O'REILLY®

Delta Lake The Definitive Guide

Modern Data Lakehouse Architectures with Delta Lake

Early Release
Raw & Unedited

Sponsored by databricks

Denny Lee,
Tathagata Das &
Vini Jaiswal

databricks

One unified platform for data and AI

Combine data warehouse performance with data lake flexibility

The complexity of maintaining both data lakes and data warehouses creates data silos, higher costs and slower decision-making.

The Databricks platform — built on lakehouse architecture — brings data warehouse quality and reliability to open, flexible data lakes. This simplified architecture provides one environment for analytics, streaming data, data science and machine learning to help you make the most of your data.

Learn more at databricks.com/lakehouse

Delta Lake: The Definitive Guide

Modern Data Lakehouse Architectures with Delta Lake

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Denny Lee, Tathagata Das, and Vini Jaiswal

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Delta Lake: The Definitive Guide
by Denny Lee, Tathagata Das, and Vini Jaiswal

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman
Development Editor: Gary O'Brien
Production Editor: Christopher Faucher

Interior Designer: David Futato
Cover Designer: Karen Montgomery

April 2022: First Edition

Revision History for the Early Release
2021-04-20: First Release
2021-05-07: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098104597* for release details.

978-1-098-10452-8

Table of Contents

CHAPTER 1
Basic Operations on Delta Lakes

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

In the previous chapter, we discussed the progression of databases to data lakes, the importance of data reliability, and how to easily and effectively use Apache Spark™ to build scalable and performant data processing pipelines. However, when building these data processing pipelines expressing the processing logic only solves half of the end-to-end problem of building a pipeline. For a data practitioner, the ultimate goal for building pipelines is to efficiently query the processed data and get insights from it. For a query to be efficient, the processed data must be saved in a storage format and system such that the query engine can efficiently read the necessary (preferably minimal) data to compute the query result. In addition, data is almost never static - it has to be continuously added, updated, corrected, and deleted when needed. It is important that the storage system and the processing engine can work together to provide *atomic transactionality* on the quality of the final result despite all the (possibly concurrent) read and write operations on the data.

Concisely, Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark™ and big data workloads. With Delta Lake providing *atomic transactionality* for your data, this allows you - the data practitioner - to need to only focus on building your data processing pipelines by expressing the processing.

What is Delta Lake?

As previously noted, over time, there have been different storage solutions built to solve this problem of data quality - from databases to data lakes. The transition from databases to data lakes had the benefit of decoupling business logic from storage as well as the ability to independently scale compute and storage. But lost in this transition was ensuring data reliability. Providing data reliability to data lakes led to the development of Delta Lake.

Built by the original creators of Apache Spark, Delta Lake was designed to combine the best of both worlds for online analytical workloads (i.e., OLAP style): *the transactional reliability of databases with the horizontal scalability of data lakes*.

Delta Lake is a file-based, open-source storage format that provides ACID transactions, scalable metadata handling, and unifies streaming and batch data processing. It runs on top of your existing data lakes and is compatible with Apache Spark and other processing engines. Specifically, it provides the following features:

ACID guarantees
: Delta Lake ensures that all data changes written to storage are committed for durability and made visible to readers atomically. In other words, no more partial or corrupted files! We will discuss more on the acid guarantees as part of the transaction log later in this chapter.

Scalable data and metadata handling:
: Since Delta Lake is built on data lakes, all reads and writes using Spark or other distributed processing engines are inherently scalable to petabyte-scale. However, unlike most other storage formats and query engines, Delta Lake leverages Spark to scale out all the metadata processing, thus efficiently handling metadata of billions of files for petabyte-scale tables. We will discuss more on the transaction log later in this chapter.

Audit History and Time travel
: The Delta Lake transaction log records details about every change made to data providing a full audit trail of the changes. These data snapshots enable developers to access and revert to earlier versions of data for audits, rollbacks, or to reproduce experiments. We will dive further into this topic in Chapter 3: Time Travel with Delta.

Schema enforcement and schema evolution
: Delta Lake automatically prevents the insertion of data with an incorrect schema, i.e. not matching the table schema. And when needed, it allows the table schema to be explicitly and safely evolved to accommodate ever-change data. We will dive further into this topic in Chapter 4 focusing on schema enforcement and evolution.

Support for deletes updates, and merge
: Most distributed processing frameworks do not support atomic data modification operations on data lakes. Delta Lake supports merge, update, and delete operations to enable complex use cases including but not limited to change-data-capture (CDC), slowly-changing-dimension (SCD) operations, and streaming upserts. We will dive further into this topic in Chapter 5: Data modifications in Delta.

Streaming and batch unification
: A Delta Lake table has the ability to work both in batch and as a streaming source and sink. The ability to work across a wide variety of latencies ranging from streaming data ingest to batch historic backfill to interactive queries all just work out of the box. We will dive further into this topic in Chapter 6: Streaming Applications with Delta.

For more information, refer to the VLDB20 paper: Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores.

How to start using Delta Lake

Delta Lake is well integrated within the Apache Spark™ ecosystem so the easiest way to start working with Delta Lake is to use it with Apache Spark. In this section, we will focus on three of the most straightforward mechanisms:

- Using Delta Lake via local Spark shells
- Leveraging GitHub or Maven
- Using Databricks Community Edition

Using Delta Lake via local Spark shells

The Delta Lake package is available as with the `--packages` option when working with Apache Spark shells. This option is the simplest option when working with a local version of Spark

```
# Using Spark Packages with PySpark shell
./bin/pyspark --packages io.delta:delta-core_2.12:1.0
# Using Spark Packages with spark-shell
```

```
./bin/spark-shell --packages io.delta:delta-core_2.12:1.0
# Using Spark Packages with spark-submit
./bin/spark-submit --packages io.delta:delta-core_2.12:1.0
# Using Spark Packages with spark-shell
./bin/spark-sql --packages io.delta:delta-core_2.12:1.0
```

Leveraging GitHub or Maven

In this book, we will not focus on building standalone applications with Apache Spark. For a good tutorial, please refer to Chapter 2 in Learning Spark 2nd Edition or building your own self-contained application.

As you build them, you can access Delta Lake via:

- GitHub: https://github.com/delta-io/delta
- Maven: https://mvnrepository.com/artifact/io.delta/delta-core

Using Databricks Community Edition

Aside from using your local machine or building your own self-contained application, you can try all of our examples in this book on the Databricks Community Edition for free (Fig. 2-15). As a learning tool for Apache Spark, the Community Edition has many tutorials and examples worthy of note. As well as writing your own notebooks in Python, R, Scala, or SQL, you can also import other notebooks, including Jupyter notebooks.

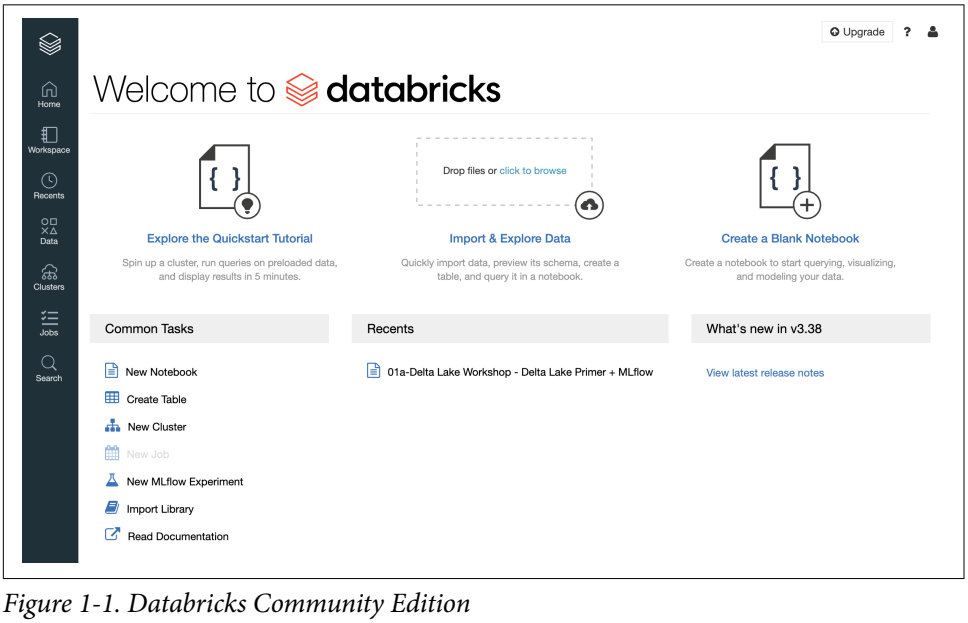


Figure 1-1. Databricks Community Edition

To get an account, go to *https://databricks.com/try*, and follow the instructions to use the Community Edition for free.

Basic operations

In this section, we will cover the basic operations of building Delta tables. As of writing this, Delta has full APIs in three languages commonly used in the big data ecosystem: SQL, Scala, and Python. It can store data in file systems like HDFS and cloud object stores including S3 and ADLS gen2. It is designed to be written primarily by Spark applications and can be read by many open-source data engines like Spark SQL, Hive, Presto (or Trino), Ballista (Rust), and several enterprise products like AWS Athena, Azure Synapse, BigQuery, and Dremio.

Creating your first Delta table

Let's create our first Delta table! Like databases, to create our Delta table we can first create a table definition and define the schema or, like data lakes simply write a Spark DataFrame to storage in the Delta format.

Writing your Delta table

When creating a Delta table, you are writing files to some storage (e.g. file system, cloud object stores). All of the files together stored in a directory of a particular structure (more on this later) make up your table. Therefore, when we create a Delta table, we are in fact writing files to some storage location. For example, the following Spark code snippet takes an existing Spark DataFrame and writes it in the Apache Parquet storage format in the folder location `/data` using the Spark DataFrame API.

```
dataframe.write.format("parquet").save("/data")
```

For more information on the Spark DataFrame API, a good reference is Learning Spark 2nd Edition.

With one simple modification of the preceding code snippet, you can now create a Delta table.

Figure 1-2. Instead of parquet, simply say delta

Let's see how that looks in code by using Apache Spark to create a Delta table from a DataFrame. In the following code example, we will first create the Spark DataFrame data and then use the write method to save the table to storage.

```
%python
# Create data DataFrame
data = spark.range(0, 5)
# Write the data DataFrame to /delta location
data.write.format("delta").save("/delta")

%scala
// Create data DataFrame
val data = spark.range(0, 5)
// Write the data DataFrame to /delta location
data.write.format("delta").save("/delta")
```

It is important to note that in most production environments when you are working with large amounts of data, it is important to partition your data. The following example partitions your data by date (a commonly followed best practice):

```
%python
# Write the Spark DataFrame to Delta table partitioned by date
data.write.partitionBy("date").format("delta").save("/delta")

%scala
// Write the Spark DataFrame to Delta table partitioned by date
data.write.partitionBy("date").format("delta").save("/delta")
```

If you already have an existing Delta table and would like to append or overwrite data to your table, include the `mode` method in your statement.

```
%python
>## Append new data to your Delta table
data.write.format("delta").mode("append").save("/delta")

# Overwrite your Delta table
data.write.format("delta").mode("overwrite").save("/delta")

%scala
// Append new data to your Delta table
```

```
data.write.format("delta").mode("append").save("/delta")

// Overwrite your Delta table
data.write.format("delta").mode("overwrite").save("/delta")
```

Reading your Delta table

Similar to writing your Delta table, you can use the DataFrame API to read the same files from your Delta table.

```
%python
# Read the data DataFrame from the /delta location
spark.read.format("delta").load("/delta").show()

%scala
// Read the data DataFrame from the /delta location
spark.read.format("delta").load("/delta").show()
```

You can also read the table using SQL by specifying the file location after specifying `delta`.

```
%sql
SELECT * FROM delta.`/delta`
```

The output of this table can be seen below.

```
+---+
| id|
+---+
|  4|
|  2|
|  3|
|  0|
|  1|
+---+
```

Reading your metastore defined Delta table

In the previous sections, we have been reading our Delta tables directly from the file system. But how would you read a metastore defined Delta table? For example, instead of reading our Delta table in SQL using the file path:

```
%sql
SELECT * FROM delta.`/delta`
```

how would you read using a metastore-defined table such as:

```
%sql
SELECT * FROM myTable
```

To do this, you would need to first define the table within the metastore using the `saveAsTable` method or a `CREATE TABLE` statement.

```
%python
# Write the data DataFrame to the metastore defined as myTable
data.write.format("delta").saveAsTable("myTable")

%scala
// Write the data DataFrame to the metastore defined as myTable
data.write.format("delta").saveAsTable("myTable")
```

Note, when using the `saveAsTable` method, you will save the Delta table files into a location managed by the metastore (e.g. `/user/hive/warehouse/myTable`). If you want to use SQL or control the location of your Delta table, then you initially save it using the `save` method where you specify the location (e.g. `/delta`) and then create the table using the following SQL statements.

```
%sql
-- Create a table in the metastore
CREATE TABLE myTable (
  id INTEGER)
USING DELTA
LOCATION "/delta"
```

As noted previously, it will be important to partition your large tables.

```
%sql
CREATE TABLE id (
  date DATE
  id INTEGER)
USING DELTA
PARTITION BY date
LOCATION "/delta"
```

Note, the `LOCATION` property that points to the underlying files that make up your Delta table.

Side Note: What is a metastore?

Generally in the Hadoop ecosystem, to reference a dataset via the flavors of SQL (Spark SQL, Hive, Presto, Impala, etc), you need to make a table definition in a **metastore** (commonly this is a Hive metastore). This table definition is a metadata entry that will describe to data processing frameworks such as Apache Spark the data location, storage format, table schema, as well as other properties.

Unpacking the Transaction Log

In the previous section where we discuss the basic operations of Delta Lake, many readers may have been under the impression that there was more to ensuring data reliability from a *usability* or *user experience* perspective. The basic operations of Delta Lake appear to be simply Spark SQL statements. We have alluded to improved

functionality such as time travel (chapter 3) and DML operations (chapter 5) but this may not seem apparent from those basic operations. But the real benefit of Delta Lake is that the operations themselves provide ACID transactional protection. And to understand how Delta Lake provides those protections, we need to first unpack and understand the transaction log.

For example, when running this notebook in Databricks Community Edition, note that there is little difference between the Parquet and Delta tables generated from the same source.

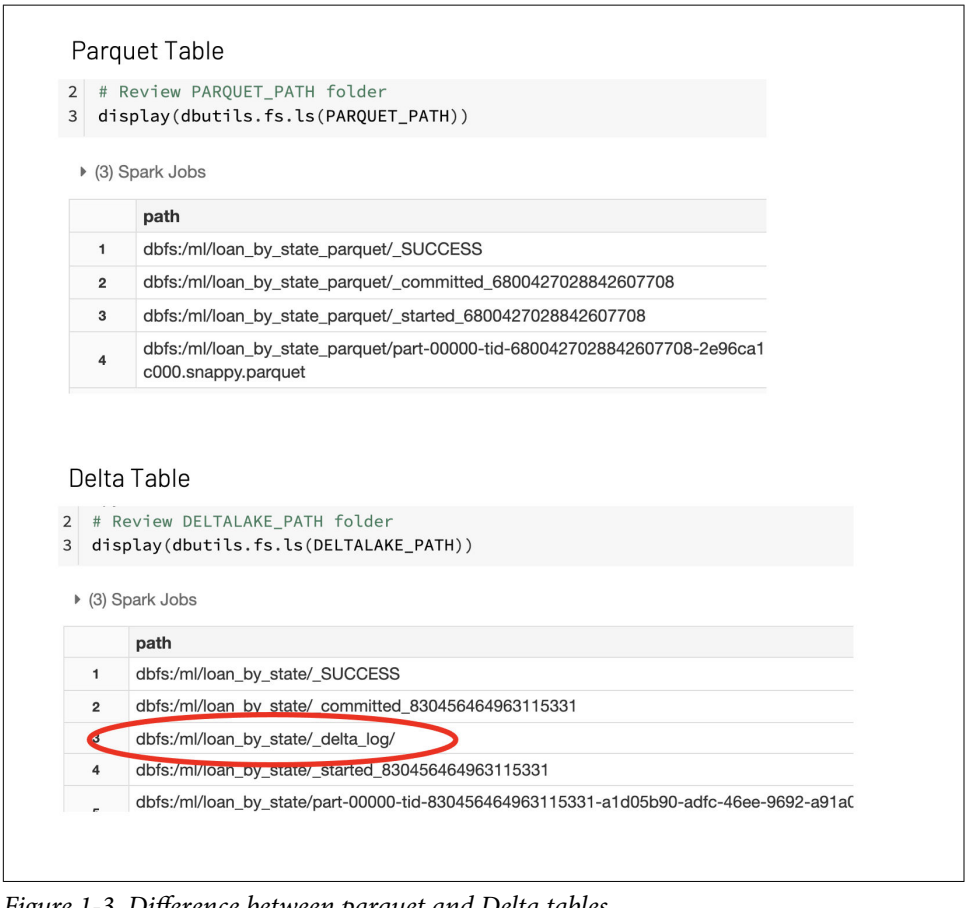


Figure 1-3. Difference between parquet and Delta tables

The key difference is the `_delta_log` folder which is the Delta transaction log. This transaction log is key to understanding Delta Lake because it is the underlying infrastructure for many of its most important features including but not limited to ACID transactions, scalable metadata handling, and time travel. In this section, we'll explore

what the Delta Lake transaction log is, how it works at the file level, and how it offers an elegant solution to the problem of multiple concurrent reads and writes.

What Is the Delta Lake Transaction Log?

The Delta Lake transaction log (also known as the `Delta Log`) is an ordered record of every change that has ever been performed on a Delta Lake table since its inception.

Single Source of Truth

Delta Lake is built on top of Apache Spark™ in order to allow multiple readers and writers of a given table to all work on the table at the same time. To show users correct views of the data at all times, the Delta Lake transaction log serves as a single source of truth – the central repository that tracks all changes that users make to the table. You could compare this transaction log feature in Delta Lake as a single source of truth to the .git directory in a git-managed source code repository.

The concept of a single source of truth is important because, over time, processing jobs will fail in your data lake. The result is that processing jobs or tasks will leave partial files that do not get cleaned up. Subsequent processing or queries will not be able to ascertain which files should or should not be included in their queries.

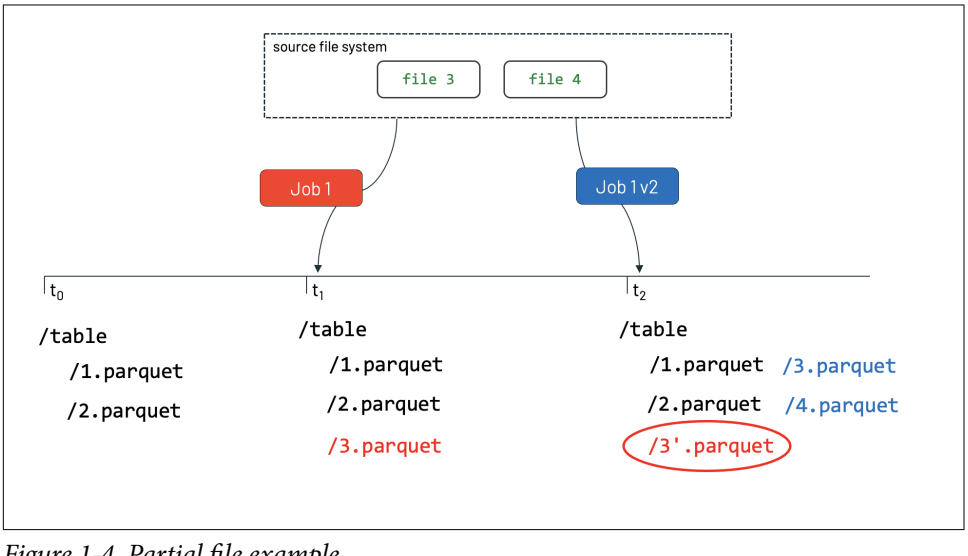


Figure 1-4. Partial file example

For example, the preceding diagram represents a common data lake processing scenario.

Time Period	Table

t0	The table is represented by 2 Parquet files
t1	A processing job (`job 1`) extracts files 3 and 4 and writes them to disk. But due to some error (e.g. network hiccup, storage temporary offline, etc.), an incomplete part of file 3 and none of file 4 are written to storage. There is no automated mechanism to clean up this data; this partial of `3.parquet` can be queried within your data lake.
t2	A new version of the same processing job (`job 1 v2`) is executed and this time it completes its job successfully with `3'.parquet` and `4.parquet`. But because the partial `3.parquet` exists alongside with `3'.parquet` there will be double counting.

By using a transaction log to track which files are valid, we can avoid the preceding scenario. Thus, when a user reads a Delta Lake table for the first time or runs a new query on an open table that has been modified since the last time it was read, Spark checks the transaction log to see what new transactions have posted to the table, and then updates the end user's table with those new changes. This ensures that a user's version of a table is always synchronized with the master record as of the most recent query and that users cannot make divergent, conflicting changes to a table.

Let's repeat the same partial file example except for this time we run this on a Delta table with the Delta transaction log.

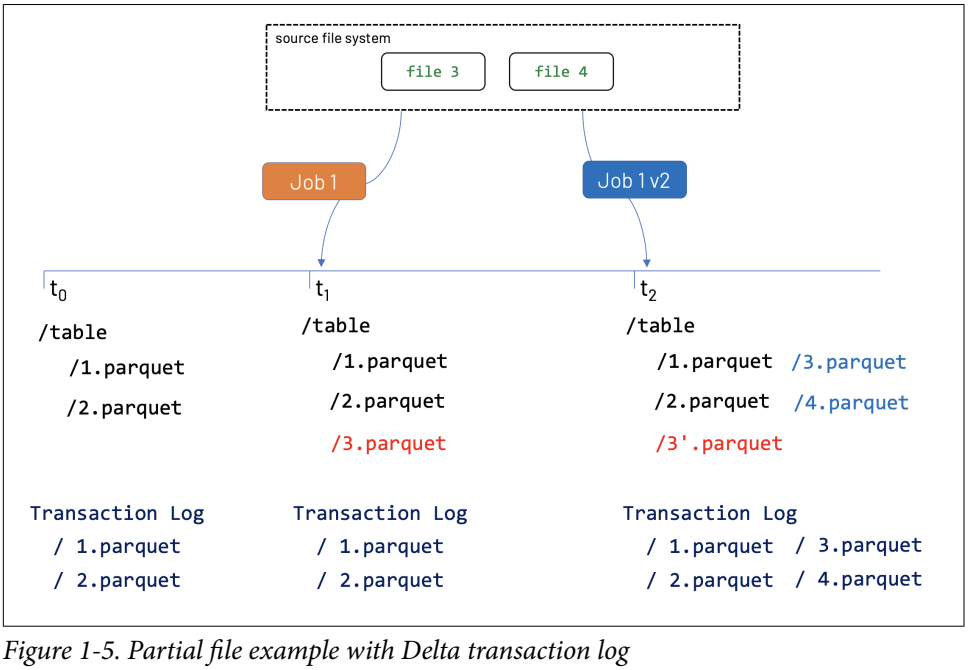


Figure 1-5. Partial file example with Delta transaction log

Time Period	Table	Transaction Log

t0	The table is represented by 2 Parquet files	Version 0: The transaction log records the two files that make up the Delta table.
t1	A processing job (`job 1`) extracts files 3 and 4 and writes them to disk. But due to some error (e.g. network hiccup, storage temporary offline, etc.), an incomplete part of file 3 and none of file 4 are written to storage. There is no automated mechanism to clean up this data; this partial of `3.parquet` can be queried within your data lake.	Because the job fails, the transaction was NOT committed to the transaction log. No new files are recorded. Any queries against the Delta table at this time will be provided a list of the initial two files and Spark will only query these two files even if other files are in the folder.
t2	A new version of the same processing job (`job 1 v2`) is executed and this time it completes its job successfully with `3'.parquet` and `4.parquet`. But because the partial `3.parquet` exists alongside with `3'.parquet` there will be double counting.	Version 1: Because the job completes successfully, a new transaction is committed with the two new files. Thus any queries against the table will be provided with these four files. Because a new transaction is committed, it is saved as a new version (i.e. V1 instead of V0).

The exact same actions happened as in the previous example but this time the Delta transaction log provides *atomicity* and ensures *data reliability*.

The Implementation of Atomicity on Delta Lake

One of the four properties of ACID transactions, atomicity, guarantees that operations (like an INSERT or UPDATE) performed on your data lake either complete fully or don't complete at all. Without this property, it's far too easy for a hardware failure or a software bug to cause data to be only partially written to a table, resulting in messy or corrupted data.

The transaction log is the mechanism through which Delta Lake is able to offer the guarantee of atomicity - if it's not recorded in the transaction log, it never happened. By only recording transactions that execute fully and completely, and using that record as the single source of truth, the Delta Lake transaction log allows users to reason about their data, and have peace of mind about its fundamental trustworthiness, at petabyte scale.

How Does the Transaction Log Work?

This section provides internals of how the Delta transaction log works.

Breaking Down Transactions Into Atomic Commits

Whenever a user performs an operation to modify a table (e.g., `INSERT`, `DELETE`, `UPDATE` or `MERGE`), Delta Lake breaks that operation down into a series of discrete steps composed of one or more of the actions below.

Update metadata
: Updates the table's metadata including but not limited to changing the table's name, schema or partitioning.

Add file
: adds a data file to the transaction log.

Remove file
: removes a data file from the transaction log.

Set transaction
: Records that a structured streaming job has committed a micro-batch with the given ID.

Change protocol
: enables new features by switching the Delta Lake transaction log to the newest software protocol.

Commit info
: Contains information around the commit, which operation was made, from where, and at what time.

Those actions are then recorded in the transaction log as ordered, atomic units known as commits.

Delta Transaction Log Protocol

In this section, we describe how the Delta transaction log brings ACID properties to large collections of data, stored as files, in a distributed file system or object-store. The protocol was designed with the following goals in mind:

Serializable ACID Writes
: multiple writers can concurrently modify a Delta table while maintaining ACID semantics. (Watch this video on how each concurrent writes are handled)

Snapshot Isolation for Reads
: readers can read a consistent snapshot of a Delta table, even when multiple writers are concurrently writing.

Scalability to billions of partitions or files
: queries against a Delta table can be planned on a single machine or in parallel.

Self-describing
: all metadata for a Delta table is stored alongside the data. This design eliminates the need to maintain a separate metastore just to read the data and also allows static tables to be copied or moved using standard filesystem tools.

Support for incremental processing
: readers can tail the Delta log to determine what data has been added in a given period of time, allowing for efficient streaming.

Logstore

Think about the existence of the delta files for a second. The logs, versions, and files that are being generated must exist somewhere, some system or store for files. LogStore is the general interface for all critical file system operations required to read and write the Delta transaction log. Because most storage systems do not provide atomicity guarantees out-of-the-box, Delta Lake transactional operations go through the LogStore API instead of accessing the storage system directly.

- Any file written through this store must be made visible, atomically. In other words, it should be visible in its entirety or not visible at all. It should not generate partial files.
- Only one writer must be able to create a file at the final destination. This is because many processes can occur simultaneously and to ensure decent speed, many writers write to their own files in parallel.
- The Logstore offers ACID consistent listing of files

The Delta Lake Transaction Log at the File Level

As noted previously, when a Delta table is created, that table's transaction log is automatically created in the `_delta_log` subdirectory. As they make changes to that table, those changes are recorded as ordered, atomic commits in the transaction log. Each commit is written out as a JSON file, starting with `000000.json`. Additional changes to the table generate subsequent JSON files in ascending numerical order so that the next commit is written out as `000001.json`, the following as `000002.json`, and so on. Each numeric JSON file increment represents a new version of the table.

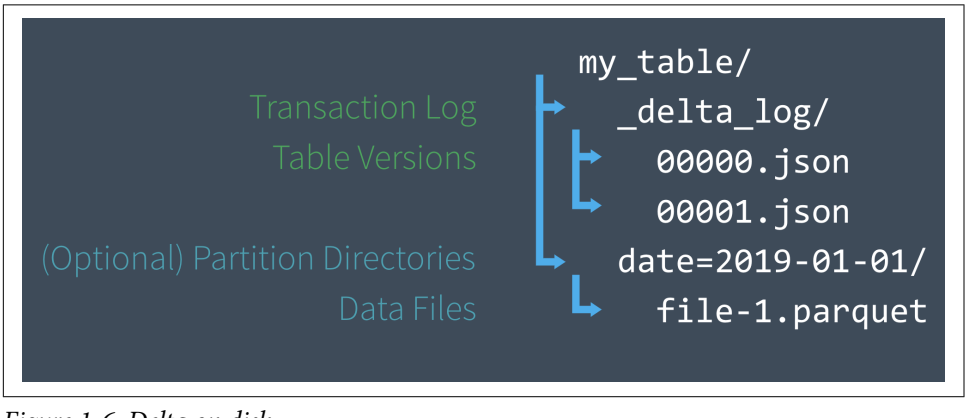


Figure 1-6. Delta on disk

Implementing Atomicity. An important thing to understand is the transaction log is the single source of truth for your Delta table. So any reader that's reading through

your Delta table, will take a look at the transaction log first. Therefore changes to the table are stored as ordered atomic units called commits.

To continue the above example, we add additional records to our Delta table from the data files `1.parquet` and `2.parquet`. That transaction would automatically be added to the transaction log, saved to disk as commit `000000.json`. Then, perhaps we change our minds and decide to remove those files (e.g., run a `DELETE` from table operation) and add a new file instead (`3.parquet`) through an `INSERT` operation. Those actions would be recorded as the next commit - and table version - in the transaction log, as `000001.json`, as shown in the following diagram.

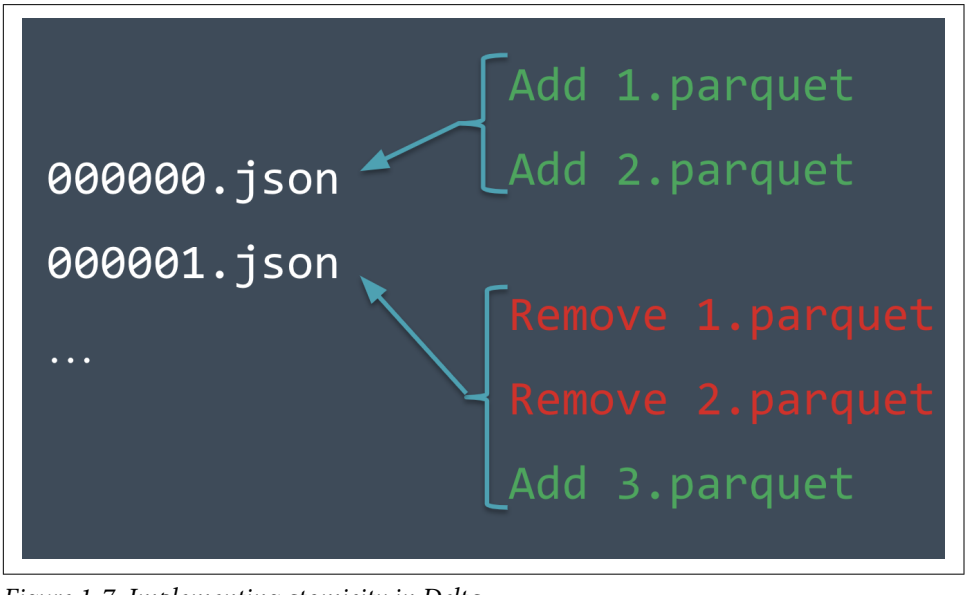


Figure 1-7. Implementing atomicity in Delta

Even though `1.parquet` and `2.parquet` are no longer part of our Delta Lake table, their addition and removal are still recorded in the transaction log because those operations were performed on our table – despite the fact that they ultimately canceled each other out. Delta Lake still retains atomic commits like these to ensure that in the event we need to audit our table or use "time travel" (to discuss in chapter 3) to see what our table looked like at a given point in time, we could do so accurately.

To provide context, let's continue with our notebook and read the JSON file that makes up the first transaction (version 1) inserting data into our Delta table.

```
>%python
# Read the transaction log of version 1
j0 = spark.read.json("/.../000001.json")
```

```
%scala
// Read the transaction log of version 1
val j0 = spark.read.json("/.../000001.json")
```

Figure 1-8. Reviewing the transaction log structure

The transaction contains many pieces of information stored as strings or columns within the transaction log JSON. Let's focus on the `commit`, `add`, and `CRC` pieces.

Commit Information. The Delta transaction log commits metadata can be read using the following code snippets.

```
%python
# Commit Information
display(j0.select("commitInfo").where("commitInfo is not null"))
```

```
%scala
// Commit Information
display(j0.select("commitInfo").where("commitInfo is not null"))
```

The following is a text version of the JSON output read from the preceding command.

```
clusterId: "0127-045215-pined152"
isBlindAppend: true
isolationLevel: "WriteSerializable"
notebook: {"notebookId": "8476282"}
operation: "STREAMING UPDATE"
operationMetrics: {"numAddedFiles": "1", "numOutputBytes": "492", "numOutpu-
tRows": "0", "numRemovedFiles": "0"}
operationParameters: {"epochId": "0", "outputMode": "Append", "queryId":
"892c1abd-581f-4c0f-bbe7-2c386feb3dbd"}
readVersion: 0
timestamp: 1603581133495
userId: "100599"userName: "denny[dot]lee[@]databricks.com"
```

The metadata contains a lot of interesting information, but let's focus on the ones around the file system.

metadata	description
operation	The type of operation that is happening, in this case, data was inserted via Spark Structured streaming writeStream job (i.e. `STREAMING UPDATE`)
operationMetrics	Notes how many files were added (`numAddedFiles`), removed (`numRemovedFiles`), output rows (`numOutputRows`), and output bytes (`numOutputBytes`)
operationParameters	From the files perspective, whether this operation would **append** or **overwrite** the data within the table.
readVersion	The table version associated with this transaction commit.
clusterID, notebook	Identifies which Databricks cluster and notebook that executed this commit
userID, userName	ID and name of the user executing this operation

Add Information. The Delta transaction log `add` metadata can be read using the following code snippets.

```
%python
# Commit Information
display(j0.select("add").where("add is not null"))
```

```
%scala
// Commit Information
display(j0.select("add").where("add is not null"))
```

The following is a text version of the JSON output read from the preceding command.

```
dataChange: true
modificationTime: 1603581134000
```

```
path: "part-00000-95648a41-bf33-4b67-9979-1736d07bef0e-c000.snappy.parquet"
size: 492
stats: "{\"numRecords\":0,\"minValues\":{},\"maxValues\":{},\"nullCount\":{}}"
```

The metadata is particularly important in that it lists the file that makes up the table version.

metadata	description
path	A list of the file(s) added to a Delta table per a committed transaction
size	The size of the file(s)
stats	The statistics stored in the transaction log to help the reader understand the size and scope of data files that need to be read.

Typically when reading files from storage, Spark (and other distributed processing frameworks) will simply read all of the files in a folder. Prior to reading the files, it must first list the files (`listFrom`) which can be extremely inefficient especially on cloud object stores.

But in the case of Delta Lake, the files are listed directly within the transaction log itself. So instead of Spark (or any other processing framework that can read the Delta transaction log), listing all the files, the transaction log provides all of the files that are associated with a table version. Therefore Spark can just query the files directly through their individual paths which can have a significant performance improvement than `listFrom` especially for internet-scale systems with petabytes of data.

CRC file. For each transaction, there is both a JSON file as well as a CRC file. This file contains key statistics for the table version (i.e. transaction) allowing Delta Lake to help Spark optimize its queries.

To review this data, let's just review the file directly from the file system using the following command.

```
%sh head /dbfs/ml/loan_by_state/_delta_log/
00000000000000000001.crc
```

Below are the results from this query

```
{
"tableSizeBytes":1855,
"numFiles":2,
"numMetadata":1,
"numProtocol":1,
"numTransactions":1
}
```

The two most notable pieces of metadata are:

`tableSizeInBytes`
: The table size in bytes so Spark and Delta Lake can optimize their queries

`numFiles`
: Important for Spark especially in scenarios like dynamic partition pruning

As you can read from the preceding high-level primer, the Delta Lake transaction log tracks the files and other metadata to ensure both *atomic transactions* and *data reliability*.

Spark does not eagerly remove the files from disk, even though we removed the underlying data files from our table. Users can delete the files that are no longer needed using VACUUM (more on this later under the Table Utilities section).

Quickly Recomputing State With Checkpoint Files

Prior to this, our focus is that of a single transaction log in the form of JSON files. But for large-scale systems or any streaming system, this would result in creating the "small-file" problem where it becomes ever more inefficient to query the transaction log folder (i.e. `_delta_log` subdirectory).

To alleviate this issue, Delta Lake creates a checkpoint file in Parquet format after it creates the 10th commits (i.e. transaction).

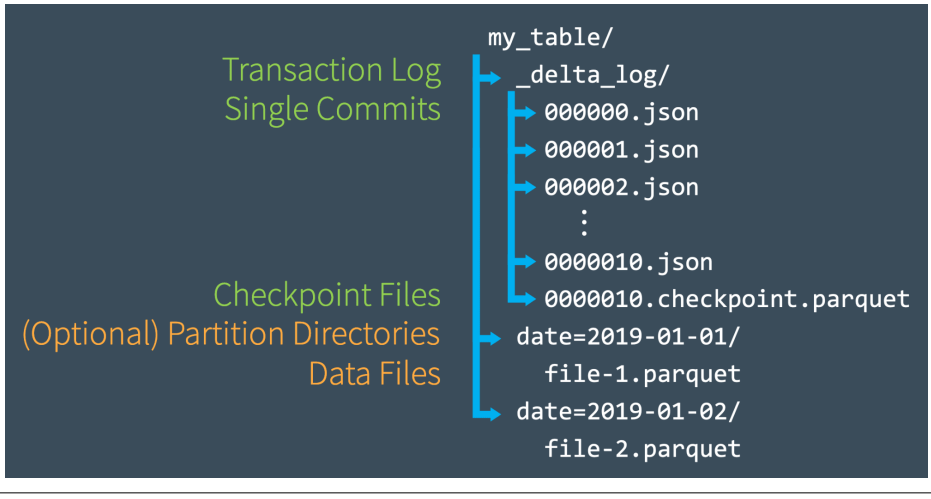


Figure 1-9. Create transaction log after every 10th commit

These checkpoint files save the entire state of the table at a point in time – in native Parquet format that is quick and easy for Spark to read. In other words, they offer the

Spark reader a sort of "shortcut" to fully reproducing a table's state that allows Spark to avoid reprocessing what could be thousands of tiny, inefficient JSON files.

When recomputing the state of the table, Spark will read and cache the available JSON files that make up the transaction log. For example, if there have been only three committed operations or commits to the table (including the table creation), Spark will read all three files and cache the results into memory (i.e. cache version 2).

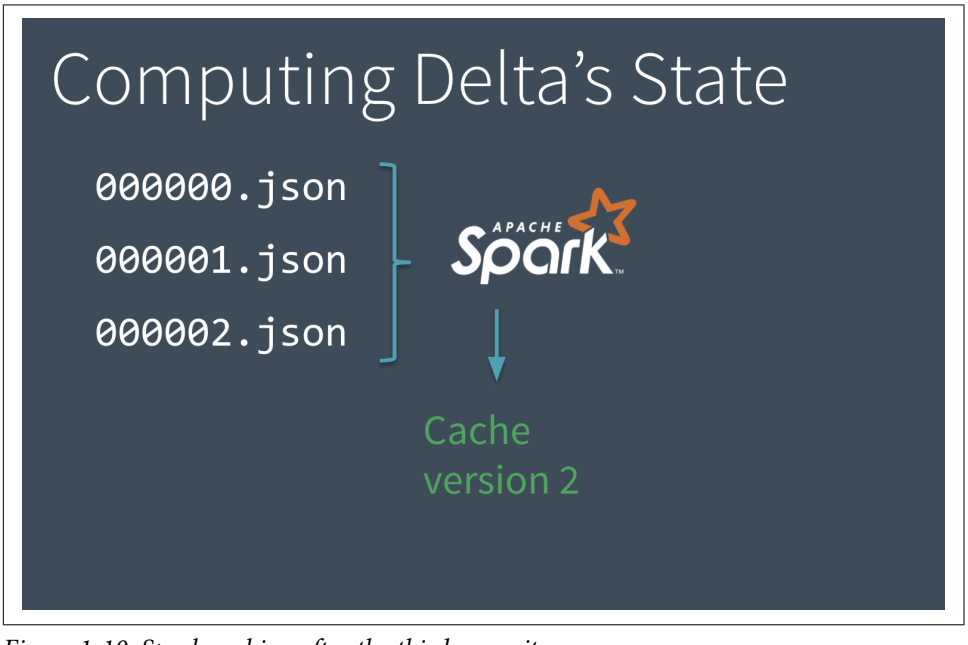


Figure 1-10. Spark caching after the third commit

Instead of continually reading the transaction log for this information, all of the Spark readers requesting data from this table can simply reference the cached copy of Delta's state. As more commits are performed against the Delta table, more JSON files will be added to the `_delta_log` folder.

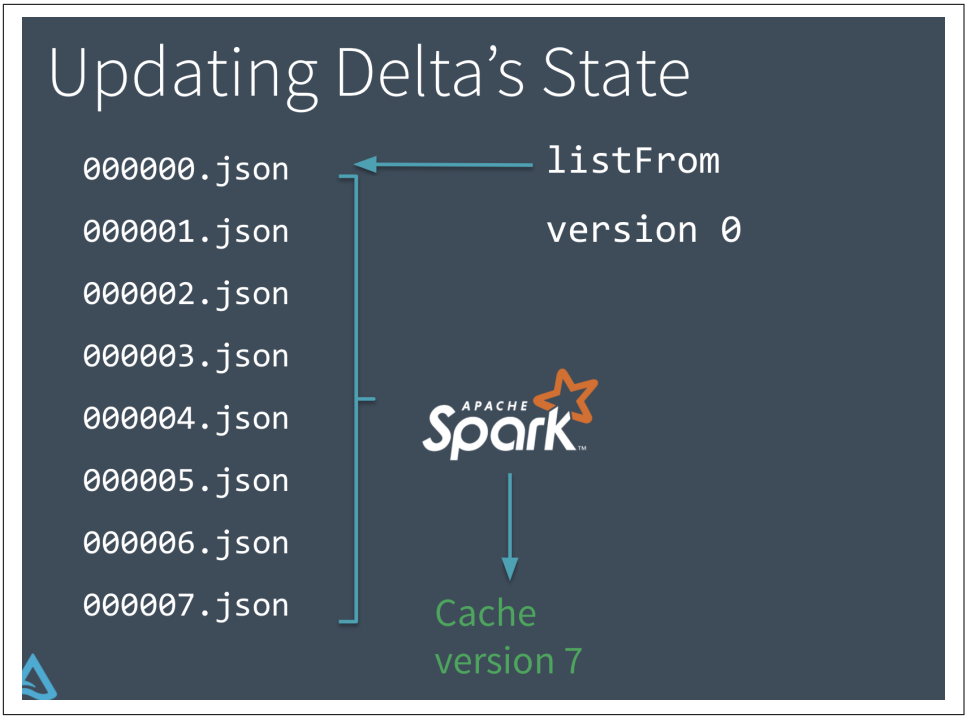


Figure 1-11. Spark caching after the eight commit

To continue this example, let's say this table five additional commits, then Spark will cache version 7 of the data. Note, at this point, Spark will list all of the transactions from version 0 instead of reading from version 2 to ensure that earlier transactions have completed.

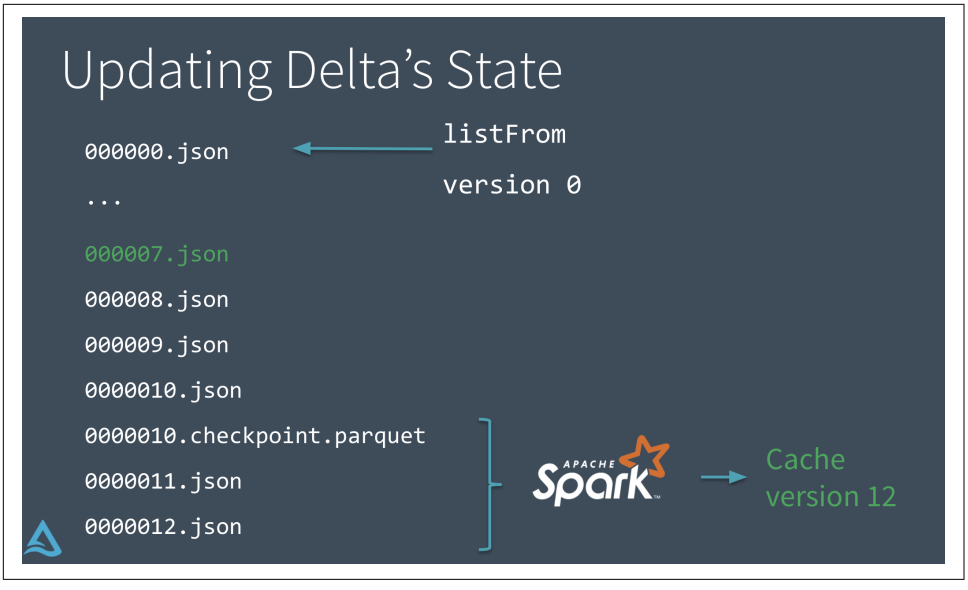


Figure 1-12. Spark caching after the 13th commit including checkpoint creation

As noted earlier, Delta Lake will create a checkpoint file (`0000010.checkpoint.parquet`) after the 10th commit. Delta Lake will still listFrom version 0 to avoid late transactions and cache version 12.

To read the checkpoint file, run the following statement.

```
%python
# Review the transaction log checkpoint file
chkpt0 = spark.read.parquet("/ml/loan_by_state/_delta_log/000010.check
point.parquet")

%scala
// Review the transaction log checkpoint file
val chkpt0 = spark.read.parquet("/ml/loan_by_state/_delta_log/000010.check
point.parquet")
```

The metadata you will read is a union of all of the previous transactions. This becomes apparent when you read the query for the `add` information.

```
%python
# Add Information
display(chkpt0.select("add").where("add is not null"))

%scala
// Add Information
display(chkpt0.select("add").where("add is not null"))
```

Figure 1-13. Add information in the checkpoint file

In addition to the checkpoint file being in Parquet format (thus Spark can read it even faster) and containing all of the transactions prior to it, notice how the `stats` in the original JSON file were in string format.

```
stats: "{\"numRecords\":7,\"minValues\":{\"addr_state\":\"IA\",\"count\":
3,\"stream_no\":3},\"maxValues\":{\"addr_state\":\"TX\",\"count\":9,\"stream_no
\":3},\"nullCount\":{\"addr_state\":0,\"count\":0,\"stream_no\":0}}"
```

As part of the checkpoint creation process, there is a `stats_parsed` column that contains the statistics as nested columns instead of strings.

```
stats_parsed:{
"numRecords": 7,
"minValues": {
"addr_state": "IA",
"count": 3,
"stream_no": 3
},
"maxValues": {
"addr_state": "TX",
"count": 9,
"stream_no": 3
```

```
},
"nullCount": {
"addr_state": 0,
"count": 0,
"stream_no": 0
}
}
```

By using nested columns instead of strings, Spark can read the statistics significantly faster especially when it needs to read all of the files created for petabyte-scale data lakes.

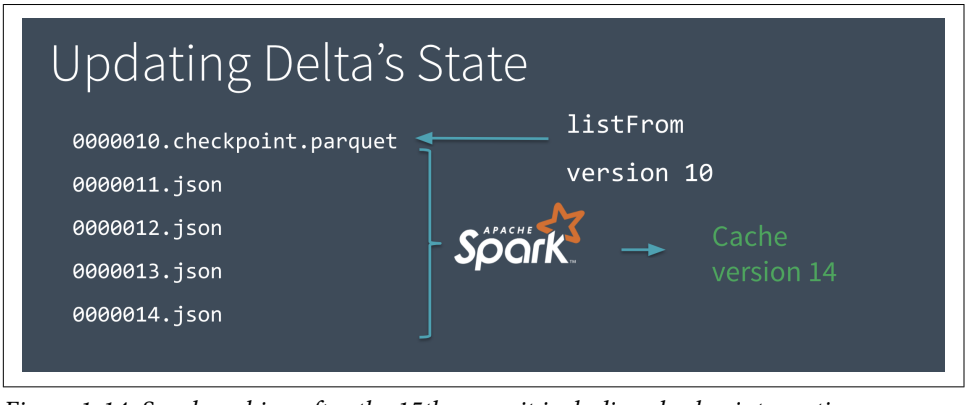


Figure 1-14. Spark caching after the 15th commit including checkpoint creation

As it is important to note that once the Parquet checkpoint file has been created, subsequent listFrom calls are from the checkpoint file instead of going back to version 0. Thus after the 15 commits, Delta will cache version 14 but need only to listFrom the version 10 checkpoint file.

Dealing With Multiple Concurrent Reads and Writes

Now that we understand how the Delta Lake transaction log works at a high level, let's talk about concurrency. So far, our examples have mostly covered scenarios in which users commit transactions linearly, or at least without conflict. But how does Delta Lake deal with multiple concurrent reads and writes? Because Delta Lake is powered by Apache Spark, the expectation is that multiple users concurrently modify a single table. To do this, Delta Lake employs optimistic concurrency control.

What Is Optimistic Concurrency Control?

Optimistic concurrency control is a method of dealing with concurrent transactions that assumes that transactions made to a table by different users can complete without conflicting with one another. It is incredibly fast because when dealing with petabytes of data, there's a high likelihood that users will be working on different

parts of the data altogether, allowing them to complete non-conflicting transactions simultaneously.

When working with tables, as long as different clients are modifying different parts of the table (e.g. different partitions) or performing actions that do not conflict (e.g. two clients reading from the table), those operations do not conflict so we can optimistically let them all complete their task. But for situations where clients modify the same parts of the table concurrently, Delta Lake has a protocol to resolve this.

Solving conflicts optimistically

Ensuring serializability. Another key piece that Delta requires for consistent guarantees is **mutual exclusion**. We need to agree on the order of changes, even when there are multiple writers and this provides a guarantee in databases called *serializability*. The fact that even though things are happening concurrently, you could play them as if they happened in a synchronous ordered manner.

For example, we have user 1 reading 000000.json

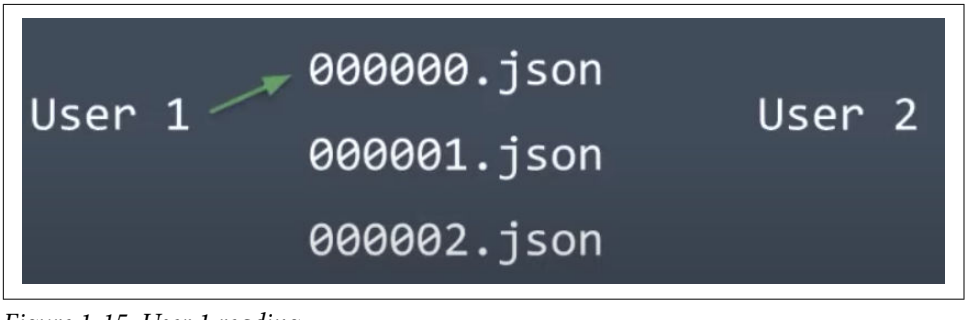


Figure 1-15. User 1 reading

And we have user2 that reads 000001.json

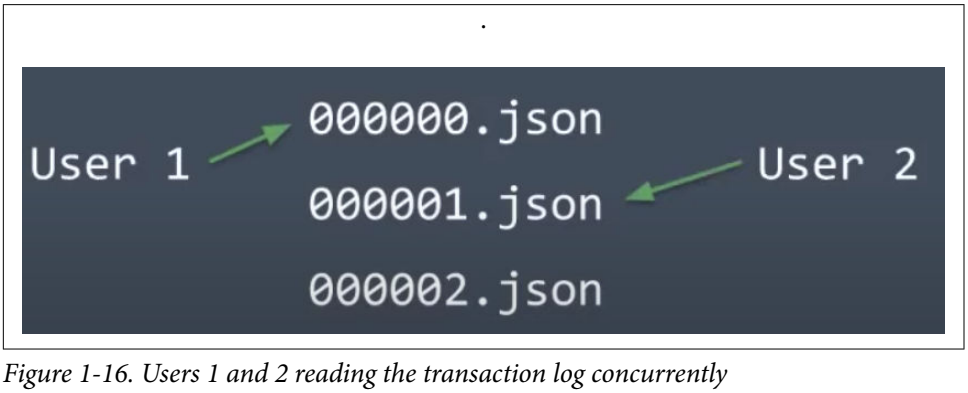


Figure 1-16. Users 1 and 2 reading the transaction log concurrently

As you're trying to commit `000002.json`, user2 two wins and user1 takes a look and sees that, `000002.json` is already there. Due to this requirement of mutual exclusion, it will have to say, "Oh this commit failed" and let me try to commit `000003.json` instead.

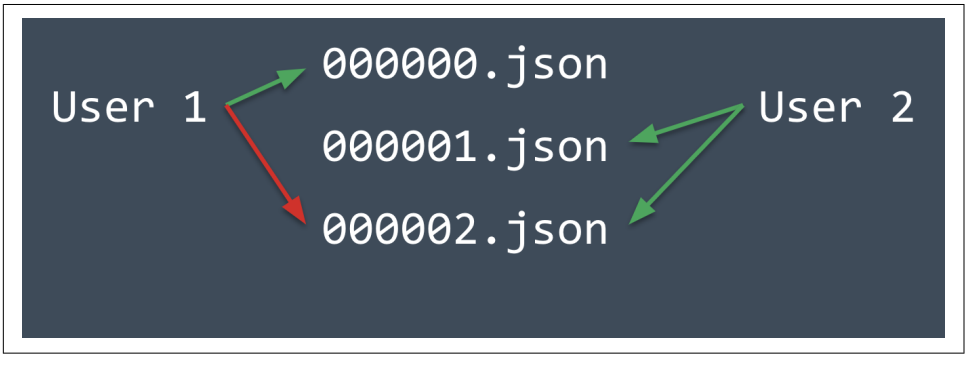


Figure 1-17. Ensuring serializability

Applying Optimistic Concurrency Control in Delta. In order to offer ACID transactions, Delta Lake has a protocol for figuring out how commits should be ordered (known as the concept of **serializability** in databases) and determining what to do in the event that two or more commits are made at the same time.

Delta Lake handles these cases by implementing a rule of **mutual exclusion**, then attempting to solve any conflict optimistically. This protocol allows Delta Lake to deliver on the ACID principle of **isolation**, which ensures that the resulting state of the table after multiple, concurrent writes is the same as if those writes had occurred serially, in isolation from one another.

In general, the process proceeds like this:

1. Record the starting table version.
2. Record reads/writes.
3. Attempt a commit.
4. If someone else wins, check whether anything you read has changed.
5. Repeat.

For example, if two users read from the same table and then both attempt to insert data at the same time.

Figure 1-18. Solving conflicts optimistically

- Delta Lake records the starting table version of the table (version 0) that is read prior to making any changes.
- Users 1 and 2 both attempt to append some data to the table at the same time. Here, we've run into a conflict because only one commit can come next and be recorded as `000001.json`.
- Delta Lake handles this conflict with the concept of mutual exclusion, i.e. only one user can successfully commit `000001.json`. User 1's commit is accepted, while User 2's is rejected.
- Rather than throw an error for User 2, Delta Lake prefers to handle this conflict optimistically. It checks to see whether any new commits have been made to the table, and updates the table silently to reflect those changes, then simply retries User 2's commit on the newly updated table (without any data processing), successfully committing `000002.json`.

In the vast majority of cases, this reconciliation happens silently, seamlessly, and successfully. However, in the event that there's an irreconcilable problem that Delta Lake cannot solve optimistically (for example, if User 1 deleted a file that User 2 also deleted), the only option is to throw an error.

As a final note, since all of the transactions made on Delta Lake tables are stored directly to storage, this process satisfies the ACID property of **durability**, meaning it will persist even in the event of system failure.

Multiversion Concurrency Control. How all of this works within the file system is that Delta's transactions are implemented using Multiversion Concurrency Control (MVCC). It is a concurrency control method commonly used by relational database management systems to provide concurrent access to the database (and its tables)

within the context of transactions. As Delta Lake's data objects and log are immutable, Delta Lake utilizes MVCC to both protect existing data, i.e. provides transactional guarantees between writes, as well as speed up query and write performance. It also has the benefit of making it straightforward to query a past snapshot of the data, as is common in MVCC implementations.

Under this mechanism, writes operate in three stages:

Read
: First the system reads the latest available version of the table to identify which rows need to be modified.

Write
: Stages all the changes by writing new data files. Note, all changes whether inserts or modifications are in the form of writing new files.

Validate and commit
: Before committing the changes, it checks whether the proposed changes conflict with any other changes that may have been concurrently committed since the snapshot that was read. If there are no conflicts, all the staged changes are committed as a new versioned snapshot, and the write operation succeeds. However, if there are conflicts, the write operation fails with a concurrent modification exception rather than corrupting the table as would happen with the write operation on a Parquet table.

As a table changes, Delta's MVCC algorithm keeps multiple copies of the data around rather than immediately replacing files that contain records that are being updated or removed. MVCC allows serializability and snapshot isolation for consistent views of a state of a table so that the readers continue to see a consistent snapshot view of the table that the Apache Spark job started with, even when a table is modified during a job. They can efficiently query a snapshot by using the transaction log to selectively choose which data files to process when there are concurrent modifications being done by writers.

Therefore, writers modify a table in two phases:

- First, they optimistically write out new data files or updated copies of existing ones.
- Then, they commit, creating the latest atomic version of the table by adding a new entry to the log. In this log entry, they record which data files to logically add and remove, along with changes to other metadata about the table.

Other Use Cases

There are many interesting use cases that can be implemented because of Delta Lake's transaction log. Below are a couple of examples that we will dive into in chapter 3.

Time Travel

Every table is the result of the sum total of all of the commits recorded in the Delta Lake transaction log – no more and no less. The transaction log provides a step-by-step instruction guide, detailing exactly how to get from the table's original state to its current state.

Therefore, we can recreate the state of a table at any point in time by starting with an original table, and processing only commits made prior to that point. This powerful ability is known as "time travel," or data versioning, and can be a lifesaver in any number of situations. For more information, read the blog post Introducing Delta Time Travel for Large Scale Data Lakes, or refer to the Delta Lake time travel documentation.

Data Lineage and Debugging

As the definitive record of every change ever made to a table, the Delta Lake transaction log offers users a verifiable data lineage that is useful for governance, audit, and compliance purposes. It can also be used to trace the origin of an inadvertent change or a bug in a pipeline back to the exact action that caused it. Users can run DESCRIBE HISTORY to see metadata around the changes that were made.

Diving further into the transaction log

In this section, we dove into the details of how the Delta Lake transaction log works, including:

- What the transaction log is, how it's structured, and how commits are stored as files on disk.
- How the transaction log serves as a single source of truth, allowing Delta Lake to implement the principle of atomicity.
- How Delta Lake computes the state of each table – including how it uses the transaction log to catch up from the most recent checkpoint.
- Using optimistic concurrency control to allow multiple concurrent reads and writes even as tables change.
- How Delta Lake uses mutual exclusion to ensure that commits are serialized properly, and how they are retried silently in the event of a conflict.

For more information, refer to:

- Diving into Delta Lake: Unpacking the Transaction Log (blog)
- Diving into Delta Lake: Unpacking the Transaction Log (video tech talk)
- Diving into Delta Lake: Unpacking the Transaction Log v2 (Data + AI Summit EU 2020 session)

Table Utilities

If there is a central theme for reviewing the transaction log, MVCC, optimistic concurrency control, and the underlying file system when working with Delta Lake is that it is about the manipulation of the underlying files that make up the Delta table. In this section, we provide an overview of the various table utilities to simplify operations.

Review table history

You can retrieve information on the operations, user, timestamp, and so on for each write to a Delta table by running the history command. This history will tell us things like what type of write occurred (append, merge, delete, etc.), was it a blind append, was it restricted to a specific partition, and provides operational metrics on the amount of data written. All of this information is interesting, but the operational metrics provide the most insight into how your table is changing, showing how many files, rows, and bytes were added or removed.

Note, the operations are returned in reverse chronological order and by default, table history is retained for 30 days.

```
%sql
-- Review history by Delta table file path
DESCRIBE HISTORY delta.`/ml/loan_by_state`;

-- Review history by Delta table defined in the metastore as `loan_by_state`
DESCRIBE HISTORY loan_by_state;

-- Review last 5 operations by Delta table defined in the metastore as
`loan_by_state`
DESCRIBE HISTORY loan_by_state LIMIT 5;
```

```
%python
from delta.tables import *
deltaTable = DeltaTable.forPath(spark, "/ml/loan_by_state")

# get the full history of the table
fullHistoryDF = deltaTable.history()

# get the last 5 operations
last5OperationsDF = deltaTable.history(5)
```

```
%scala
import io.delta.tables._
val deltaTable = DeltaTable.forPath(spark, pathToTable)
// get the full history of the table
val fullHistoryDF = deltaTable.history()

// get the last 5 operations
val lastOperationDF = deltaTable.history(5)
```

The following is a screenshot of the full history of the Delta table stored in `/ml/loan_by_state` from this notebook.

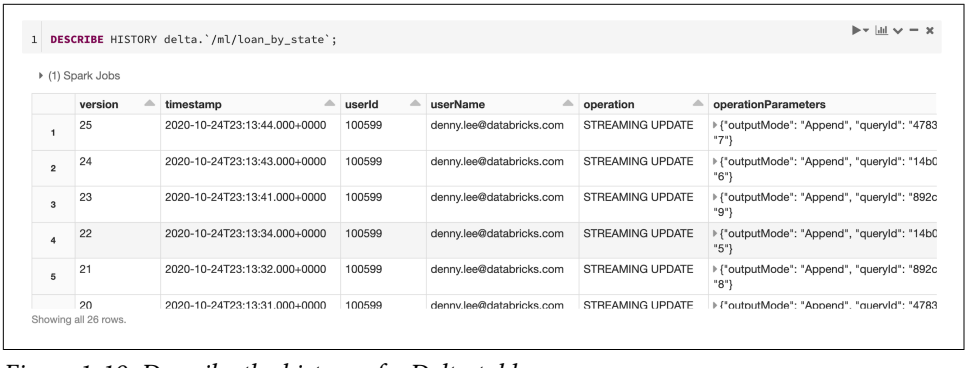


Figure 1-19. Describe the history of a Delta table

Note:

- Some of the columns may be nulls because the corresponding information may not be available in your environment.
- Columns added in the future will always be added after the last column.
- For the most current reference, refer to Delta Lake history schema and operation metric keys.

We will dive further into the application of table history to time travel in Chapter 3: Time Travel with Delta.

Vacuum History

Over time, more files will accumulate into your Delta table; many of the older files may no longer be needed because they represent previously overwritten versions of data (e.g. `UPDATE`, `DELETE`, etc.). Therefore, you can remove files no longer referenced by a Delta table and are older than the retention threshold by running the `VACUUM` command on the table. Important note, `VACUUM` is **not** triggered automatically. When it is triggered, the default retention threshold for the files is 7 days, i.e. no longer referenced files older than 7 days will be removed.

```
%sql
-- vacuum files in a path-based table by default retention threshold
VACUUM delta.'/data/events'

-- vacuum files by metastore defined table by default retention threshold
VACUUM eventsTable

-- vacuum files by metastore defined table that are no longer required older
than 100 hours old
VACUUM eventsTable  RETAIN 100 hours

-- dry run: get the list of files to be deleted
VACUUM eventsTable DRY RUN
```

```
%python
from delta.tables import *

# vacuum files in path-based tables
deltaTable = DeltaTable.forPath(spark, pathToTable)

# vacuum files in metastore-based tables
deltaTable = DeltaTable.forName(spark, tableName)

# vacuum files in path-based table by default retention threshold
deltaTable.vacuum()

# vacuum files not required by versions more than 100 hours old
deltaTable.vacuum(100)
```

```
%scala
import io.delta.tables._

# vacuum files in path-based tables
val deltaTable = DeltaTable.forPath(spark, pathToTable)

# vacuum files in metastore-based tables
val deltaTable = DeltaTable.forName(spark, tableName)

# vacuum files in path-based table by default retention threshold
deltaTable.vacuum()

# vacuum files not required by versions more than 100 hours old
deltaTable.vacuum(100)
```

Configure Log and Data History

Log history. How much history your Delta table retains is configurable per table using the config `spark.databricks.delta.logRetentionDuration`, defaulting to 30 days. The Delta Transaction Log is cleaned up during new commits to the table if the logs have passed the set retention period. This is done so that the Delta Log does not

grow indefinitely, but the Delta Log only contains information about what files are in the table.

In conjunction with the Log retention period, there are also the data files to consider. Every time data is changed in a Delta table, there are old files marked for deletion and new files added. While the Delta Log is cleaned up automatically on write, data files must be deleted explicitly with a call to the vacuum API. `VACUUM` can be quite slow on cloud object storage but needs very little resources, so it makes sense to schedule this separately to run on some cadence, e.g. weekly.

Data history. The config `spark.databricks.delta.deletedFileRetentionDuration` controls how long ago the files were marked for deletion before they are deleted by `VACUUM`. The default here is 7 days, as an application must actively call the API for the files to be deleted.

Parallel deletion of files during vacuum. When using VACUUM, to configure Spark to delete files in parallel (based on the number of shuffle partitions) set the session configuration "`spark.databricks.delta.vacuum.parallelDelete.enabled`" to "`true`".

Warning

We do not recommend that you set a retention interval shorter than 7 days because old snapshots and uncommitted files can still be in use by concurrent readers or writers to the table. If vacuum cleans up active files, concurrent readers can fail or, worse, tables can be corrupted when vacuum deletes files that have not yet been committed.

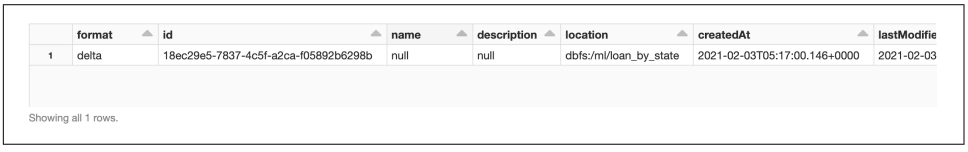
Delta Lake has a safety check to prevent you from running a dangerous vacuum command. If you are certain that there are no operations being performed on this table that take longer than the retention interval you plan to specify, you can turn off this safety check by setting the Apache Spark configuration property `spark.databricks.delta.retentionDurationCheck.enabled` to `false`. You must choose an interval that is longer than the longest-running concurrent transaction and the longest period that any stream can lag behind the most recent update to the table.

Retrieve Delta table details

`DESCRIBE DETAIL` functionality allows you to review the table metadata including (but not limited to) table size, schema, partition columns, and other metrics on file and file sizes. You can use the table name or file path to specify the Delta table as demonstrated below.

```
%sql
-- Describe detail using Delta file path
DESCRIBE DETAIL delta.`/ml/loan_by_state`;
```

```
-- Describe detail using metastore-defined Delta table
DESCRIBE DETAIL loan_by_state;
```



	format	id	name	description	location	createdAt	lastModifie
1	delta	18ec29e5-7837-4c5f-a2ca-f05892b6298b	null	null	dbfs:/ml/loan_by_state	2021-02-03T05:17:00.146+0000	2021-02-03

Showing all 1 rows.

Figure 1-20. Describe Delta table details

Below is a transposed view of these values.

Column name	value
format	delta
id	18ec29e5-7837-4c5f-a2ca-f05892b6298b
name	loan_by_state
description	null
location	dbfs://ml/loan_by_state
createdAt	2021-02-03T05:17:00.146+0000
lastModified	2021-02-03T05:26:37.000+0000
partitionColumns	[]
numFiles	292
sizeInBytes	308386
properties	{"delta.checkpoint.writeStatsAsStruct": "true"}
minReaderVersion	1
minWriterVersion	2

For the current schema of the columns described by `DESCRIBE DETAIL`, refer to Detail schema.

Note, while the following metadata queries are not specific to Delta Lake, they are helpful when working with metastore-defined tables.

DESCRIBE TABLE

Describe Table returns the basic metadata information of a table. The metadata information includes column name, column type and column comment. Optionally you can specify a partition spec or column name to return the metadata pertaining to a partition or column respectively.

Display detailed information about the specified columns, including the column statistics collected by the following command. Note, this command is applicable to only metastore-defined tables.

```
%sql
DESCRIBE TABLE loan_by_state;
```

```
# result

+---------------+---------+-------+
|       col_name|data_type|comment|
+---------------+---------+-------+
|     addr_state|   string|       |
|          count|   bigint|       |
|      stream_no|      int|       |
|               |         |       |
| # Partitioning|         |       |
|Not partitioned|         |       |
+---------------+---------+-------+
```

Along with the basic metadata information and partitioning, `DESCRIBE TABLE EXTENDED` returns detailed table information such as parent database, table name, location of table, provider, table properties, etc.

```
%sql
DESCRIBE TABLE EXTENDED loan_by_state;

+--------------------+--------------------+-------+
|            col_name|           data_type|comment|
+--------------------+--------------------+-------+
|          addr_state|              string|       |
|               count|              bigint|       |
|           stream_no|                 int|       |
|                    |                    |       |
|      # Partitioning|                    |       |
|     Not partitioned|                    |       |
|                    |                    |       |
|# Detailed Table ...|                    |       |
|                Name|denny_db.loan_by_...|       |
|            Location|dbfs:/ml/loan_by_...|       |
|            Provider|               delta|       |
|    Table Properties|[delta.checkpoint...|       |
+--------------------+--------------------+-------+
```

For the current schema of the columns described by DESCRIBE TABLE, refer to Spark SQL Guide > DESCRIBE TABLE.

Generate a manifest file

To allow non-Spark systems to query Delta Lake without querying the transaction log, you can create a manifest file that those systems will query. To learn more about how to configure systems like Presto and Athena to read a Delta table, jump to Chapter 9: Integrating with other engines.

The following code generates the manifest file itself.

```
%sql
-- Generate manifest
GENERATE symlink_format_manifest FOR TABLE delta.`<path-to-delta-table>`
```

```
%python
# Generate manifest
deltaTable = DeltaTable.forPath(<path-to-delta-table>)
deltaTable.generate("symlink_format_manifest")

%scala
// Generate manifest
val deltaTable = DeltaTable.forPath(<path-to-delta-table>)
deltaTable.generate("symlink_format_manifest")
```

The preceding command creates the following file if the <path-to-delta-table> is defined as `/ml/loan-by-state`.

```
$/ml/loan-by-state.delta/_symlink_format_manifest
```

Convert a Parquet table to a Delta table

Converting an existing Parquet table to a Delta table in-place is straightforward. This command lists all the files in the directory, creates a Delta Lake transaction log that tracks these files, and automatically infers the data schema by reading the footers of all Parquet files. If your data is partitioned, you must specify the schema of the partition columns as a DDL-formatted string (that is, `<column-name1> <type>, <column-name2> <type>, ...`).

```
%sql
-- Convert non partitioned parquet table at path '<path-to-table>'
CONVERT TO DELTA parquet.`<path-to-table>`

-- Convert partitioned Parquet table at path '<path-to-table>' and partitioned
by integer columns named 'part' and 'part2'
CONVERT TO DELTA parquet.`<path-to-table>` PARTITIONED BY (part int, part2 int)

%python
from delta.tables import *

# Convert non partitioned parquet table at path '<path-to-table>'
deltaTable = DeltaTable.convertToDelta(spark, "parquet.`<path-to-table>`")

# Convert partitioned parquet table at path '<path-to-table>' and partitioned
by integer column named 'part'
partitionedDeltaTable = DeltaTable.convertToDelta(spark, "parquet.`<path-to-
table>`", "part int")

%scala
import io.delta.tables._

// Convert non partitioned Parquet table at path '<path-to-table>'
val deltaTable = DeltaTable.convertToDelta(spark, "parquet.`<path-to-table>`")

// Convert partitioned Parquet table at path '<path-to-table>' and partitioned
by integer columns named 'part' and 'part2'
```

```
val partitionedDeltaTable = DeltaTable.convertToDelta(spark, "parquet.`<path-to-
table>`", "part int, part2 int")
```

Some important notes:

- If a Parquet table was created by Structured Streaming, the listing of files can be avoided by using the `_spark_metadata` sub-directory as the source of truth for files contained in the table by setting the SQL configuration `spark.databricks.delta.convert.useMetadataLog` to `true`.
- Any file not tracked by Delta Lake is invisible and can be deleted when you run a vacuum. You should avoid updating or appending data files during the conversion process. After the table is converted, make sure all writes go through Delta Lake.

Convert a Delta table to a Parquet table

In case you need to convert a Delta table to parquet you can follow the following steps:

1. If you have performed Delta Lake operations that can change the data files (for example, delete or merge), run `VACUUM` with a retention of 0 hours to delete all data files that do not belong to the latest version of the table.
2. Delete the `_delta_log` directory in the table directory.

Restore a table version

You can restore your Delta table to a previous version. As noted in the previous Review table history section, a Delta table internally maintains historic versions of the table that enable it to be restored to an earlier state.

```
%sql
-- restore version 9 of metastore defined Delta table
INSERT OVERWRITE INTO loan_by_state
SELECT * FROM loan_by_state VERSION AS OF 9
```

The `RESTORE` command is currently only available on Databricks

If you're using Databricks, you can also use the `RESTORE` command to simplify this process.

```
%sql
RESTORE TABLE loan_by_state TO VERSION AS OF 9
RESTORE TABLE delta.`/ml/loan_by_state/` TO TIMESTAMP AS OF 9

%python
from delta.tables import *

# path-based Delta tables
deltaTable = DeltaTable.forPath(spark, `/ml/loan_by_state/`)

# metastore-based tables
deltaTable = DeltaTable.forName(spark, `loan_by_state`)

# restore table to version 9
deltaTable.restoreToVersion(9)

%scala
import io.delta.tables._

// path-based Delta tables
val deltaTable = DeltaTable.forPath(spark, `/ml/loan_by_state/`)

// metastore-based tables
val deltaTable = DeltaTable.forName(spark, `loan_by_state`)

// restore table to version 9
deltaTable.restoreToVersion(9)
```

Clone your Delta Tables

Table cloning or creating copies of tables in a data lake or data warehouse has several practical uses. But, given the expansive volume of data in tables in a data lake and the rate of its growth, making physical copies of tables is an expensive operation. Using `CLONE` makes the process simpler and cost-effective with the help of table clones.

What are clones anyway?

Clones are replicas of a source table at a given point in time. They have the same metadata as the source table: same schema, constraints, column descriptions, statistics, and partitioning. However, they behave as a separate table with a separate lineage or history. Any changes made to clones only affect the clone and not the source. Any changes that happen to the source during or after the cloning process also do not get reflected in the clone due to snapshot isolation. As of this writing, there are two types of clones: shallow or deep.

The `CLONE` command is currently only available on Databricks

Shallow Clones. A *shallow* (also known as Zero-Copy) clone only duplicates the metadata of the table being cloned; the data files of the table itself are not copied. This type of cloning does not create another physical copy of the data resulting in minimal storage costs. Shallow clones are inexpensive and can be extremely fast to create. These clones are not self-contained and **depend on the source from which they were cloned as the source of data**. If the files in the source that the clone depends on are removed, for example with `VACUUM`, a shallow clone may become unusable. Therefore, shallow clones are typically used for short-lived use cases such as testing and experimentation.

Deep Clones. Shallow clones are great for short-lived use cases, but some scenarios require a separate and independent copy of the table's data. A deep clone makes a full copy of the metadata and data files of the table being cloned. In that sense, it is similar in functionality to copying with a CTAS command (`CREATE TABLE.. AS… SELECT…`). But it is simpler to specify since it makes a faithful copy of the original table at the specified version and you don't need to re-specify partitioning, constraints, and other information as you have to do with CTAS. In addition, it is much **faster**, robust, and can work in an **incremental** manner against failures.

With deep clones, we copy additional metadata, such as your streaming application transactions and `COPY INTO` transactions, so you can continue your ETL applications exactly where it left off on a deep clone.

Where do clones help?

There are many scenarios where you need a copy of your datasets – for exploring, sharing, or testing ML models or analytical queries. Below are some example use cases.

Testing and experimentation with a production table. When users need to test a new version of their data pipeline they often have to rely on sample test datasets that are not representative of all the data in their production environment. Data teams may also want to experiment with various indexing techniques to improve the performance of queries against massive tables. These experiments and tests cannot be carried out in a production environment without risking production data processes and affecting users.

It can take many hours or even days, to spin up copies of your production tables for a test or a development environment. Add to that, the extra storage costs for your development environment to hold all the duplicated data – there is a large overhead in setting a test environment reflective of the production data. With a shallow clone, this is trivial:

```
%sql
-- Shallow clone table
CREATE TABLE delta.`/some/test/location` SHALLOW CLONE prod.events

%python
# Shallow clone table
DeltaTable.forName("spark", "prod.events").clone("/some/test/location", isShal
low=True)

%scala
// Shallow clone table
DeltaTable.forName("spark", "prod.events").clone("/some/test/location", isShal
low=true)
```

After creating a shallow clone of your table in a matter of seconds, you can start running a copy of your pipeline to test out your new code, or try optimizing your table in different dimensions to see how you can improve your query performance, and much more. These changes will only affect your shallow clone, not your original table.

Staging major changes to a production table. Sometimes, you may need to perform some major changes to your production table. These changes may consist of many steps, and you don't want other users to see the changes which you're making until you're done with all of your work. A shallow clone can help you out here:

```
%sql
-- Create a shallow clone
CREATE TABLE temp.staged_changes SHALLOW CLONE prod.events;

-- Test out deleting table from shallow clone table
DELETE FROM temp.staged_changes WHERE event_id is null;

-- Update shallow clone table
UPDATE temp.staged_changes SET change_date = current_date() WHERE change_date
is null;
...
-- Perform your verifications
```

Once you're happy with the results, you have two options. If no other change has been made to your source table, you can replace your source table with the clone. If changes have been made to your source table, you can merge the changes into your source table.

```
%sql
-- If no changes have been made to the source
REPLACE TABLE prod.events CLONE temp.staged_changes;

-- If the source table has changed
MERGE INTO prod.events USING temp.staged_changes
ON events.event_id <=> staged_changes.event_id
WHEN MATCHED THEN UPDATE SET *;
```

```
-- Drop the staged table
DROP TABLE temp.staged_changes;
```

Machine Learning result reproducibility. Training machine learning models is typically an iterative process. Throughout this process of optimizing the different parts of the model, data scientists need to assess the accuracy of the model against a fixed dataset. This is hard to do in a system where the data is constantly being loaded or updated. A snapshot of the data used to train and test the model is required. This snapshot allows the results of the ML model to be reproducible for testing or model governance purposes. We recommend leveraging Time Travel to run multiple experiments across a snapshot; an example of this in action can be seen in Machine Learning Data Lineage with MLflow and Delta Lake. Once you're happy with the results and would like to archive the data for later retrieval, for example, next Black Friday, you can use deep clones to simplify the archiving process. MLflow integrates really well with Delta Lake, and the auto logging feature (`mlflow.spark.autolog()`) will tell you, which version of the table was used to run a set of experiments.

```
# Run your ML workloads using Python and then
DeltaTable.forName(spark, "feature_store").cloneAtVersion(128, "fea
ture_store_bf2020")
```

Data Migration. A massive table may need to be moved to a new, dedicated bucket or storage system for performance, disaster recovery, or governance reasons. The original table will not receive new updates going forward and will be deactivated and removed at a future point in time. Deep clones make the copying of massive tables more robust and scalable.

```
%sql
-- Data migration using deep clone
CREATE TABLE delta.`zz://my-new-bucket/events` CLONE prod.events;
ALTER TABLE prod.events SET LOCATION 'zz://my-new-bucket/events';
```

With deep clones, since we copy your streaming application transactions and COPY INTO transactions, you can continue your ETL applications from **exactly where it left off after this migration**!

Data Sharing. It is common for users from different departments to look for data sets that they can use to enrich their analysis or models and you may want to share your data with them. But rather than setting up elaborate pipelines to move the data to yet another store, it is often easier and economical to create a copy of the relevant data set for users to explore and test the data to see if it is a fit for their needs without affecting your own production systems. Here deep clones significantly simplify this process.

```
%sql
-- The following code can be scheduled to run at your convenience
```

```
-- for data sharing
CREATE OR REPLACE TABLE data_science.events CLONE prod.events;
```

Data Archiving. For regulatory or archiving purposes all data in a table needs to be preserved for a certain number of years, while the active table retains data for a few months. If you want your data to be updated as soon as possible, but however you have a requirement to keep data for several years, storing this data in a single table and performing time travel may become prohibitively expensive. In this case, archiving your data in a daily, weekly, or monthly manner is a better solution. The incremental cloning capability of deep clones will really help you here.

```
%sql
-- The following code can be scheduled to run at your convenience
-- for data archiving
CREATE OR REPLACE TABLE archive.events CLONE prod.events;
```

Note that this table will have an **independent** history compared to the source table, therefore time travel queries on the source table and the clone may return different results based on your frequency of archiving.

For more on clones

For more information on clones:

- Delta CLONE Language Manual
- How to Easily Clone Your Delta Lake Data Tables with Databricks
- Attack of the Delta Clones (Against Disaster Recovery Availability Complexity)
- We will have an Operational scenarios chapter for more examples

Summary

In this chapter, we began by covering the three simple steps for you to get started on Delta Lake: using `--packages`, building standalone applications via GitHub and maven, and using Databricks community edition. We provided the basic operations of Delta Lake which look suspiciously similar to Spark basic operations - because they are! We provided an intermediate-level primer on how the Delta Lake transaction log works and how it can provide ACID transactions to provide *data reliability*. Finally, we described some of the Delta Lake table utility commands.

Because the Delta transaction log uses MVCC, there can be many files associated with different versions of the Delta table. Fortunately, the table utility commands simplify this process. But there are distinct feature advantages such as Delta time travel (or data versioning), which we cover in our next chapter, Chapter 3: Time Travel with Delta.

CHAPTER 2

Time Travel with Delta Lake

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

Introduction

Data engineering pipelines often go awry, especially when ingesting "dirty" data from external systems. However, in a traditional data lake design, it is hard to undo updates that added objects into a table. In addition, it is challenging to audit data changes which are critical, both in terms of data compliance as well as simple debugging, to understand how data has changed over time. Other data workloads, such as machine learning training, require faithfully reproducing an old version of the data (e.g., to compare a new and old training algorithm on the same data). For example, if some upstream pipeline modifies the source data, data scientists are often caught unaware by such upstream data changes and hence struggle to reproduce their experiments.

All of these issues created significant challenges for data engineers and data scientists before Delta Lake, requiring them to design complex remediations to data pipeline errors or to haphazardly duplicate datasets. Through this chapter, we will explore the

capabilities of Delta Lakes that allow data practitioners to go back in time and solve for these challenges. It is uncertain if time travel to the past is physically possible but data practitioners can time travel programmatically with Delta Lake. Delta Lake allows automatic versioning of all data stored in the data lake and we can time travel to any version. It also allows us to create a copy of an existing Delta table at a specific version using the clone command to help with a few time travel use cases. To understand these functionalities and how time travel works, we first need to unpack the file structure of Delta tables and deep dive into each of the components.

Under the hood of a Delta Table

In chapter 2, one of our sections was to focus on unpacking the transaction log. But recall that the transaction log is just one part of the Delta table. In this section, we will focus on the components that make up the Delta table.

The Delta Directory

A Delta table is stored within a directory and is composed of the file types shown in Figure 2-1.

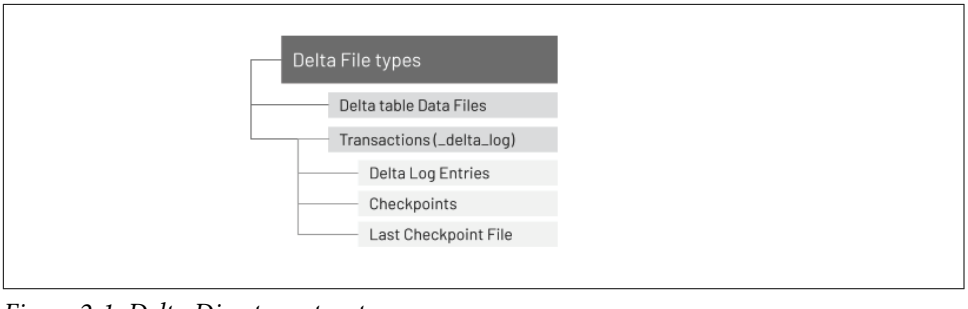


Figure 2-1. Delta Directory structure

As an example, we have a delta table named `customer_data` located in the filePath: `dbfs:/user/hive/warehouse/deltaguide.db`

To visualize the delta file system, see below a snippet from our example. You can also follow along with us by using the notebooks present here.

Root of the Delta directory : This is the hive warehouse location where we created a delta table and using the command below, we can explore the contents within the delta directory. For the full output, please refer to this location.

```
%fs ls dbfs:/user/hive/warehouse/deltaguide.db/customer_data
```

Table 2-1. Delta Root Directory

path	name	size

dbfs:/dbfs:/..filePath../customer_data/_delta_log/	_delta_log/	0
dbfs:/dbfs:/..filePath../customer_data/part-00000-41…-c001.snappy.parquet	part-00000-41...-c001.snappy.parquet	542929260
dbfs:/dbfs:/..filePath../customer_data/part-00000-45...-c002.snappy.parquet	part-00000-45...-c002.snappy.parquet	461126371
dbfs:/dbfs:/..filePath../customer_data/part-00000-62...-c000.snappy.parquet	part-00000-62...-c000.snappy.parquet	540468009
dbfs:/dbfs:/..filePath../customer_data/part-00001-40...-c002.snappy.parquet	part-00001-40...-c002.snappy.parquet	541858629
dbfs:/dbfs:/..filePath../customer_data/part-00001-a7...-c001.snappy.parquet	part-00001-a7...-c001.snappy.parquet	542859315
dbfs:/dbfs:/..filePath../customer_data/part-00001-c3...-c000.snappy.parquet	part-00001-c3...-c000.snappy.parquet	541186721

Delta Logs Directory

When a user creates a Delta Lake table, that table's transaction log is automatically created in the _delta_log subdirectory. As he or she makes changes to that table, those changes are recorded as ordered, atomic commits in the transaction log. Each commit is written out as a JSON file, starting with 000000.json. Additional changes to the table generate subsequent JSON files in ascending numerical order so that the next commit is written out as 000001.json, the following as 000002.json, and so on.

Figure 2-2 shows the logical diagram of Delta Transaction Log Protocol.

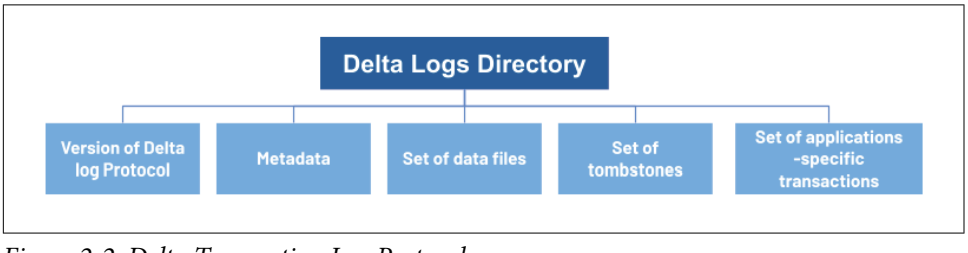


Figure 2-2. Delta Transaction Log Protocol

The state of a table at a given version is called a *snapshot* and includes the following properties:

- **Version of the Delta log protocol :** This is required to correctly read or write the table
- **Metadata** of the table (e.g., the schema, a unique identifier, partition columns, and other configuration properties)
- **Set of files** present in the table, along with metadata about those files
- **Set of tombstones** for files that were recently deleted

- **Set of applications-specific transactions** that have been successfully committed to the table

We will explore each of the properties in the upcoming sections and to provide you a mental mapping of how this actually looks from a file system perspective, here is a snippet.

```
%fs ls dbfs:/fil ePath/customer_t2/_delta_log
```

Table 2-2. Delta Logs Directory

path	name	size
dbfs:/.../customer_t2/_delta_log/.s3-optimization-0	.s3-optimization-0	0
dbfs:/.../customer_t2/_delta_log/.s3-optimization-1	.s3-optimization-1	0
dbfs:/.../customer_t2/_delta_log/.s3-optimization-2	.s3-optimization-2	0
dbfs:/.../customer_t2/_delta_log/ 00000000000000000010.checkpoint.parquet	00000000000000000010.checkpoint.parquet	120950
: : :	: : :	: : :
dbfs:/.../customer_t2/_delta_log/ 00000000000000000020.checkpoint.parquet	00000000000000000020.checkpoint.parquet	94341
dbfs:/.../customer_t2/_delta_log/00000000000000000020.crc	00000000000000000020.crc	97
dbfs:/.../customer_t2/_delta_log/00000000000000000020.json	00000000000000000020.json	347291
dbfs:/.../customer_t2/_delta_log/00000000000000000021.crc	00000000000000000021.crc	95
dbfs:/.../customer_t2/_delta_log/00000000000000000021.json	00000000000000000021.json	41461
dbfs:/.../customer_t2/_delta_log/_last_checkpoint	_last_checkpoint	26

As previously noted, Delta Lake is an open-source storage layer that runs on top of your existing data lake and is fully compatible with Apache Spark APIs. Delta Lake uses versioned Parquet files to store data in your cloud storage, enabling Delta Lake to leverage the efficient compression and encoding schemes that are native to Parquet. Apart from the versions, Delta Lake also stores a transaction log to keep track of all the commits made to the table or blob store directory to provide ACID transactions.

You can use your favorite Apache Spark APIs to read and write data with Delta Lake. As noted in the previous chapter, to create a Delta table, write a DataFrame out in the `delta` format.

```
%python
# Generate Spark DataFrame
data = spark.range(0, 5)

# Write the table in parquet format
data.write.format("parquet").save("/table_pq")
```

```
# Write the table in delta format
data.write.format("delta").save("/table_delta")
```

```
%scala
// Generate Spark DataFrame
val data = spark.range(0, 5)

// Write the table in parquet format
data.write.format("parquet").save("/table_pq")

// Write the table in delta format
data.write.format("delta").save("/table_delta")
```

Once you write the files in delta, you will notice that a number of files are created under a folder in the format recognized by Spark as a Parquet table and/or stored in a metastore (e.g. Hive, Glue, etc.).

Note, the files of a metastore-defined table are stored under the default `hive/warehouse` location though you have an option to specify a location of your choice at the time of writing delta files as noted in the previous example.

To review the underlying file structure, run the following command from your

```
-table>
```

The following is the shell command for the parquet table previously generated.

```
%sh ls -R /dbfs/table_pq/

/dbfs/table_pq/:
part-00000-775edc03-7c05-4190-a964-fcdfc0428014-c000.snappy.parquet
part-00001-94940738-a4e6-4b4b-b30f-797a76b32087-c000.snappy.parquet
part-00003-0e5d49e6-4dd0-4746-8944-f64cba9df97c-c000.snappy.parquet
part-00004-bed772f9-1045-4e3a-8048-e179096e3c25-c000.snappy.parquet
part-00006-e28f9fa4-516b-4cf6-acdd-22d70e8841f2-c000.snappy.parquet
part-00007-5276d994-b9ca-4da8-b594-4bbfafbd7dcb-c000.snappy.parquet
```

The following is the shell command for the Delta table previously generated.

```
%sh ls -R /dbfs/table_delta/

/dbfs/table_delta/:

_delta_log
part-00000-775edc03-7c05-4190-a964-fcdfc0428014-c000.snappy.parquet
part-00001-94940738-a4e6-4b4b-b30f-797a76b32087-c000.snappy.parquet
part-00003-0e5d49e6-4dd0-4746-8944-f64cba9df97c-c000.snappy.parquet
part-00004-bed772f9-1045-4e3a-8048-e179096e3c25-c000.snappy.parquet
part-00006-e28f9fa4-516b-4cf6-acdd-22d70e8841f2-c000.snappy.parquet
```

```
part-00007-5276d994-b9ca-4da8-b594-4bbfafbd7dcb-c000.snappy.parquet

/dbfs/table_delta/_delta_log:
00000000000000000000.crc
00000000000000000000.json
```

Majority of Data Practitioners are familiar with Parquet and its structure, let's simplify the understanding of Delta by comparing it to Parquet.

What is the difference between the Parquet and Delta tables?

The only difference between the Parquet and Delta tables is the `_delta_log` folder which is the Delta transaction log (more on this in Chapter 2).

What is the same between the Parquet and Delta tables?

In both Parquet and Delta tables, the data itself is in the form of snappy compressed Parquet part files, which is the common format of Spark tables.

- Apache Parquet is a columnar storage format that is optimized for BI-type queries (group by, aggregates, joins, etc.). Because of this, it is the default storage format for Apache Spark when it writes its data files to storage.
- Snappy compression was developed by Google to provide high compression/decompression speeds with reasonable compression. By default, Spark tables are snappy compressed Parquet files.
- By default, the reference implementation of Delta lake stores data files in directories named after partition values for data in that file (i.e. `part1=value1/part2=value2/...`) that is why we see part files and a suffix in the name as a globally unique identifier to ensure uniqueness of each file. This is especially important because each directory in cloud object stores are not actual folders but logical representations of folders. For more information on the implications, read more at How to list and delete files faster in Databricks.

The files of a Delta table

ny time data is modified in a Delta table, new files are created as a Version which is a snapshot of the delta table at that specific time and is a result of a set of actions that were performed by the user.

Version 0: Table creation

Let's start by looking at the Delta table created in the previous section by running the following commands to review the `path` column of the `add` metadata.

```
%python
# Read first transaction
j0 = spark.read.json("/table_delta/_delta_log/00000000000000000000.json")
```

```
# Review Add Information
j0.select("add.path").where("add is not null").show(20, False)
```

```
%scala
// Read first transaction
val j0 = spark.read.json("/table_delta/_delta_log/00000000000000000000.json")

// Review Add Information
j0.select("add.path").where("add is not null").show(20, False)
```

Metadata in the transaction log include all of the files that make up the table version (version 0). The output of the preceding commands is:

```
+-----------------------------------------------------------------------+
|path                                                                   |
+-----------------------------------------------------------------------+
|part-00000-775edc03-7c05-4190-a964-fcdfc0428014-c000.snappy.parquet|
|part-00001-94940738-a4e6-4b4b-b30f-797a76b32087-c000.snappy.parquet|
|part-00003-0e5d49e6-4dd0-4746-8944-f64cba9df97c-c000.snappy.parquet|
|part-00004-bed772f9-1045-4e3a-8048-e179096e3c25-c000.snappy.parquet|
|part-00006-e28f9fa4-516b-4cf6-acdd-22d70e8841f2-c000.snappy.parquet|
|part-00007-5276d994-b9ca-4da8-b594-4bbfafbd7dcb-c000.snappy.parquet|
+-----------------------------------------------------------------------+
```

Notice how this matches the file listing you see when you run the `ls` command. While this file listing is pretty fast for small amounts of data, as noted in Chapter 2, `listFrom` can be quite slow on cloud object storage especially for petabyte-scale data lakes.

When reviewing the table history (see Table 3-2), let's focus on the operationMetrics for version 0 (table creation).

```
%sql
-- Describe table history using file path
DESCRIBE HISTORY delta.`/table_delta`
```

Table 3-x transposes the values of the preceding figure.

Column Name	Value
version	0
timestamp	userID
userName	vini[dot]jaiswal[at]databricks.com
operation	WRITE
operationParameters	{"mode": "ErrorIfExists", "partitionBy": "[]"}
job	null
notebook	{"notebookId": "9342327"}
clusterId	
readVersion	null
isolationLevel	WriteSerializable

isBlindAppend	true
operationMetrics	{"numFiles": "6", "numOutputBytes": "2713", "numOutputRows": "5"}
userMetadata	null

Of particular interest in Table 3-x is the **operationMetrics** column for table version 0.

```
{
"numFiles": "6",
"numOutputBytes": "2713",
"numOutputRows": "5"
}
```

Notice how `numFiles` corresponds to the six files listed in the previous `add.path` query as well as the file listing (`ls`).

Version 1: Appending data

What happens when we add new data to this table? The following command will add 4 new rows to our table.

```
%python
# Add 4 new rows of data to our Delta table
data = spark.range(6, 10)
data.write.format("delta").mode("append").save("/table_delta")

%scala
// Add 4 new rows of data to our Delta table
val data = spark.range(6, 10)
data.write.format("delta").mode("append").save("/table_delta")
```

We're now going to go over how to review data, file system, added files, etc.

Review data. You can confirm there are a total number of 9 rows in the table by running the following command.

```
%python
spark.read.format("delta").load("/table_delta").count()

%scala
spark.read.format("delta").load("/table_delta").count()
```

Review file system. But what does the underlying file system look like? When we rerun our file listing, as expected, there are more files and a new JSON and CRC file within `_delta_log` corresponding to a new version of the table.

```
%sh ls -R /dbfs/table_delta/

/dbfs/table_delta/:
_delta_log
part-00000-0267038a-f818-4823-8261-364fd7401501-c000.snappy.parquet
part-00000-775edc03-7c05-4190-a964-fcdfc0428014-c000.snappy.parquet
part-00001-94940738-a4e6-4b4b-b30f-797a76b32087-c000.snappy.parquet
```

```
part-00001-cd3a1a49-0a0a-4284-8a1b-283d83590dff-c000.snappy.parquet
part-00003-0e5d49e6-4dd0-4746-8944-f64cba9df97c-c000.snappy.parquet
part-00003-56bc7589-6266-4ce3-9515-62caf9af9109-c000.snappy.parquet
part-00004-bed772f9-1045-4e3a-8048-e179096e3c25-c000.snappy.parquet
part-00005-3d298fe6-8795-4558-92c7-70e0520a3d47-c000.snappy.parquet
part-00006-e28f9fa4-516b-4cf6-acdd-22d70e8841f2-c000.snappy.parquet
part-00007-5276d994-b9ca-4da8-b594-4bbfafbd7dcb-c000.snappy.parquet
part-00007-60de290b-7147-4ec4-b535-ca9abf4ce2d1-c000.snappy.parquet

/dbfs/table_delta/_delta_log:
00000000000000000000.crc
00000000000000000000.json
00000000000000000001.crc
00000000000000000001.json
```

Review added files. Let's follow up by looking at the Delta table version 1 by running the following commands to review the `path` column of the `add` metadata.

```
%python
# Read version 1
j1 = spark.read.json("/table_delta/_delta_log/00000000000000000001.json")

# Review Add Information
j1.select("add.path").where("add is not null").show(20, False)
```

```
%scala
// Read version 1
j1 = spark.read.json("/table_delta/_delta_log/00000000000000000001.json")

// Review Add Information
j1.select("add.path").where("add is not null").show(20, False)
```

Metadata in the transaction log include all of the files that make up table version 1. The output of the preceding commands is:

```
+-----------------------------------------------------------------------+
|path                                                                   |
+-----------------------------------------------------------------------+
|part-00000-0267038a-f818-4823-8261-364fd7401501-c000.snappy.parquet|
|part-00001-cd3a1a49-0a0a-4284-8a1b-283d83590dff-c000.snappy.parquet|
|part-00003-56bc7589-6266-4ce3-9515-62caf9af9109-c000.snappy.parquet|
|part-00005-3d298fe6-8795-4558-92c7-70e0520a3d47-c000.snappy.parquet|
|part-00007-60de290b-7147-4ec4-b535-ca9abf4ce2d1-c000.snappy.parquet|
+-----------------------------------------------------------------------+
```

Interestingly, this output notes that only 5 files were needed to hold the 4 new rows of data; recall there were 6 files for the initial 5 rows of data.

Review table history. This can be confirmed by reviewing the table history, let's focus on the operationMetrics for version 1 (table creation).

```
%sql
-- Describe table history using file path
DESCRIBE HISTORY delta.`/table_delta`
```

Below is the value of the **operationMetrics** column of version 1 which confirmed in fact there are only 5 files with the number of output rows being 4.

```
{
"numFiles": "5",
"numOutputBytes": "2232",
"numOutputRows": "4"
}
```

Notice how `numFiles` corresponds to the five files listed in the previous `add.path` query as well as the file listing (`ls`).

Query added files. You can further validate this by running the following query

```
%sql
-- Run this statement first as Delta will do a format check
SET spark.databricks.delta.formatCheck.enabled=false

%python
delta_path = "/table_delta/"

# Files listed in add.path metadata
files = [delta_path + "part-00000-0267038a-f818-4823-8261-364fd7401501-
c000.snappy.parquet",
         delta_path + "part-00001-cd3a1a49-0a0a-4284-8a1b-283d83590dff-
c000.snappy.parquet",
         delta_path + "part-00003-56bc7589-6266-4ce3-9515-62caf9af9109-
c000.snappy.parquet",
         delta_path + "part-00005-3d298fe6-8795-4558-92c7-70e0520a3d47-
c000.snappy.parquet",
         delta_path + "part-00007-60de290b-7147-4ec4-b535-ca9abf4ce2d1-
c000.snappy.parquet"]

# Show values stored in these files
spark.read.format("parquet").load(files).show()

# Results
+---+
| id|
+---+
|  8|
|  6|
|  7|
|  9|
+---+
```

As noted, these five files correspond to the four rows (id values 6, 7, 8, and 9) added to our Delta table.

Version 2: Deleting data

What happens when we remove data from this table? The following command will `DELETE` some of the values from our Delta table. Note, we will dive deeper into data modifications such as delete, update, and merge in Chapter 5: Data Modifications in Delta tables. But this example is an interesting showcase of what is happening in the file system.

```
%sql
-- Delete from Delta table where id <= 2
DELETE FROM delta.`/table_delta` WHERE id <= 2
```

Review data. You can confirm there are a total number of 6 rows (the 3 values of 0, 1, and 2 were removed) in the table by running the following command.

```
%python
spark.read.format("delta").load("/table_delta").count()

%scala
spark.read.format("delta").load("/table_delta").count()
```

Review file system. But what does the underlying file system look like? Let's re-run our file listing and as expected, there are more files and a new JSON and CRC file within `_delta_log` corresponding to a new version of the table.

```
%sh ls -R /dbfs/table_delta/

/dbfs/table_delta/:
_delta_log
part-00000-0267038a-f818-4823-8261-364fd7401501-c000.snappy.parquet
part-00000-74e6c7ea-7321-44f0-9fb6-55257677cb1f-c000.snappy.parquet
part-00000-775edc03-7c05-4190-a964-fcdfc0428014-c000.snappy.parquet
part-00001-94940738-a4e6-4b4b-b30f-797a76b32087-c000.snappy.parquet
part-00001-cd3a1a49-0a0a-4284-8a1b-283d83590dff-c000.snappy.parquet
part-00003-0e5d49e6-4dd0-4746-8944-f64cba9df97c-c000.snappy.parquet
part-00003-56bc7589-6266-4ce3-9515-62caf9af9109-c000.snappy.parquet
part-00004-bed772f9-1045-4e3a-8048-e179096e3c25-c000.snappy.parquet
part-00005-3d298fe6-8795-4558-92c7-70e0520a3d47-c000.snappy.parquet
part-00006-e28f9fa4-516b-4cf6-acdd-22d70e8841f2-c000.snappy.parquet
part-00007-5276d994-b9ca-4da8-b594-4bbfafbd7dcb-c000.snappy.parquet
part-00007-60de290b-7147-4ec4-b535-ca9abf4ce2d1-c000.snappy.parquet

/dbfs/table_delta/_delta_log:
00000000000000000000.crc
00000000000000000000.json
00000000000000000001.crc
00000000000000000001.json
00000000000000000002.crc
00000000000000000002.json
```

As expected, there is a new transaction log file (`000...00002.json`) but now there are 12 files in our Delta table folder after we deleted three rows (previously there were 11 files). Let's review Delta table version 2 by running the following commands to review the `path` column of the `add` and `remove` metadata.

Review removed files. There were three files *removed* from our table; the following query reads the version 2 transaction log and specifically extracts the path of *removed* files.

```
%python
# Remove information
j2.select("remove.path").where("remove is not null").show(20, False)

%scala
// Remove information
j2.select("remove.path").where("remove is not null").show(20, False)

# Result
+-----------------------------------------------------------------------+
|path                                                                   |
+-----------------------------------------------------------------------+
|part-00004-bed772f9-1045-4e3a-8048-e179096e3c25-c000.snappy.parquet|
|part-00003-0e5d49e6-4dd0-4746-8944-f64cba9df97c-c000.snappy.parquet|
|part-00001-94940738-a4e6-4b4b-b30f-797a76b32087-c000.snappy.parquet|
+-----------------------------------------------------------------------+
```

It is important to note that in the transaction log the deletion is a tombstone, i.e. we have not deleted the files. That is, these files are identified as *removed* so that when you query the table, Delta will **not** include the `remove` files.

Review table history. This can be confirmed by reviewing the table history, let's focus on the operationMetrics for version 2 (table creation).

```
%sql
-- Describe table history using file path
DESCRIBE HISTORY delta.`/table_delta`
```

Below is the value of the **operationMetrics** column of version 2 which confirmed in fact that 3 files were removed, 1 file was added, and 3 rows were deleted.

```
{
"numRemovedFiles": "3",
"numDeletedRows": "3",
"numAddedFiles": "1",
"numCopiedRows": "0"
}
```

Query removed files. As stated earlier, the files may be *removed* in the transaction log, but they are not removed from the file system. The Delta data files are created in this fashion to correspond to the concept of multi-version concurrency control (MVCC).

For example, as seen in Figure 3-x, if user 1 is querying version 1 of the table while user 2 is deleting values (0, 1, 2) that creates table version 2. if the *data* files that represent version 1 (that contained the values 0, 1, and 2), user 1's query would irrevocably fail.

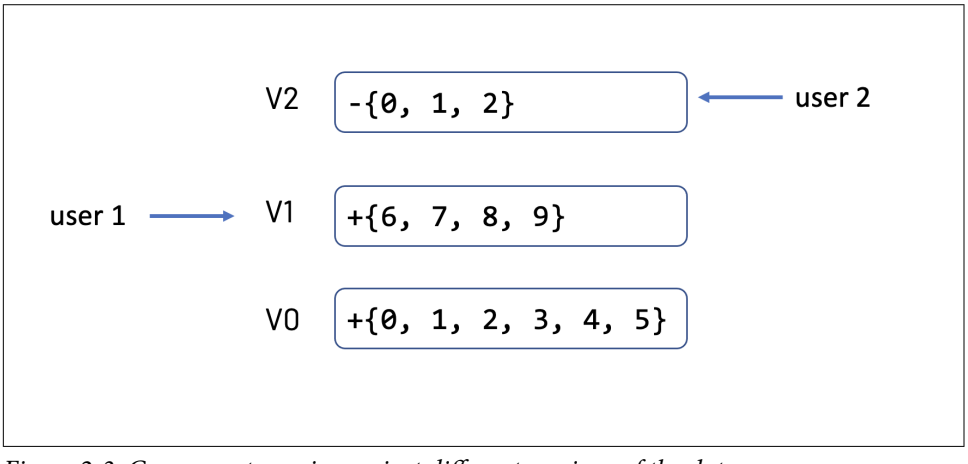


Figure 2-3. Concurrent queries against different versions of the data

This is especially important for long-running queries on data lakes where the queries and operations may take a long time. It is important to note that by default user 1 had executed the read query *before* the version 2 of the table was committed. If at a later time, as seen in Figure 3-x, another user (user 3) queries the same table, then by default when they read the data, the three files representing values {0, 1, 2} would not be included and they would only be returned six rows.

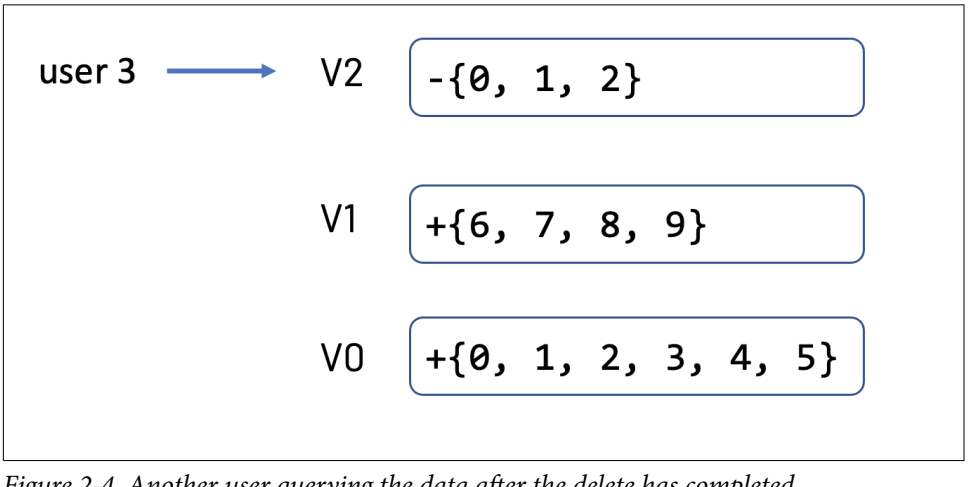


Figure 2-4. Another user querying the data after the delete has completed

To further validate the data is still there, run the following query; the three files (`files`) correspond to the three *removed* files in the previous section.

```
%sql
-- Run this statement first as Delta will do a format check
SET spark.databricks.delta.formatCheck.enabled=false

%python
delta_path = "/table_delta/"

# Files listed in add.path metadata
files = [delta_path + "part-00004-bed772f9-1045-4e3a-8048-e179096e3c25-
c000.snappy.parquet",
         delta_path + "part-00003-0e5d49e6-4dd0-4746-8944-f64cba9df97c-
c000.snappy.parquet",
         delta_path + "part-00001-94940738-a4e6-4b4b-b30f-797a76b32087-
c000.snappy.parquet"]

# Show values stored in these files
spark.read.format("parquet").load(files).show()

# Results
+---+
| id|
+---+
|  2|
|  1|
|  0|
+---+
```

Review added files. Even though we had deleted three rows, we've also added a file in the file system as recorded in the transaction log.

```
%python
# Read version 2
j2 = spark.read.json("/table_delta/_delta_log/00000000000000000002.json")

>#  Review Add Information
j2.select("add.path").where("add is not null").show(20, False)

%scala
// Read version 2
j2 = spark.read.json("/table_delta/_delta_log/00000000000000000002.json")

// Review Add Information
j2.select("add.path").where("add is not null").show(20, False)
```

To reiterate, even though we deleted three rows, as noted in Add Information there is an additional file!

```
+----------------------------------------------------------------------+
|path                                                                  |
+----------------------------------------------------------------------+
```

```
|part-00000-74e6c7ea-7321-44f0-9fb6-55257677cb1f-c000.snappy.parquet|
+--------------------------------------------------------------------+
```

In fact, there is nothing in this file if you were to read it using the query below as the description of this column is apt.

```
%python
# Read the add file from version 2
spark.read.format("parquet").load("/table_delta/
part-00000-74e6c7ea-7321-44f0-9fb6-55257677cb1f-c000.snappy.parquet").show()

%scala
// Read the add file from version 2
spark.read.format("parquet").load("/table_delta/
part-00000-74e6c7ea-7321-44f0-9fb6-55257677cb1f-c000.snappy.parquet").show()

# Result
+---+
| id|
+---+
+---+
```

During our delete operation, no new rows or files were added to the table as was expected for this small example.

Why are files added during a delete?. For larger datasets, it is common that not all the rows will be removed from a file.

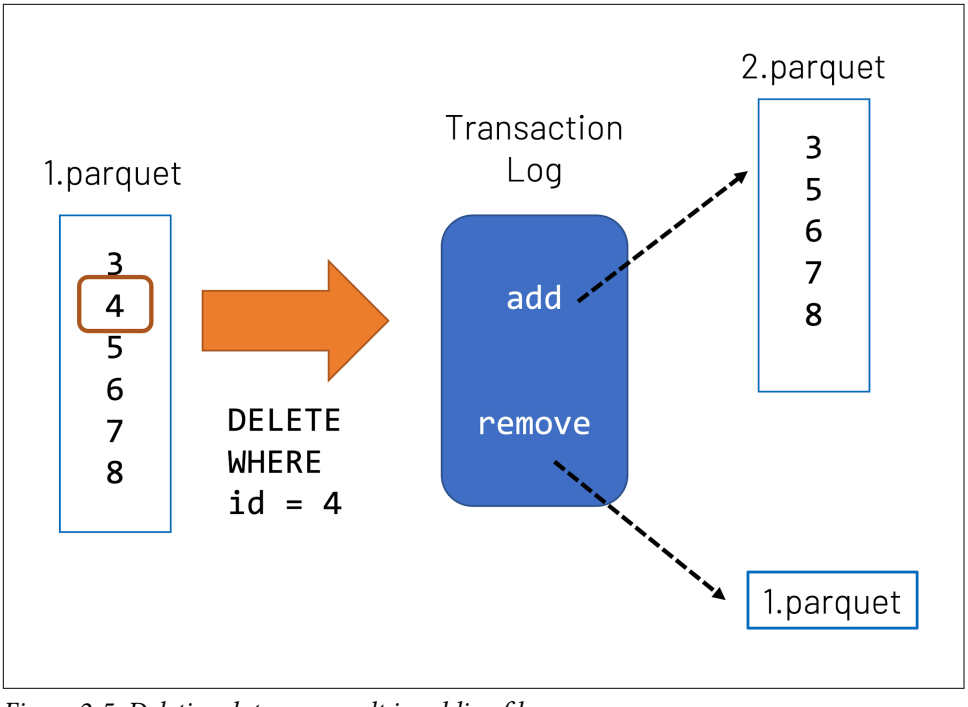


Figure 2-5. Deleting data can result in adding files

Extending the current scenario, let's say that the values `{3-8}` are stored in `1.parquet` and you run a `DELETE` statement to remove the values `id = 4`. In this example:

- Within the transaction log, the `1.parquet` file is added to the `remove` column.
- But if we stop here, we would be removing not only `{4}` but also `{3,5-8}`.
- Therefore, a new file will be created - in this case the `2.parquet` file - that contains the values that were not deleted `{3,5-8}`.

To demonstrate this with this scenario, let's compact our current Delta table. If you're running this on Databricks, you can use the `OPTIMIZE` command; more on this command in Chapter 12: Performance Tuning.

```
%sql
-- Optimize our Delta table
OPTIMIZE delta.`/table_delta/`
```

This command is used to tune the performance of your Delta tables and it also has the benefit of compacting your files. In this case, this means that instead of 11 separate files, representing our 9 values, now we only have one file. The optimize operation itself is the third operation per the table history.

```
%sql
-- Table history
DESCRIBE HISTORY delta.`/table_delta/`

-- Abridged results
+-------+---------+---------------------------+
|version|operation|operationMetrics           |
+-------+---------+---------------------------+
|3      |OPTIMIZE |[numRemovedFiles -> 9,  ... |
|2      |DELETE   |[numRemovedFiles -> 3,  ... |
|1      |WRITE    |[numFiles -> 5,  ...        |
|0      |WRITE    |[numFiles -> 6,  ...        |
+-------+---------+---------------------------+
```

The results from running the `OPTIMIZE` command can be seen below (which is also recorded in the `operationMetrics` column within table history.

```
path: /table_delta/

Metrics:
 {
"numFilesAdded": 1,
"numFilesRemoved": 9,
"filesAdded": {"min": 507, "max": 507, "avg": 507, "totalFiles": 1, "total-
Size": 507},
 "filesRemoved": {"min": 308, "max": 481, "avg": 423, "totalFiles": 9, "total-
Size": 3810},
 "partitionsOptimized": 0,
"zOrderStats": null,
"numBatches": 1
}
```

The two metrics most interesting for this scenario are `numFilesAdded` and `numFiles Removed` which denote that now there is only 1 file that contains all the values and 9 files were removed.

The following query to validate the files that were removed.

```
%python
# Read version 3
j3 = spark.read.json("/table_delta/_delta_log/00000000000000000003.json")

# Remove Information
j3.select("remove.path").where("add is not null").show(20, False)

%scala
// Read version 3
val j3 = spark.read.json("/table_delta/_delta_log/00000000000000000003.json")

// Remove Information
j3.select("remove.path").where("add is not null").show(20, False)

-- Results
+------------------------------------------------------------------------+
```

```
|path                                                                 |
+---------------------------------------------------------------------+
|part-00000-775edc03-7c05-4190-a964-fcdfc0428014-c000.snappy.parquet|
|part-00000-0267038a-f818-4823-8261-364fd7401501-c000.snappy.parquet|
|part-00000-74e6c7ea-7321-44f0-9fb6-55257677cb1f-c000.snappy.parquet|
|part-00006-e28f9fa4-516b-4cf6-acdd-22d70e8841f2-c000.snappy.parquet|
|part-00007-5276d994-b9ca-4da8-b594-4bbfafbd7dcb-c000.snappy.parquet|
|part-00001-cd3a1a49-0a0a-4284-8a1b-283d83590dff-c000.snappy.parquet|
|part-00003-56bc7589-6266-4ce3-9515-62caf9af9109-c000.snappy.parquet|
|part-00005-3d298fe6-8795-4558-92c7-70e0520a3d47-c000.snappy.parquet|
|part-00007-60de290b-7147-4ec4-b535-ca9abf4ce2d1-c000.snappy.parquet|
+---------------------------------------------------------------------+
```

The following query allows you to validate the files that were added.

```
%python
# Add Information
j3.select("add.path").where("add is not null").show(20, False)

%scala
// Add Information
j3.select("add.path").where("add is not null").show(20, False)

-- Results
+---------------------------------------------------------------------+
|path                                                                 |
+---------------------------------------------------------------------+
|part-00000-f661f932-f54f-4b38-8ed3-51f7770d3e55-c000.snappy.parquet|
+---------------------------------------------------------------------+
```

You can query the file and validate that it contains all the expected values `{3-8}`.

```
%sql
-- Temporarily disable the Delta format check
SET spark.databricks.delta.formatCheck.enabled=false

%python / %scala
# Python or Scala
spark.read.parquet("/table_delta/part-00000-f661f932-
f54f-4b38-8ed3-51f7770d3e55-c000.snappy.parquet").show()

-- Results
+---+
| id|
+---+
|  4|
|  8|
|  3|
|  6|
|  9|
|  7|
+---+
```

So now that all of the data is in one file, what happens when we delete a value in our table?

```
%sql
DELETE FROM delta.`/tmp/delta/` WHERE id = 4
```

As we know this is version 4 of our table history, let's review the transaction log using the following query.

```
%python
# Read version 4
j4 = spark.read.json("/table_delta/_delta_log/00000000000000000004.json")

# Remove Information
j4.select("remove.path").where("add is not null").show(20, False)
```

```
%scala
// Read version 4
val j4 = spark.read.json("/table_delta/_delta_log/00000000000000000004.json")

// Remove Information
j4.select("remove.path").where("add is not null").show(20, False)
```

```
--  Results
+----------------------------------------------------------------------+
|path                                                                  |
+----------------------------------------------------------------------+
|part-00000-f661f932-f54f-4b38-8ed3-51f7770d3e55-c000.snappy.parquet|
+----------------------------------------------------------------------+
```

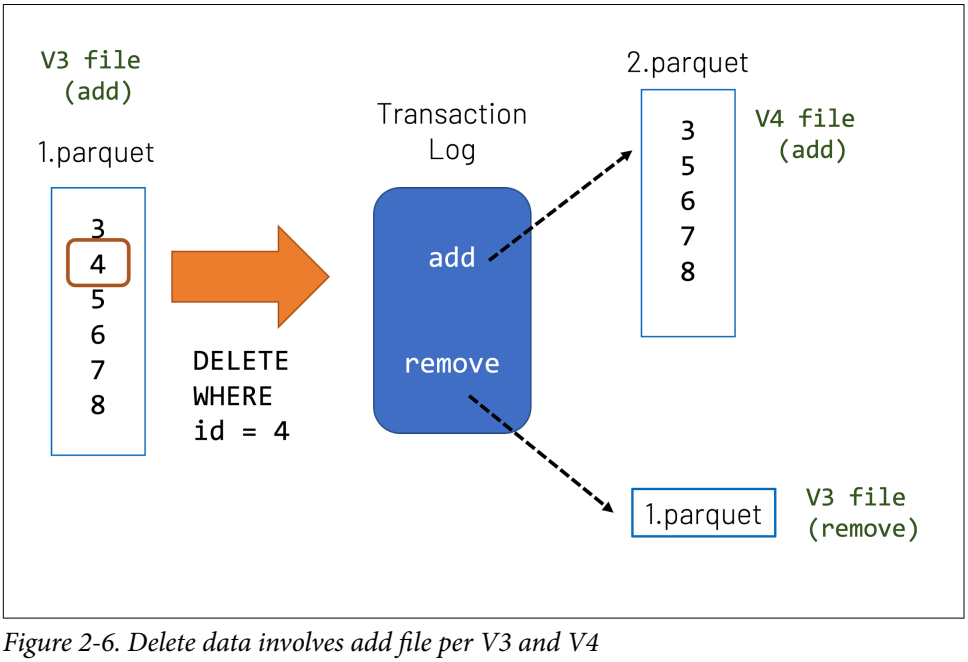


Figure 2-6. Delete data involves add file per V3 and V4

Notice how the `remove.path` value is the same file as the preceding version 3 file. But as you can see from the following query, a new file is created.

```
%python
# Add Information
j4.select("add.path").where("add is not null").show(20, False)

%scala
// Add Information
j4.select("add.path").where("add is not null").show(20, False)

-- Result
+------------------------------------------------------------------------+
|path                                                                    |
+------------------------------------------------------------------------+
|part-00000-68bf51ea-6e00-4591-b267-62fcd36602ed-c000.snappy.parquet|
+------------------------------------------------------------------------+
```

If you query this file, observe that it contains only the expected values of `{3, 5-8}`.

```
%python
# Read the V4 added file
spark.read.parquet("/table_delta/part-00000-68bf51ea-6e00-4591-
b267-62fcd36602ed-c000.snappy.parquet").show()

%python
// Read the V4 added file
spark.read.parquet("/table_delta/part-00000-68bf51ea-6e00-4591-
b267-62fcd36602ed-c000.snappy.parquet").show()

-- Result
+---+
| id|
+---+
|  8|
|  3|
|  6|
|  9|
|  7|
+---+
```

To reiterate, instead of deleting the value 4 from the original file, instead Delta is creating a new file with the values not deleted (in this case) and keeping the old file in place.

Let's go back in time. What you're seeing here is multiversion concurrency control (MVCC) in action where there are two separate files associated with two different table versions. But how about if you do want to query the previous data? Well, this is an excellent segue to Delta time travel options.

Time Travel

In Chapter 1 we unpacked what was happening with the transaction log to provide ACID transactional semantics. In the previous section, we unpacked what was happening with the data files in relation to the transaction log in light of Delta transactional protocol with particular focus on multiversion concurrency control (MVCC). An interesting byproduct of MVCC is that all of the files from previous transactions still reside and are accessible in storage. That is, the positive byproduct of the implementation of MVCC in Delta is *data versioning* or *time travel*.

From a high-level, Delta automatically versions the big data that you store in your data lake, and you can access any historical version of that data. This temporal data management simplifies your data pipeline by making it easy to audit, roll back data in case of accidental bad writes or deletes, and reproduce experiments and reports. You can standardize on a clean, centralized, versioned big data repository in your own cloud storage for your analytics.

Common Challenges with Changing Data

Data versioning is an important tool to ensure data reliability to address common challenges with changing data.

Audit data changes
: Auditing data changes is critical from both in terms of data compliance as well as simple debugging to understand how data has changed over time. Organizations moving from traditional data systems to big data technologies and the cloud struggle in such scenarios.

Reproduce experiments & reports
: During model training, data scientists run various experiments with different parameters on a given set of data. When scientists revisit their experiments after a period of time to reproduce the models, typically the source data has been modified by upstream pipelines. Lot of times, they are caught unaware by such upstream data changes and hence struggle to reproduce their experiments. Some scientists and organizations engineer best practices by creating multiple copies of the data, leading to increased storage costs. The same is true for analysts generating reports.

Rollbacks
: Data pipelines can sometimes write bad data for downstream consumers. This can happen because of issues ranging from infrastructure instabilities to messy data to bugs in the pipeline. For pipelines that do simple appends to directories or a table, rollbacks can easily be addressed by date-based partitioning. With updates and deletes, this can become very complicated, and data engineers typically have to engineer a complex pipeline to deal with such scenarios.

Working with Time Travel

Delta's time travel capabilities simplify building data pipelines for the preceding use cases which we will further explore in the following sections. As noted previously, as you write into a Delta table or directory, every operation is automatically versioned. You can access the different versions of the data two different ways:

1. Using a timestamp
2. Using a version number

In the following examples, we will use the generated TPC-DS dataset so we can work with the following example customer table (`customer_t1`). To review the historical data, run the `DESCRIBE HISTORY` command.

```
%sql
DESCRIBE HISTORY customer_t1;
```

The following results are an abridged version of the table initially focusing only on the `version`, `timestamp`, and `operation` columns.

```
-- Abridged Results
+-------+-------------------+---------+
|version|          timestamp|operation|
+-------+-------------------+---------+
|     19|2020-10-30 18:15:03|    WRITE|
|     18|2020-10-30 18:10:47|   DELETE|
|     17|2020-10-30 08:41:47|    WRITE|
|     16|2020-10-30 08:37:29|   DELETE|
|     15|2020-10-30 05:34:29|  RESTORE|
|     14|2020-10-30 02:50:07|    WRITE|
|     13|2020-10-30 02:37:17|   DELETE|
|     12|2020-10-30 02:31:36|    WRITE|
|     11|2020-10-28 23:33:34|   DELETE|
+-------+-------------------+---------+
```

The following sections describe the various techniques to work with your versioned data.

Using a timestamp

You can provide the timestamp or date string as an option to the DataFrame reader using the following syntax.

```
%sql
-- Query metastore-defined Delta table by timestamp
SELECT * FROM my_table TIMESTAMP AS OF "2019-01-01"
SELECT * FROM my_table TIMESTAMP AS OF date_sub(current_date(), 1)
SELECT * FROM my_table TIMESTAMP AS OF "2019-01-01 01:30:00.000"

-- Query Delta table by file path by timestamp
```

```
SELECT * FROM delta.`<path-to-delta>` TIMESTAMP AS OF "2019-01-01"
SELECT * FROM delta.`<path-to-delta>` TIMESTAMP AS OF date_sub(current_date(),
1)
SELECT * FROM delta.`<path-to-delta>` TIMESTAMP AS OF "2019-01-01 01:30:00.000"
```

```
%python
# Load into Spark DataFrame from Delta table by timestamp
(df = spark.read
  .format("delta")
  .option("timestampAsOf", "2019-01-01")
  .load("/path/to/my/table"))
```

```
%scala
// Load into Spark DataFrame from Delta table by timestamp
val df = spark.read
  .format("delta")
  .option("timestampAsOf", "2019-01-01")
  .load("/path/to/my/table")
```

Specific to the `customer_t1` table, the following queries will allow you to query by two different timestamps. The following SQL queries will use the metastore-defined Delta table syntax.

```
%sql
-- Row count as of timestamp t1
SELECT COUNT(1) FROM customer_t1 TIMESTAMP AS OF "2020-10-30T18:15:03.000+0000"

-- Row count as of timestamp t2
SELECT COUNT(1) FROM customer_t1 TIMESTAMP AS OF "2020-10-30T18:10:47.000+0000"
```

```
%python
# timestamps
t1 = "2020-10-30T18:15:03.000+0000"
t2 = "2020-10-30T18:10:47.000+0000"
DELTA_PATH="/demo/customer_t1"

>#  Row count as of timestamp t1
(spark.read.format("delta")
     .option("timestampAsOf", t1).load(DELTA_PATH).count())

# Row count as of timestamp t2
(spark.read.format("delta")\
     .option("timestampAsOf", t2).load(DELTA_PATH).count())
```

```
%scala
// timestamps
val t1 = "2020-10-30T18:15:03.000+0000"
val t2 = "2020-10-30T18:10:47.000+0000"
val DELTA_PATH="/demo/customer_t1"

// Row count as of timestamp t1
spark.read.format("delta")
     .option("timestampAsOf", t1).load(DELTA_PATH).count()
```

```
// Row count as of timestamp t2
spark.read.format("delta")
    .option("timestampAsOf", t2).load(DELTA_PATH).count()
```

The preceding queries have the following results below.

Timestamp	Row Count
2020-10-30T18:15:03.000+0000	65000000
2020-10-30T18:10:47.000+0000	58500000

If the reader code is in a library that you don't have access to, and if you are passing input parameters to the library to read data, you can still travel back in time for a table by passing the timestamp in `yyyyMMddHHmmssSSS` format to the path:

```
spark.re
ad.format("delta").load(<path-to-delta>@yyyyMMddHHmmssSSS)
```

For example, to query for the timestamp `2020-10-30T18:15:03.000+0000`, use the following Python and Scala syntax.

```
%python
# Delta table base path
BASE_PATH="/demo/customer_t1"

# Include timestamp in yyyyMMddHHmmssSSS format
DELTA_PATH=BASE_PATH + "@20201030181503000"

# Get row count of the Delta table by timestamp using @ parameter
spark.read.format("delta").load(DELTA_PATH).count()
```

```
%scala
// Delta table base path
val BASE_PATH="/demo/customer_t1"

// Include timestamp in yyyyMMddHHmmssSSS format
val DELTA_PATH=BASE_PATH + "@20201030181503000"

// Get row count of the Delta table by timestamp using @ parameter
spark.read.format("delta").load(DELTA_PATH).count()
```

Using a version number

In Delta, every write has a version number, and you can use the version number to travel back in time as well.

```
%sql
-- Query metastore-defined Delta table by version
SELECT COUNT(*) FROM my_table VERSION AS OF 5238
SELECT COUNT(*) FROM my_table@v5238
```

```
-- Query Delta table by file path by version
SELECT count(*) FROM delta.`/path/to/my/table@v5238`
```

```
%python
# Query Delta table by version using versionAsOf
(df = spark.read
  .format("delta")
  .option("versionAsOf", "5238")
  .load("/path/to/my/table"))

# Query Delta table by version using @ parameter
(df = spark.read
  .format("delta")
  .load("/path/to/my/table@v5238"))
```

```
%scala
// Query Delta table by version using versionAsOf
val df = spark.read
  .format("delta")
  .option("versionAsOf", "5238")
  .load("/path/to/my/table")

// Query Delta table by version using @ parameter
val df = spark.read
  .format("delta")
  .load("/path/to/my/table@v5238")
```

Specific to the `customer_t1` table, the following queries will allow you to query by two different versions. The following SQL queries will use the metastore-defined Delta table syntax.

```
%sql
-- Query metastore-defined Delta table by version 19
SELECT COUNT(1) FROM customer_t1 VERSION AS OF 19
SELECT COUNT(1) FROM customer_t1@v19

-- Query metastore-defined Delta table by version 18
SELECT COUNT(1) FROM customer_t1 VERSION AS OF 18
SELECT COUNT(1) FROM customer_t1@v18
```

```
%python
# Delta table base path
DELTA_PATH="/demo/customer_t1"

# Row count of Delta table by version 19
(spark.read.format("delta")
     .option("versionAsOf", 19).load(DELTA_PATH).count())

# Row count of Delta table by version 18
(spark.read.format("delta")
  .option("versionAsOf", 18).load(DELTA_PATH).count())

# Row count of Delta table by @ parameter for version 18
(spark.read
```

```
  .format("delta")
  .load(DELTA_PATH + "@v18").count())

%scala
# Delta table base path
val DELTA_PATH="/demo/customer_t1"

# Row count of Delta table by version 19
spark.read.format("delta")
     .option("versionAsOf", 19).load(DELTA_PATH).count()

# Row count of Delta table by version 18
spark.read.format("delta")
     .option("versionAsOf", 18).load(DELTA_PATH).count()

# Row count of Delta table by @ parameter for version 18
spark.read
  .format("delta")
  .load(DELTA_PATH + "@v18").count()
```

The preceding queries have the following results below.

Version	Row Count
19	65000000
18	58500000

Time travel use cases

In this section, we will focus on how to apply time travel for various use cases, and why it's important to them.

Debug
: To troubleshoot the ETL pipeline or data quality issues, or to fix the accidental broken data pipelines. This is the most common use case for time travel and we will be exploring this use case throughout the book.

Governance and Auditing
: Time travel, offers a verifiable data lineage that is useful for governance, audit and compliance purposes. As the definitive record of every change ever made to a table, you can trace the origin of an inadvertent change or a bug in a pipeline back to the exact action that caused it. If your GDPR pipeline job had a bug that accidentally deleted user information, you can fix the pipeline. Retention and vacuum allows you to act on Data subject requests well within the timeframe.

Rollbacks
: Time travel also makes it easy to do rollbacks in case of bad writes. Due to a previous error, it may be necessary to rollback to a previous version of the table before continuing forward with data processing.

Reproduce experiments & reports
: Time travel also plays an important role in machine learning and data science. Reproducibility of models and experiments is a key consideration for data scientists, because they often create 100s of models before they put one into production, and in that time-consuming process would like to go back to earlier models.

Pinned view of a continuously updating Delta table across multiple downstream jobs
: With AS OF queries, you can now pin the snapshot of a continuously updating Delta table for multiple downstream jobs.

Time series analytics
: Time travel simplifies time series analytics. You can use `timestamp` and `As Of` queries to extract meaningful statistics associated with the selected data points and shift in associated variables over time.

Next we will provide further details for each use case. Note, we will cover more advanced use cases which include time travel in chapter 14.

Use Case: Governance and Auditing

The use case of Governance described here is about protecting people's information in regards to GDPR (General Data Protection Regulation) and CCPA (California Consumer Privacy Act). The role of governance is to set the direction of protecting information in the form of policies, standards, and guidelines.

One of the most common scenarios surrounding governance to protect people's information is the need to perform a delete request as part of any GPDR or CCPA compliance. What that means for data engineers is that you need to locate the records from your Delta Lake and delete it within the specified time period. Users can run `DESCRIBE HISTORY` to see metadata around the changes that were made. An example of this scenario is how Starbucks implements governance, risk and control for their Delta lake.

Let's dive further in the simplified governance use case (see Figure 3-x)..

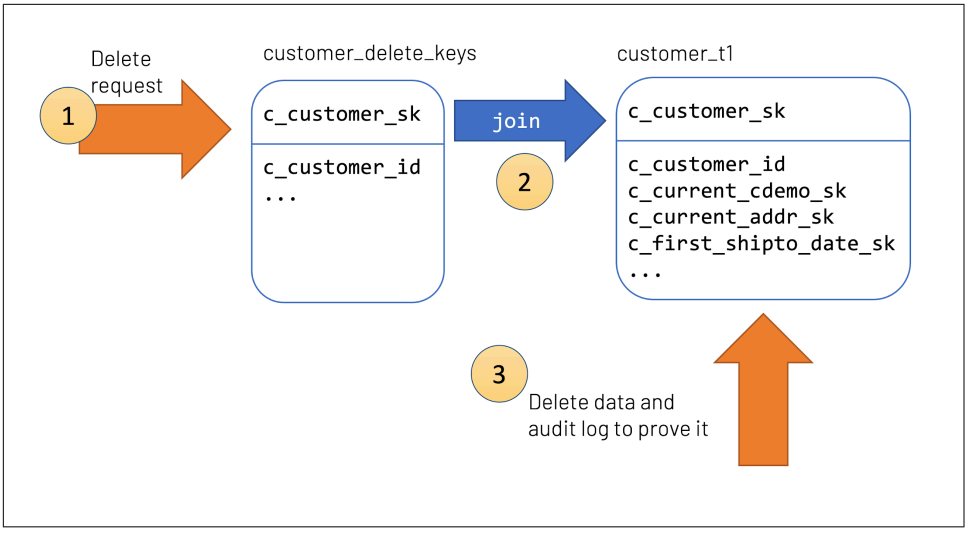


Figure 2-7. Simplified governance use case with time travel

1. A delete request is recorded in the system where the user has requested that their information be removed from the data lake. This information is often recorded in its own table or store for auditing and lineage purposes.
2. This table (`customer_delete_keys`) will need to join with the customer table (`customer_t1`) to identify which users in the table need to be removed.
3. You will then delete this data from the system to address this delete request.

With Delta Lake, you can achieve the above listed goals by running the following steps. You can also use this notebook for the whole code.

Identify users that need to be removed

As noted earlier, the `customer_delete_keys` table contains the set of customer keys that need to be removed. To review the table, run the following query.

```
%sql
SELECT * FROM customer_delete_keys

%python
# Query table using metastore defined table
sql("SELECT * FROM customer_delete_keys").show()

# Query table from file path
spark.read.format(“delta”).load("/customer_delta_keys").show()

%scala
// Query table using metastore defined table
sql("SELECT * FROM customer_delete_keys”).show()
```

```
// Query table from file path
spark.read.format(“delta”).load("/customer_delta_keys").show()
```

Note, going forward in this section, we will only use metastore-defined SQL queries. In Python and Scala, you can also use the syntax:

```
sql("<sql-statement-here>").show()
```

A partial result of the above queries can be seen in the output below.

```
+----------------+-------------+
|   c_customer_id|c_customer_sk|
+----------------+-------------+
|AAAAAAAAKHJBCKBA|     27400570|
|AAAAAAAAFIJBCKBA|     27400581|
|AAAAAAAAGIJBCKBA|     27400582|
|AAAAAAAAHIJBCKBA|     27400583|
|...             |          ...|
+----------------+-------------+
```

The following SQL query will allow you to quickly identify the records that need to be removed from `customer_t1`.

```
%sql
-- Identify the records
SELECT COUNT(*)
  FROM customer_t1
 WHERE c_customer_id IN (
  SELECT c_customer_id
    FROM customer_delete_keys
)
```

```
-- Result
+--------+
|count(1)|
+--------+
| 6500000|
+--------+
```

Delete the identified users

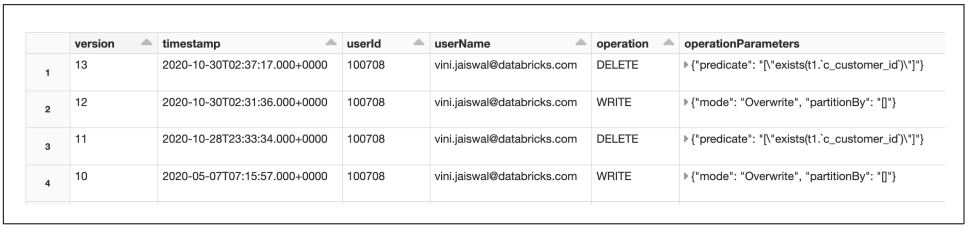
Once we have identified the matches, you can now go ahead and use the DELETE function to remove those records from the Delta table.

```
%sql
-- Delete identified records
DELETE
  FROM customer_t1
 WHERE c_customer_id IN (
  SELECT c_customer_id
```

```
    FROM customer_delete_keys
)
```

Fortunately this process is pretty straightforward as you can remove the `SELECT` clause and replace it with a `DELETE` clause. You can validate this by reviewing the transaction log via the `DESCRIBE HISTORY` command (See Figure 3-x).

```
%sql
DESCRIBE HISTORY customer_t1;
```



	version	timestamp	userId	userName	operation	operationParameters
1	13	2020-10-30T02:37:17.000+0000	100708	vini.jaiswal@databricks.com	DELETE	{"predicate": "[\"exists(t1.`c_customer_id`)\"]"}
2	12	2020-10-30T02:31:36.000+0000	100708	vini.jaiswal@databricks.com	WRITE	{"mode": "Overwrite", "partitionBy": "[]"}
3	11	2020-10-28T23:33:34.000+0000	100708	vini.jaiswal@databricks.com	DELETE	{"predicate": "[\"exists(t1.`c_customer_id`)\"]"}
4	10	2020-05-07T07:15:57.000+0000	100708	vini.jaiswal@databricks.com	WRITE	{"mode": "Overwrite", "partitionBy": "[]"}

Figure 2-8. Reviewing history after fulfilling GDPR delete request

Below is the transposed table of the DELETE transaction so we can inspect information.

Column Name	Column Value
version	13
operation	DELETE
operationParameters	[predicate -> ["exists(t1.`c_customer_id`)"]]
operationMetrics	[numRemovedFiles -> 23, numDeletedRows -> 6500000, numAddedFiles -> 100, numCopiedRows -> 42250002]

Let's dive further in the audit use case. Auditor's role is to review and inspect that delivery conforms to those guidelines. For auditing purposes, you can review the Delta transaction log which tracks all of the changes to your Delta table. In this case, for version 13 of the table:

- Through the `operation` column, a `DELETE` activity was recorded.
- The `operationParameters` denote how the delete occurred, i.e. the delete was done by the `c_customer_id` column.
- From an auditing perspective, the important callout is the 6,500,000 rows deleted (`numDeletedRows`) under `operationMetrics`. This allows us to determine the magnitude of impact on the records.

Delete is a tombstone. It is important to note that when a `DELETE` is executed on a Delta table, the most current version of the table no longer reads this data, as it is explicitly excluded. If you recall earlier in the chapter, the `Remove` column of the transaction log will contain the files that no longer should be included.

For some organizations, this meets their compliance requirements as the data is no longer accessible by default. But depending on your legal requirements, there are additional considerations:

- If the data must be removed immediately, run the `VACUUM` command to remove the data files
- Instead of deleting, `UPDATE` the data (e.g. update the name column with some other value such as a GUID) so you can keep the associated IDs without having any personally identifiable information.
- Place personally identifiable information into a separate table so that GDPR delete requests would result in only updating or deleting the information from this demographics table with little to no impact on your fact data.

It is important to note that this example is showcasing a simplified governance use case. You will need to work with your legal department to determine your actual requirements. The key takeaway for our discussion here is that Delta Lake time travel allows you to safely remove data from your Delta Lake, reverse it in case there are mistakes, and track the progress through the Delta Lake transaction log.

For more information on governance:

- Tech Talk | Addressing GDPR and CCPA Scenarios with Delta Lake and Apache Spark
- Data+AI Summit 2020 | Operationalizing Big Data Pipelines At Scale at Starbucks
- Webinar | Is Your Data Lake GDPR Ready? How to Avoid Drowning in Data Requests

Use Case: Rollbacks

Time travel also makes it easy to do rollbacks so you can reset your data to a previous snapshot. If a user accidentally inserted new data or your pipeline job inadvertently deleted user information, you can easily fix it by restoring the previous version of the table.

The following code sample takes our existing `customer_t1` table and restores it to version 17. Let's start by reviewing the history of our table.

```
%sql
DESCRIBE HISTORY customer_t1

--- abridged results
+-------+---------+-----------------------------------------+
|version|operation|operationParameters                      |
+-------+---------+-----------------------------------------+
|18     |DELETE   |[predicate -> ["exists(t1.`c_customer_id`)"]]|
|17     |WRITE    |[mode -> Overwrite, partitionBy -> []]   |
+-------+---------+-----------------------------------------+
```

Insert and overwrite. To rollback, what we really are doing is that once we identify which version of the table we want to travel back to, we will load that version using `as of`. Behind the scenes, delta inserts it as a new version and overwrites the latest snapshot. In this scenario, you can see from the abridged history results that version 18 of the table contains a `DELETE` operation that we want to rollback from. The following code allows you to perform the rollback.

```
%python
# Restore customer_t1 table to Version 17
>#  Load Version 17 data using df
df = spark.read.format("delta").option("versionAsOf", "17").load("/customer_t1")

# Overwrite Version 17 as the current version
df.write.format("delta").mode("overwrite").save("/customer_t1")

%scala
// Restore customer_t1 table to Version 17
// Load Version 17 data using df
val df = spark.read.format("delta").option("versionAsOf", "17").load("/
customer_t1")

// Overwrite Version 17 as the current version
df.write.format("delta").mode("overwrite").save("/customer_t1")
```

You can validate this by either querying the data or reviewing the table history. The following is how you would do this using SQL.

```
%sql
-- Get the row counts between two table versions and calculate the difference
 SELECT a.v19_cnt, b.v17_cnt, a.v19_cnt - b.v17_cnt AS difference
   FROM
 -- Get V19 row count
(SELECT COUNT(1) AS v19_cnt
   FROM customer_t1 VERSION AS OF 19) a
  CROSS JOIN
 -- Get V17 row count
(SELECT COUNT(1) AS v17_cnt
>   FROM customer_t1 VERSION AS OF 17) b

-- Result
+--------+--------+----------+
| v19_cnt| v17_cnt|difference|
```

```
+--------+--------+----------+
|65000000|65000000|         0|
+--------+--------+----------+
```

A comparable code can be written in Python and Scala versions as shown below .

```
%python
# Get V19 row count
v19_cnt = sql("SELECT COUNT(1) AS v19_cnt FROM customer_t1 VERSION AS OF
19").first()[0]

# Get V17 row count
v17_cnt = sql("SELECT COUNT(1) AS v17_cnt FROM customer_t1 VERSION AS OF
17").first()[0]

# Calculate the difference
print("Difference in row count between v19 and v17: %s" % (v19_cnt - v17_cnt))
```

```
%scala
// Get V19 row count
val v19_cnt = sql("SELECT COUNT(1) AS v19_cnt FROM customer_t1 VERSION AS OF
19").first()(0).toString.toInt

// Get V19 row count
val v17_cnt = sql("SELECT COUNT(1) AS v17_cnt FROM customer_t1 VERSION AS OF
17").first()(0).toString.toInt

// Calculate the difference
print("Difference in row count between v19 and v17: " + (v19_cnt - v17_cnt))
```

You will get the same result for both the Python and Scala examples.

```
-- Result
Difference in row count between v19 and v17: 0
```

You can also validate the restore by reviewing the table history; note that there is now version 19 that records the overwrite statement.

```
%sql
DESCRIBE HISTORY customer_t1
```

```
--- abridged results
 +-------+---------+-------------------------------------------+
|version|operation|operationParameters                        |
+-------+---------+-------------------------------------------+
|19     |WRITE    |[mode -> Overwrite, partitionBy -> []]     |
|18     |DELETE   |[predicate -> ["exists(t1.`c_customer_id`)"]]|
|17     |WRITE    |[mode -> Overwrite, partitionBy -> []]     |
+-------+---------+-------------------------------------------+
```

RESTORE command. Note, if you are using Databricks, this step will be a little easier when restoring with at least DBR 7.4 with the RESTORE command. You can perform the same steps as above with a single atomic command.

```
%sql
-- Restore table as of version 17
RESTORE customer_t1 VERSION AS OF 17
```

In addition to the previous queries comparing the count of two different versions, once you run the `RESTORE`, additional information gets stored within the transaction log which are shown under the `operationMetrics` column map.

```
%sql
DESCRIBE HISTORY customer_t1
```

```
-- Abridged results

+-------+---------+-------------------------------------------+
|version|operation|operationParameters                        |
+-------+---------+-------------------------------------------+
|20     |RESTORE  |[version -> 17, timestamp ->]              |
|19     |WRITE    |[mode -> Overwrite, partitionBy -> []]     |
|18     |DELETE   |[predicate -> ["exists(t1.`c_customer_id`)"]]|
|17     |WRITE    |[mode -> Overwrite, partitionBy -> []]     |
+-------+---------+-------------------------------------------+
```

We are highlighting the `record 20` from our output above for the `restore` operation of the delta table to explain the `operationMetrics` restore-based parameters.

```
{
"numRestoredFiles": "31",
"removedFilesSize": "3453647999",
"numRemovedFiles": "31",
"restoredFilesSize": "3453647999",
"numOfFilesAfterRestore": "31",
"tableSizeAfterRestore": "3453647999"
}
```

For more information about the RESTORE command, see Restore a Delta table.

Time travel considerations

We need to keep additional considerations in mind with respect to time travel.

Data retention:

- One very important thing to remember is Data retention. You can only travel back so far depending on what time limits we set and how much data is retained.
- To time travel to a previous version, you must retain both the log and the data files for that version.

Vacuum:

- The ability to time travel back to a version older than the retention period is lost after running vacuum.
- By default, `vacuum()` retains all the data needed for the last 7 days.
- When you run the `vacuum` with specified retention of 0 hours, it will throw an exception.
  - Delta protects you by giving this warning to save you from permanent changes to your data or if a downstream reader or application is referencing your table.
  - If you are certain that there are no operations being performed on this table, such as insert/upsert/delete/optimize, then you may turn off this check by setting:
    `spark.databricks.delta.retentionDurationCheck.enabled = false`
- `VACUUM` doesn't clean up log files; log files are automatically cleaned up after checkpoints are written.
- Another note to satisfy the delete requirement is that you also need to delete it from your blob storage. So as a best practice, whenever the ETL pipeline is set you should apply the retention policy from the start itself so you don't have to worry later.

Summary

In this chapter, we focussed on a powerful benefit of the MVCC (Multi-Version Concurrency Control) implementation of Delta, that enables users to have the benefit of data versioning or *time travel*. In addition to showing common challenges with changing data, we explained the Delta time machine and its properties, how time travel works and what operations you can perform while traveling back in time.

As alluded to earlier in the chapter, time travel allows developers to debug what happened with their Delta table, improving developer productivity tremendously. One of the common reasons developers need to debug their table is due to data modification such as delete, update, and merge. This brings us to our next chapter, Chapter 4: Data Modifications in Delta tables.

CHAPTER 3

Continuous Applications with Delta Lake

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

In the previous chapters, we discussed the Delta transaction log (Chapter 2), time travel (Chapter 3), schema enforcement and evolution (Chapter 4) and data modifications (Chapter 5). While this was never explicitly stated, one of the primary reasons for creating Delta Lake was to provide *atomic transactionality* for continuous applications. At its most fundamental level, the concept of continuous applications is the ability to develop streaming applications without the need *to reason* for streaming applications. In this chapter, we will initially provide the background of streaming applications, the importance of placing structure around streaming, and how this progression of structured streaming eventually led to the development of Delta Lake. In the process of creating structured streaming, for developers to write application logic not defined by if the code was streaming or batch processing, we would need to decouple business logic from latency. It is then ironic that we needed to pull a concept from tightly coupled relational database systems to perform this task - to bring back ACID transactions - hence the development of Delta Lake.

Note, this chapter will review high-level concepts of Spark Streaming and Spark Structured Streaming including example code. But, it will not provide an in-depth review of all of these concepts; for more information, please refer to Learning Spark 2nd Edition, Chapter 8: Structured Streaming and Continuous Applications. Instead, this chapter will provide a retrospective from streaming to continuous applications and how these concepts helped define what is now Delta Lake.

Make All Your Streams Come True

Let's go back in time to the early 2010s where the context of streaming and complex event processing was with the Apache Storm project. Note, other open-source and proprietary streaming projects did exist at this time but due to the popularity of Twitter and the open-sourcing of the Storm project, Apache Storm *stormed* onto the streaming scene (sorry, not sorry for the pun). So at least from an open-source big data perspective, the Storm project was the baseline for development of streaming.

The benefits of streaming were that it was tuned for lower latency with reasonable throughput for lower or medium size data. But as this was the nascent age of Big Data, what was desired was the ability to have higher throughput and lower latency for larger amounts of data. Because a streaming system like Storm was continuously processing data, you could achieve performing an event processing against large amounts of data since the event processing itself was against a small amount of data. A simple example of this would be an `onclick` event where when a user clicked on a link, this would trigger some other process downstream (e.g. As an ad-business, you provide an advertisement specific to the link(s) the user clicked).

Achieving event processing with state proved to be a much more complicated task. Calculating aggregates against continuous streams of data required memory and file management, moreover doing so in a reliable manner became exceedingly difficult. A simple example of this would be for you to calculate across hourly time windows how many users had clicked on a link. Because the aggregations could become exceedingly complex and expensive, typically this was achieved by building separate pipelines to process the event processing data while another pipeline to process the state or aggregate data. This latter design led to the creation of the *lambda architecture* which allowed us to build two different systems as defined by the batch/serving and speed layers. This enabled developers to choose and customize their frameworks to optimize for each respective layer. For more details on the lambda architecture, refer to Chapter 7: Medallion Architecture.

The problem with this approach was that it required building and maintaining two different pipelines. These pipelines were often created by two different teams with different programming paradigms; for example, the speed layer was addressed by using storm-clojure while the batch/server layer was addressed by using Hadoop and

Hive jobs. These differences prevented utilizing the same team to address these problems and often required a third team to reconcile these two layers.

To simplify this, Spark Streaming was built to unify these two layers. Let's deep dive into this in the next section.

Spark Streaming Was Built to Unify Batch and Streaming

Many developers in the Big Data community were already familiar with and using Apache Spark™ for batch processing large amounts of data. Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

The benefits of using the same framework to address both speed and batch processing were immense including:

- A **high-level API** that simplified and extended event processing functionality including joins and windows often with significantly less code.
- Spark streaming is **fault-tolerant** with exactly-once semantics even for stateful ops (more on exactly-once semantics later in the chapter)
- Because it is Spark, there were the benefits of **integration** with other features of Spark: Machine Learning with MLlib, SQL and DataFrames, and graph processing with GraphX to name a few.

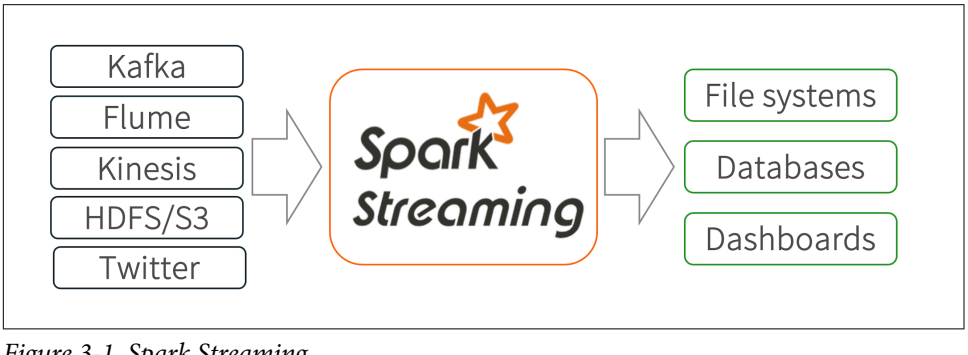


Figure 3-1. Spark Streaming

As noted previously, for more information on Spark Streaming, refer to Learning Spark 2nd Edition, Chapter 8: Structured Streaming and Continuous Applications or Spark Streaming - Spark 3.0.1 Documentation (apache.org).

What are the primary use cases for streaming?

For many, Spark Streaming opened the door for a new paradigm in data processing as many data practitioners had focused on batch processing. As our fellow author Tathagata Das noted[1]:

Most companies are collecting more data than ever and wanting to get value from that data in real-time. Whether it was sensors, IoT devices, social networks, or online transactions, they all generated a massive amount of data that needs to be monitored constantly and acted upon quickly. As a result, the need for large-scale, real-time stream processing has become more evident than ever before.

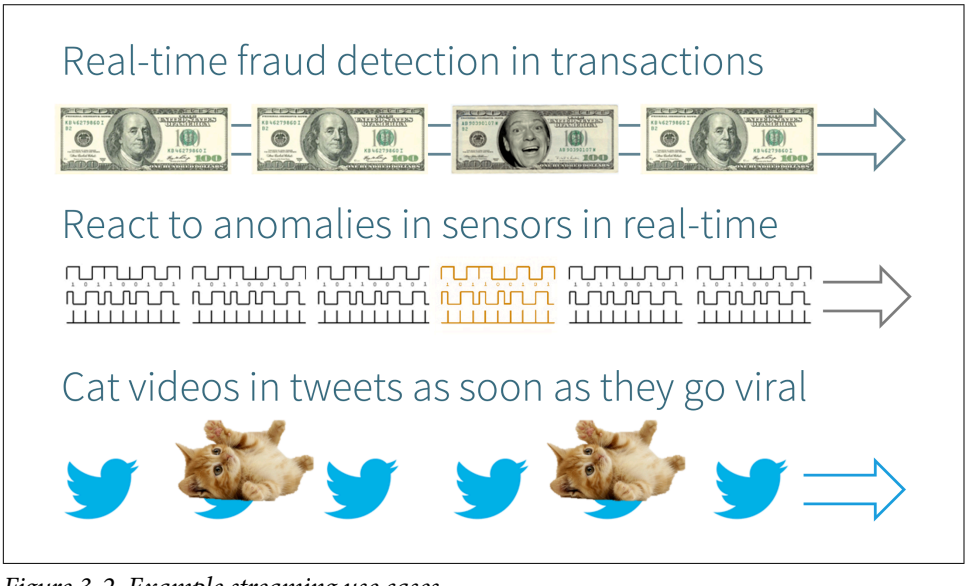


Figure 3-2. Example streaming use cases

That same streaming data is likely collected and used in batch jobs when generating daily reports and updating models. This means that a modern stream processing pipeline needs to be built, taking into account not just the real-time aspect, but also the associated pre-processing and post-processing aspects (e.g. model building).

Before Spark Streaming, building complex pipelines that encompass streaming, batch, or even machine learning capabilities with open source software meant dealing with multiple frameworks, each built for a niche purpose, such as Storm for real-time actions, Hadoop MapReduce for batch processing, etc. Besides the pain of developing disparate programming models, there was a huge cost of managing multiple frame-

1 See Spark Streaming: What Is It and Who's Using It?

works in production. Spark and Spark Streaming, with its unified programming model and processing engine, makes all of this very simple.

Thus, if you were to view the data world from the lens of latency, then we could view this purely from the standpoint of traditional streaming systems (e.g. alerting) and traditional batch systems (e.g. ETL).

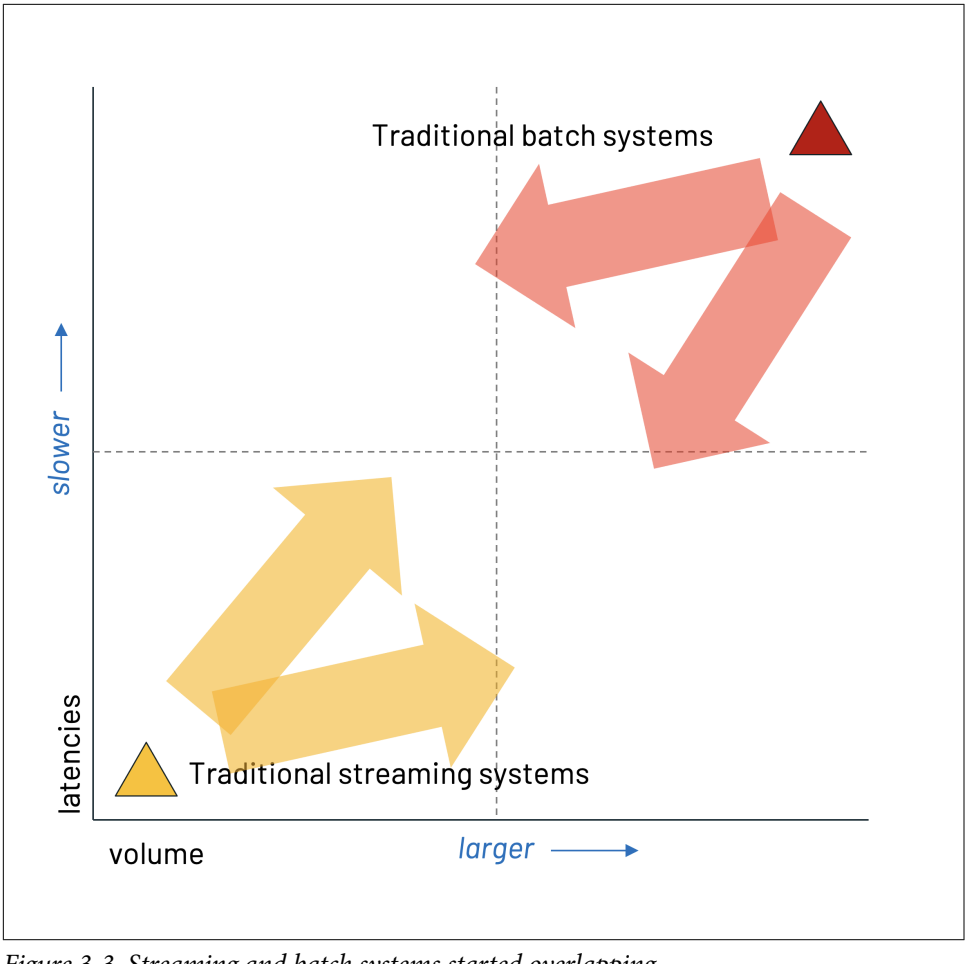


Figure 3-3. Streaming and batch systems started overlapping

But these primary use cases started overlapping as traditional batch systems required lower latencies and traditional streaming systems were required to deal with larger volumes (e.g. aggregations). This is how Spark Streaming unified batch and stream processing because it allows you to solve the continuum of real-time analytics.

Use case highlights

As they noted in their blog Can Spark Streaming Survive Chaos Monkey, Netflix receives billions of events per day from various sources, and they have used Kafka and Spark Streaming to build a real-time engine that provides movie recommendations to its users.

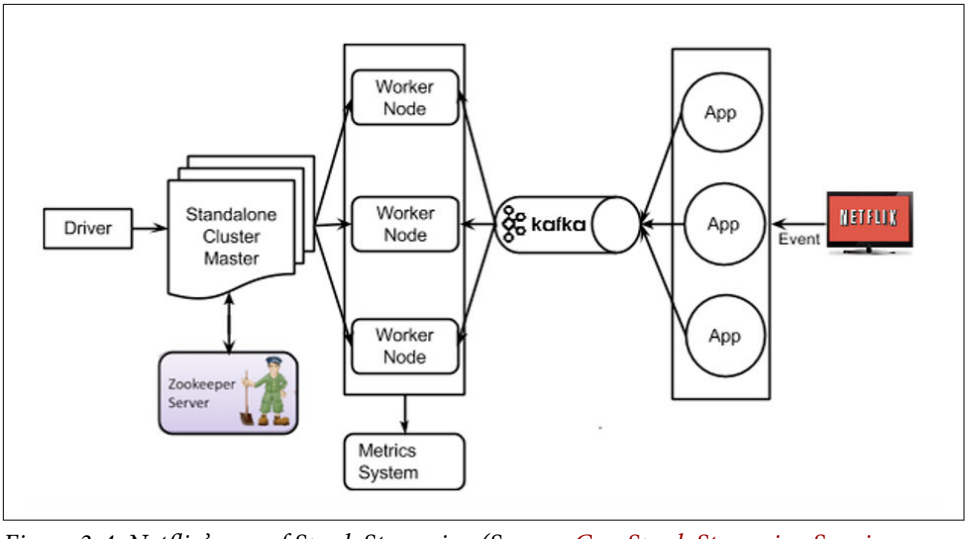


Figure 3-4. Netflix's use of Spark Streaming (Source: Can Spark Streaming Survive Chaos Monkey)

As noted in the Spark+AI Summit session Spark and Spark Streaming at Netflix, back in 2015, Netflix processed over 450 billion events/day with a peak of 8 million events/second (approx 17GB). A great quote from Netflix concerning Spark Streaming can be seen below.

> Overall, we are happy with the resiliency of spark standalone for our use cases and excited to take it to the next level where we are working towards building a unified Lambda Architecture that involves a combination of batch and real-time streaming processing.

As the preceding quote calls out nicely, Spark Streaming allowed Netflix and other organizations to build a unified architecture that combined traditional streaming and batch processing.

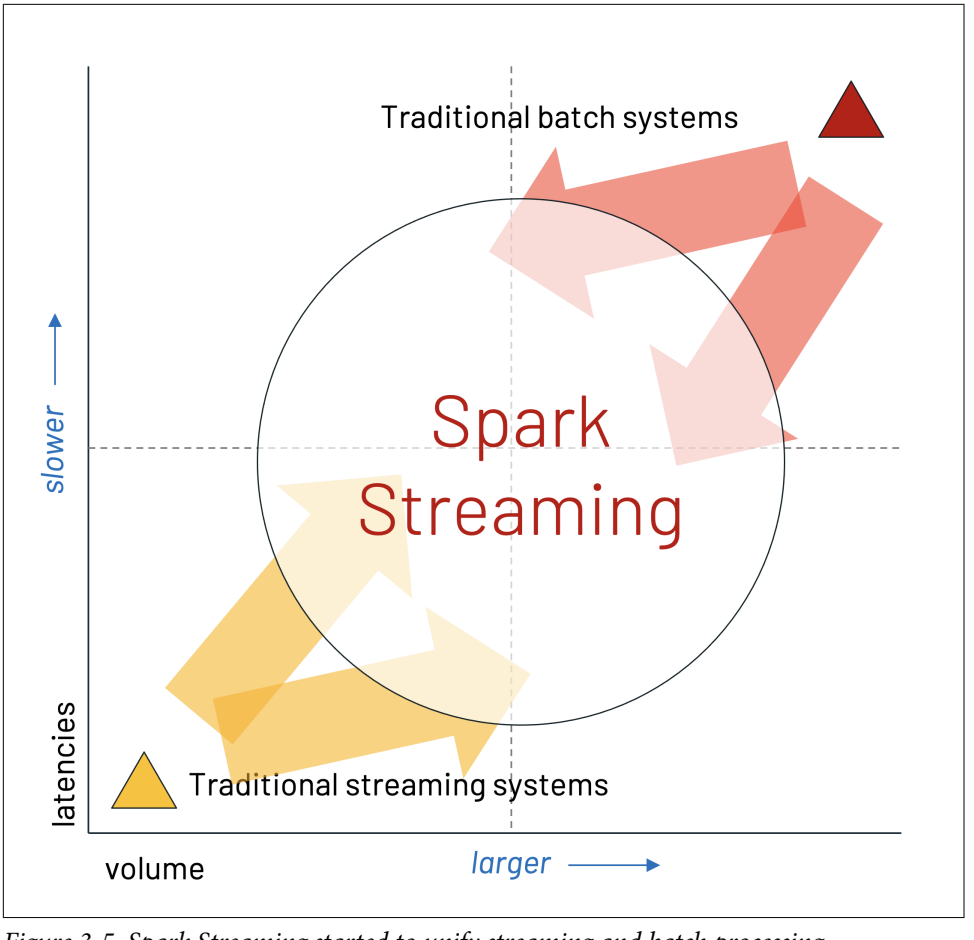


Figure 3-5. Spark Streaming started to unify streaming and batch processing

Spark Streaming lacked structure

While Spark Streaming started unifying the concepts of streaming and batch processing, it could not take advantage of the architectural changes of the Apache Spark™ project. For starters, while streaming and batch were using the same framework, they were using different APIs - `DStreams` and `RDDs` respectively. At the same time, `DStreams` were not able to take advantage of the performance improvements of providing structure to Apache Spark including but not limited to the new structured file formats (e.g. Parquet, ORC, etc.), Spark SQL, Spark Catalyst optimizer, Project Tungsten. For more information on this, please refer to:

- Learning Spark 2nd Edition
- Structuring Spark: DataFrames, Datasets, and Streaming by Michael Armbrust.

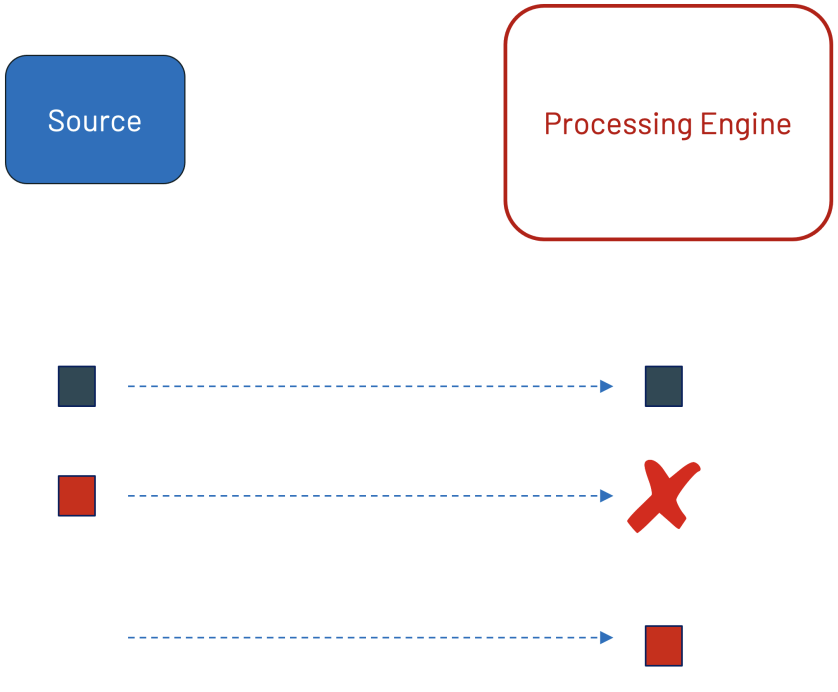
- Deep Dive into Spark SQL's Catalyst Optimizer
- Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop.

Simply put, Spark Streaming itself was not designed for structured data. Before we discuss how to provide structure to streaming - i.e. Structured Streaming - let's also discuss an important concept: exactly-once semantics.

Exactly-Once Semantics

The desired result of data processing - irrespective of latency - is exactly-once semantics. In other words, for every record end-to-end from source to target, there is neither duplicates nor any missing or incomplete data.

Let's take an example to explain this in detail.

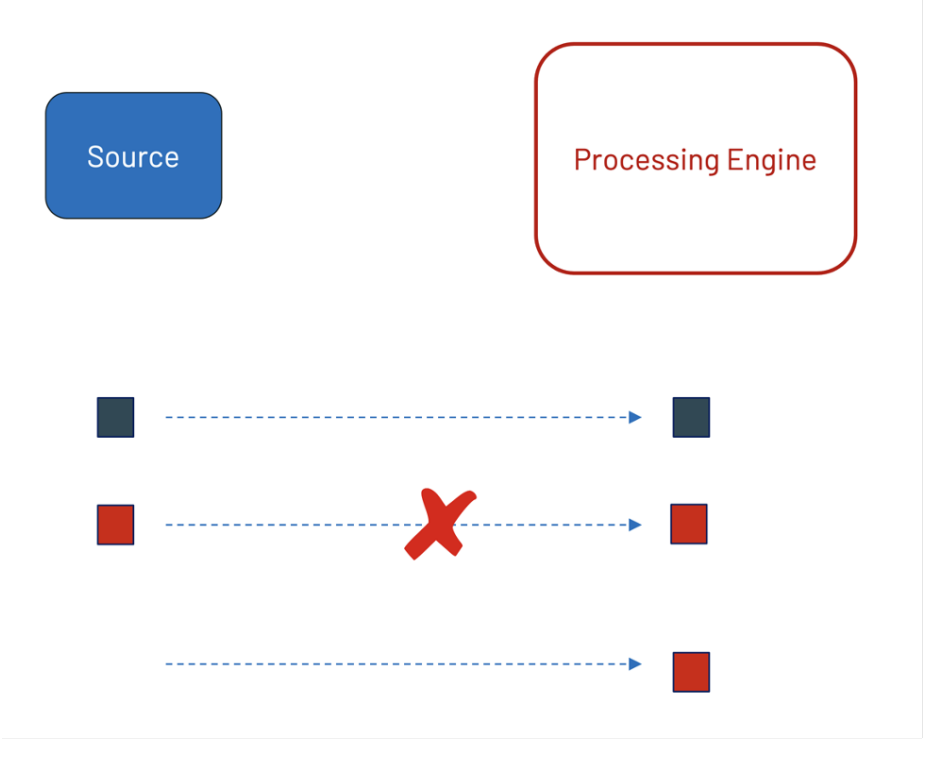


In the figure above, there are multiple streams from the source to the Processing Engine.

1. The first stream shows a complete successful event
2. The second one in red, shows a failure at the Processing engine level. If the failures occur, the end result is that the processing engine should account for each record once.
3. When finally writing the data out for storage, this must be done in an idempotent manner so that there is only one copy of this record.

While this may be an obvious requirement in building and maintaining batch data pipelines, but for streaming pipelines it was not. Let's summarize :

- Storm provided at-least once: no missing data, but duplicates could possibly occur.
- The Spark Streaming solved (2) but the writes were not idempotent (3) from the above listed scenarios
- The Structured streaming solves both. It started getting some of the ACID semantics back for inserts / appends to a Parquet table

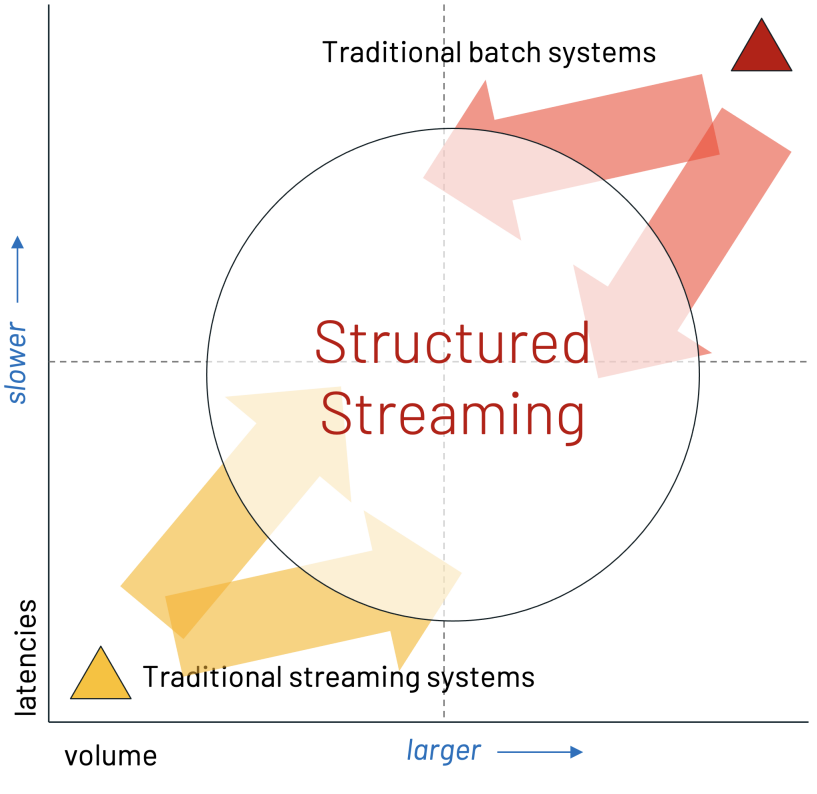


Putting Some Structure Around Streaming

In Apache Spark 2.0, adding structure to Spark, through use of high-level DataFrames and Datasets APIs, accommodates a novel approach to look at real-time streaming. That is, look at streaming not as streaming but as either a static table of data (where you know all the data) or a continuous table of data (where new data is continuously arriving).

As such you can build end-to-end continuous applications, in which you can issue the same queries to batch as to real-time data, perform ETL, generate reports, update or track specific data in the stream. This combined batch & real-time query-capabilities to a structured stream is a unique offering—not many streaming engines offer it yet. This can reduce latency and allow for incremental processing.

At a conceptual level, Structured Streaming treats live data as a table that is being continuously appended. The best thing about Structured Streaming is that it allows you to rapidly get value out of streaming systems with virtually no code changes.



Structured streaming allowed us to solve almost all the problems with ACID transactions. However building the pipelines with structured streaming only solves part of the problem. It solved the lack of ACID guarantees for inserts but data pipelines require cleaning up data, GDPR deletes, data reprocessing tasks are still a challenge.

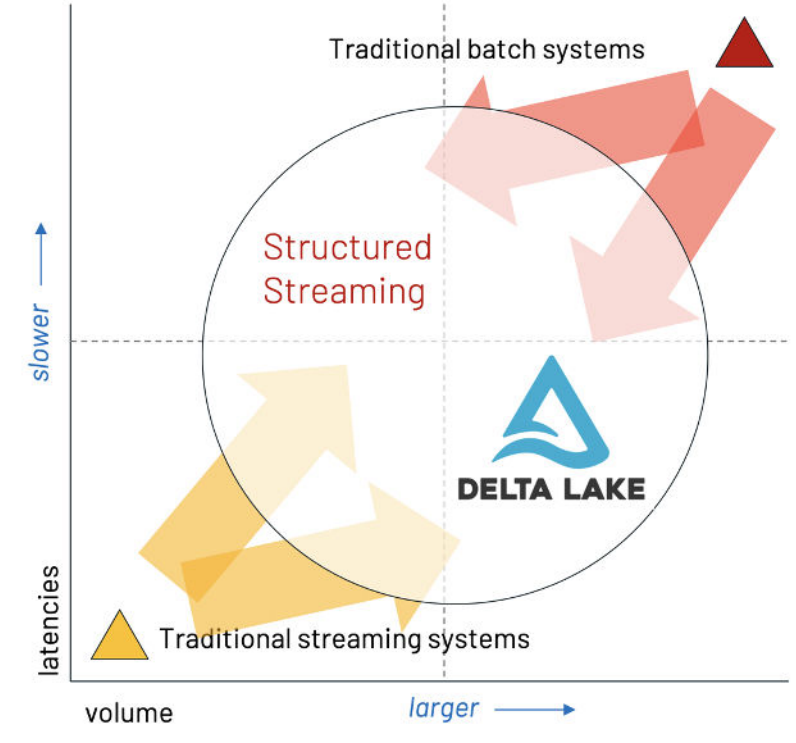
When reprocessing files in a directory, it is very easy to

- delete files that you need,
- Put them in the wrong folder, so the table cannot reconcile
- Partially written/corrupt files.

Here are more documents to get started with Structured streaming.

- Structured Streaming documentation
- Blog: Structured Streaming: A New High-Level API for Streaming
- Docs: Structured Streaming: Introductory Notebooks
- Docs: Structured Streaming in Production

This led to the idea of Delta Lake to ensure that all the changes are done in an atomic manner. More about this property was covered in detail in the Chapter 2, if you may recall.



A table in Delta Lake is a batch table as well as a streaming source and sink. Delta Lake uses Apache Parquet files to store your data, and it is fully compatible with Apache Spark APIs.

Streaming with Delta

Delta Lake is tightly integrated with Spark Structured Streaming through `readStream` and `writeStream`. Some of the challenges that Delta solves in the context of streaming are:

- Coalescing small files produced by low latency ingest
- Maintaining "exactly-once" processing with more than one stream (or concurrent batch jobs)
- Efficiently discovering which files are new when using files as the source for a stream

As described in one of our blogs with an advertising firm who serves data-driven online video ads for its clients, they leverage Structured Streaming, Delta Lake, and Databricks for the brand safety. The diagram below shows how Eyeview implemented the concept of brand safety and what challenges the company faced in doing so.

Key highlights of their use case was that they were getting numerous URLs every second and it can create following challenges while batch processing a large number of small files.

- Challenge #1: Underutilized Resources
- Challenge #2: Manual Checkpointing
- Challenge #3: Parquet Table Issues
- Challenge #4: No Concurrent Reads on Parquet Tables

To solve the above challenges, Eyeview introduced two new technologies: Spark Streaming and Delta Lake. The source of most of a large number of small files can be converted from batch processing to streaming processing.

1. Spark streaming helped in solving the first two challenges. Instead of a cluster of bidders writing files which contain the URLs to S3, they started sending URLs directly to a kinesis stream.
2. By connecting Spark Streaming with Kinesis streams they no longer need to do manual checkpointing. Since Spark Streaming is inherently fault-tolerant they didn't have to worry about failures and reprocessing of files.

The code snippet below reads the data from the Kinesis stream.

```
import org.apache.spark.sql.types._
val jsonSchema = new StructType()
 .add("entry", StringType)
 .add("ts", LongType)
val kinesisDF = spark.readStream
 .format("kinesis")
 .option("streamName", "kinesis Stream Name")
 .option("initialPosition", "LATEST")
 .option("region", "aws-region")
 .load()
val queryDf = kinesisDF
 .map(row => Charset.forName("UTF-8").newDecoder().decode(ByteBuffer.wrap(new
Base64().decode(row.get(1)).asInstanceOf[Array[Byte]])).toString)
 .selectExpr("cast (value as STRING) jsonData")
 .select(from_json(col("jsonData"), jsonSchema).as("bc"))
 .withColumn("entry",lit($"bc.entry"))
 .withColumn("_tmp", split($"entry", "\\,"))
 .select(
 $"_tmp".getItem(0).as("device_type"),
 $"_tmp".getItem(1).as("url"),
```

```
 $"_tmp".getItem(2).as("os"),
 $"bc.ts".as("ts")
).drop("_tmp")
```

1. The other two challenges with the parquet table are solved by introducing a new table format, Delta Lake. Delta Lake supports ACID transactions, which basically means they can concurrently and reliably read/write this table.
2. Delta Lake tables are also very efficient with continuous appends to the tables. A table in Delta Lake is both a batch table, as well as a streaming source and sink.

The below code shows persisting the data into Delta Lake. This also helped us in removing the millisecond partitions. See the below code for reference. (Partitions are up to only hour level.)

```
val sparkStreaming = queryDf.as[(String,String,String,Long)].mapPartitions{parti
tion=>
 val http_client = new HttpClient
 http_client.start
 val partitionsResult = partition.map{record=>
 try{
  val api_url = ApiCallingUtility.createAPIUrl(record._2,record._1,record._3)
  val result = ApiCallingUtility.apiCall(http_client.newRequest(api_url).time
out(500, TimeUnit.MILLISECONDS).send(),record._2,record._1)
  aerospikeWrite(api_url,result)
  result
 }
 catch{
  case e:Throwable=>{
  println(e)
  }
 }
 }
 partitionsResult
}
```

Once they moved the architecture from batch processing to a streaming solution, they were able to reduce the cluster size of the Spark jobs, thus significantly reducing the cost of the solution. More impressively, they only required one job to take care of all the brand safety providers, which further reduced costs.

Lets cover some more concepts of Delta streaming. We learned that a Delta table can be both - a stream source as well as sink.

Delta as a Stream Source

When you load a Delta table as a stream source and use it in a streaming query, the query processes all of the data present in the table as well as any new data that arrives after the stream is started.

You can load both paths and tables as a stream.

```
%Scala
spark.readStream.format("delta")
  .load("/mnt/delta/events")
import io.delta.implicits._
spark.readStream.delta("/mnt/delta/events")
```

or

```
%Scala
import io.delta.implicits._
spark.readStream.format("delta")
  .table("events")
```

Some helpful configuration tips:

- You can control the maximum size of any micro-batch that Delta Lake gives to streaming by setting the maxFilesPerTrigger option. This specifies the maximum number of new files to be considered in every trigger. The default is 1000.
- Rate-limit how much data gets processed in each micro-batch by setting the maxBytesPerTrigger option. This sets a "soft max", meaning that a batch processes approximately this amount of data and may process more than the limit. If you use Trigger.Once for your streaming, this option is ignored. If you use this option in conjunction with maxFilesPerTrigger, the micro-batch processes data until either the maxFilesPerTrigger or maxBytesPerTrigger limit is reached.

In cases when the source table transactions are cleaned up due to the logRetentionDuration configuration and the stream lags in processing, Delta Lake processes the data corresponding to the latest available transaction history of the source table but does not fail the stream. This can result in data being dropped.

Ignore Updates and Deletes

As discussed in an earlier section. Structured streaming can handle ingest but does not handle input that is not an append and throws an exception if any modifications

occur on the table being used as a source. There are two main strategies for dealing with changes that cannot be automatically propagated downstream:

- You can delete the output and checkpoint and restart the stream from the beginning.
- You can set either of these two options:
  - ignoreDeletes: ignore transactions that delete data at partition boundaries.
  - ignoreChanges: re-process updates if files had to be rewritten in the source table due to a data changing operation such as UPDATE, MERGE INTO, DELETE (within partitions), or OVERWRITE. Unchanged rows may still be emitted, therefore your downstream consumers should be able to handle duplicates. Deletes are not propagated downstream. ignoreChanges subsumes ignoreDeletes. Therefore if you use ignoreChanges, your stream will not be disrupted by either deletions or updates to the source table.

To explain this through an example, lets assume that you have `user_events table` with date, user_email, and action as the columns that is partitioned by `date`. As a part of GDPR you need to delete data from the user_events table.

When you delete at partition boundaries, the files are already segmented by value so the delete just drops those files from the metadata. Thus, if you just want to delete data from some partitions, you can use:

```
%scala
spark.readStream.format("delta")
  .option("ignoreDeletes", "true")
  .load("/mnt/delta/user_events")
```

However, if you have to delete data based on user_email, then you will need to use:

```
%scala
spark.readStream.format("delta")
  .option("ignoreChanges", "true")
  .load("/mnt/delta/user_events")
```

If you update a user_email with the UPDATE statement, the file containing the user_email in question is rewritten. When you use ignoreChanges, the new record is propagated downstream with all other unchanged records that were in the same file. Your logic should be able to handle these incoming duplicate records.

Specify initial position

This feature is only available on Databricks Runtime 7.3 LTS and above.

Initial position is where you want the stream to start with as a default position. Defining this position, allows you to specify a streaming source without processing the entire table. Since Delta saves each record in the form of versions along with the timestamps, you can specify initial position by using either of these two values.

- `startingVersion`: `startingVersion` allows you to specify the Delta version of the table you want to start from. All table changes starting from this version (inclusive) will be read by the streaming source.
- `startingTimestamp`: This allows you to specify the timestamp to start from. All table changes committed at or after the timestamp (inclusive) will be read by the streaming source.

You can retrieve both the above information using DESCRIBE command:

```
DESCRIBE HISTORY customer_t1;
```

The following results are an abridged version of the table, focusing only on the version, timestamp, and operation columns.

```
-- Abridged Results
+-------+-------------------+---------+
|version|timestamp|operation|
+-------+-------------------+---------+
|     19|2020-10-30 18:15:03|    WRITE|
|     18|2020-10-30 18:10:47|   DELETE|
|     17|2020-10-30 08:41:47|    WRITE|
```

The operations take effect only when starting a new streaming query. If a streaming query has started and the progress has been recorded in its checkpoint, these options are ignored.

Example

For example, suppose you have a table user_events. If you want to read changes since version 5, use:

```
%Scala
spark.readStream.format("delta")
  .option("startingVersion", "5")
  .load("/mnt/delta/user_events")
```

If you want to read changes since 2018-10-18, use:

```
%Scala
spark.readStream.format("delta")
  .option("startingTimestamp", "2018-10-18")
  .load("/mnt/delta/user_events")
```

Delta Table as a Sink

You can also write data into a Delta table using Structured Streaming. The transaction log enables Delta Lake to guarantee exactly-once processing, even when there are other streams or batch queries running concurrently against the table.

Append mode

By default, streams run in append mode, which adds new records to the table.

You can use the path method:

```
%Scala
events.writeStream
  .format("delta")
  .outputMode("append")
  .option("checkpointLocation", "/delta/events/_checkpoints/etl-from-json")
  .start("/delta/events")
```

or the table method:

```
%Scala
events.writeStream
  .format("delta")
  .outputMode("append")
  .option("checkpointLocation", "/delta/events/_checkpoints/etl-from-json")
  .table("events")
```

Complete mode

You can also use Structured Streaming to replace the entire table with every batch. One example use case is to compute a summary using aggregation:

```
%Scala
spark.readStream
  .format("delta")
  .load("/mnt/delta/events")
  .groupBy("customerId")
  .count()
  .writeStream
  .format("delta")
  .outputMode("complete")
  .option("checkpointLocation", "/mnt/delta/eventsByCustomer/_checkpoints/streaming-
agg")
  .start("/mnt/delta/eventsByCustomer")
```

The preceding example continuously updates a table that contains the aggregate number of events by customer.

For applications with more lenient latency requirements, you can save computing resources with one-time triggers. Use these to update summary aggregation tables on a given schedule, processing only new data that has arrived since the last update.

As a summary, we covered the evolution of streaming from Apache Storm to spark streaming to structured streaming and looked at the advantages of using Delta tables for the unified pipeline for batch and streaming source and sink. In the next chapter Medallion Architecture, we will cover the design of Delta Lake architecture and some of the evolving features like Delta Live tables.

Appendix

For more examples of streaming with Delta, please refer to the following notebooks:

- Using SQL Analytics to Query Your Data Lake With Delta Lake - Databricks
- Using SQL to Query Your Data Lake With Delta Lake - Databricks

About the Authors

Denny Lee is a Staff Developer Advocate at Databricks. He is a hands-on distributed systems and data sciences engineer with extensive experience developing internet-scale infrastructure, data platforms, and predictive analytics systems for both on-premise and cloud environments. He also has a Masters of Biomedical Informatics from Oregon Health and Sciences University and has architected and implemented powerful data solutions for enterprise Healthcare customers. His current technical focuses include Distributed Systems, Apache Spark, Deep Learning, Machine Learning, and Genomics.

Tathagata Das (TD) is a Staff Software Engineer at Databricks, an Apache Spark committer and a member of the Apache Spark Project Management Committee (PMC). He is one of the original developers of Apache Spark, the lead developer of Spark Streaming (DStreams) and is currently one of the core developers of Structured Streaming and Delta Lake. Tathagata holds a MS in computer science from UC Berkeley.

Vini Jaiswal is a Senior Developer Advocate at Databricks, where she helps data practitioners to be successful building on Databricks and open source technologies like Apache Spark, Delta and MLflow. She has extensive experience working with the Unicorns, Digital Natives and some of the Fortune 500 companies helping with successful implementation of Data and AI use cases in production at scale for on-premise and cloud deployments. Vini also worked as the Data Science Engineering Lead under Citi's Enterprise Operations & Technology group and interned as a Data Analyst at Southwest Airlines. She holds an MS in Information Technology and Management from University of Texas at Dallas.