_name> TO VERSION AS OF <version>
```

Optimizing Delta Lake Tables

Delta Lake provides an advanced feature for optimizing table performance through compacting small data files into larger ones. This optimization is particularly significant as it enhances the speed of read queries from a Delta Lake table. You trigger compaction by executing the `OPTIMIZE` command:

```
OPTIMIZE <table_name>
```

Say you have a table that has accumulated many small files due to frequent write operations. By running the `OPTIMIZE` command, these small files can be compacted into one or more larger files.

Table optimization improves the overall performance of the table by minimizing overhead associated with file management and enhancing the efficiency of data retrieval operations.

Z-Order Indexing

A notable extension of the `OPTIMIZE` command is the ability to leverage Z-Order indexing. Z-Order indexing involves the reorganization and co-location of column information within the same set of files. To perform Z-Order indexing, you simply add the `ZORDER BY` keyword to the `OPTIMIZE` command. This should be followed by specifying one or more column names on which the indexing will be applied:

```
OPTIMIZE <table_name>
ZORDER BY <column_names>
```

Let's consider the data files containing a numerical column such as `ID` that ranges between 1 and 100. Applying Z-Order indexing to this column during the optimization process results in different content written to the two compacted files. In this case, Z-Order indexing will aim to have the first compacted file contain values ranging from 1 to 50, while the subsequent file contains values from 51 to 100.

This strategic arrangement of data enables data skipping in Delta Lake, which helps avoid unnecessary file scans during query processing.

Vacuuming

Delta Lake's vacuuming provides an efficient mechanism for managing unused data files within a Delta table. As data evolves over time, there might be scenarios where certain files become obsolete, either due to uncommitted changes or because they are no longer part of the latest state of the table. The `VACUUM` command in Delta Lake enables you to clean up these unwanted files, ensuring efficient storage management that saves storage space and cost.

```
VACCUM <table_name> [RETAIN num HOURS]
```

The process involves specifying a retention period threshold for the files, so the command will automatically remove all files older than this threshold. The default retention period is set to seven days, meaning that the vacuum operation will prevent you from deleting files less than seven days old.

It's important to note that running the `VACUUM` command comes with a trade-off. Once the operation is executed and files older than the specified retention period are deleted, you lose the ability to time travel back to a version older than that period. This is because the associated data files are no longer available. Therefore, it is crucial to carefully consider the retention period based on your data retention policies and data storage requirements.

Dropping Delta Lake Tables

```
DROP table <table_name>
```

Upon executing this command, the table, along with its data, will be deleted from the lakehouse environment. To confirm this action, you can attempt to query the table again, only to find that it is no longer found in the database.