Chapter 9. Building Reliable Data Lakes with Apache Spark

In the previous chapters, you learned how to easily and effectively use Apache Spark to build scalable and performant data processing pipelines. However, in practice, expressing the processing logic only solves half of the end-to-end problem of building a pipeline. For a data engineer, data scientist, or data analyst, the ultimate goal of building pipelines is to query the processed data and get insights from it. The choice of storage solution determines the end-to-end (i.e., from raw data to insights) robustness and performance of the data pipeline.

In this chapter, we will first discuss the key features of a storage solution that you need to look out for. Then we will discuss two broad classes of storage solutions, databases and data lakes, and how to use Apache Spark with them. Finally, we will introduce the next wave of storage solution, called lakehouses, and explore some of the new open source processing engines in this space.

The Importance of an Optimal Storage Solution

Here are some of the properties that are desired in a storage solution:

Scalability and performance

The storage solution should be able to scale to the volume of data and provide the read/write throughput and latency that the workload requires.

Transaction support

Complex workloads are often reading and writing data concurrently, so support for ACID transactions is essential to ensure the quality of the end results.

Support for diverse data formats

The storage solution should be able to store unstructured data (e.g., text files like raw logs), semi-structured data (e.g., JSON data), and structured data (e.g., tabular data).

Support for diverse workloads

The storage solution should be able to support a diverse range of business workloads, including:

- SQL workloads like traditional BI analytics
- Batch workloads like traditional ETL jobs processing raw unstructured data
- Streaming workloads like real-time monitoring and alerting
- ML and AI workloads like recommendations and churn predictions

Openness

Supporting a wide range of workloads often requires the data to be stored in open data formats. Standard APIs allow the data to be accessed from a variety of tools and engines. This allows the business to use the most optimal tools for each type of workload and make the best business decisions.

Over time, different kinds of storage solutions have been proposed, each with its unique advantages and disadvantages with respect to these properties. In this chapter, we will explore how the available storage solutions evolved from *databases* to *data lakes*, and how to use Apache Spark with each of them. We'll then turn our attention to the next generation of storage solutions, often called data *lakehouses*, that can provide the best of both worlds: the scalability and flexibility of data lakes with the transactional guarantees of databases.

Databases

For many decades, databases have been the most reliable solution for building data warehouses to store business-critical data. In this section, we will explore the architecture of databases and their workloads, and

how to use Apache Spark for analytics workloads on databases. We will end this section with a discussion of the limitations of databases in supporting modern non-SQL workloads.

A Brief Introduction to Databases

Databases are designed to store structured data as tables, which can be read using SQL queries. The data must adhere to a strict schema, which allows a database management system to heavily co-optimize the data storage and processing. That is, they tightly couple their internal layout of the data and indexes in on-disk files with their highly optimized query processing engines, thus providing very fast computations on the stored data along with strong transactional ACID guarantees on all read/write operations.

SQL workloads on databases can be broadly classified into two categories, as follows:

Online transaction processing (OLTP) workloads

Like bank account transactions, OLTP workloads are typically high-concurrency, low-latency, simple queries that read or update a few records at a time.

Online analytical processing (OLAP)

OLAP workloads, like periodic reporting, are typically complex queries (involving aggregates and joins) that require high-throughput scans over many records.

It is important to note that Apache Spark is a query engine that is primarily designed for OLAP workloads, not OLTP workloads. Hence, in the rest of the chapter we are going to focus our discussion on storage solutions for analytical workloads. Next, let's see how Apache Spark can be used to read from and write to databases.

Reading from and Writing to Databases Using Apache Spark

Thanks to the ever-growing ecosystem of connectors, Apache Spark can connect to a wide variety of databases for reading and writing data. For databases that have JDBC drivers (e.g., PostgreSQL, MySQL), you can use the built-in JDBC data source along with the appropriate JDBC driver jars to access the data. For many other modern databases (e.g., Azure Cosmos DB, Snowflake), there are dedicated connectors that you can invoke using the appropriate format name. Several examples were discussed in detail in Chapter 5. This makes it very easy to augment your data warehouses and databases with workloads and use cases based on Apache Spark.

Limitations of Databases

Since the last century, databases and SQL queries have been known as great building solutions for BI workloads. However, the last decade has seen two major new trends in analytical workloads:

Growth in data sizes

With the advent of big data, there has been a global trend in the industry to measure and collect everything (page views, clicks, etc.) in order to understand trends and user behaviors. As a result, the amount of data collected by any company or organization has increased from gigabytes a couple of decades ago to terabytes and petabytes today.

Growth in the diversity of analytics

Along with the increase in data collection, there is a need for deeper insights. This has led to an explosive growth of complex analytics like machine learning and deep learning.

Databases have been shown to be rather inadequate at accommodating these new trends, because of the following limitations:

Databases are extremely expensive to scale out

Although databases are extremely efficient at processing data on a single machine, the rate of growth of data volumes has far outpaced the growth in performance capabilities of a single machine. The only way forward for processing engines is to scale out—that is, use multiple machines to process data in parallel. However, most databases, especially the open source ones, are

not designed for scaling out to perform distributed processing. The few industrial database solutions that can remotely keep up with the processing requirements tend to be proprietary solutions running on specialized hardware, and are therefore very expensive to acquire and maintain.

Databases do not support non–SQL based analytics very well

Databases store data in complex (often proprietary) formats that are typically highly optimized for only that database's SQL processing engine to read. This means other processing tools, like machine learning and deep learning systems, cannot efficiently access the data (except by inefficiently reading all the data from the database). Nor can databases be easily extended to perform non–SQL based analytics like machine learning.

These limitations of databases led to the development of a completely different approach to storing data, known as *data lakes*.

Data Lakes

In contrast to most databases, a data lake is a distributed storage solution that runs on commodity hardware and easily scales out horizontally. In this section, we will start with a discussion of how data lakes satisfy the requirements of modern workloads, then see how Apache Spark integrates with data lakes to make workloads scale to data of any size. Finally, we will explore the impact of the architectural sacrifices made by data lakes to achieve scalability.

A Brief Introduction to Data Lakes

The data lake architecture, unlike that of databases, decouples the distributed storage system from the distributed compute system. This allows each system to scale out as needed by the workload. Furthermore, the data is saved as files with open formats, such that any processing engine can read and write them using standard APIs. This idea was popularized in the late 2000s by the Hadoop File System (HDFS) from the Apache Hadoop project, which itself was heavily inspired by the research paper "The Google File System" by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung.

Organizations build their data lakes by independently choosing the following:

Storage system

They choose to either run HDFS on a cluster of machines or use any cloud object store (e.g., AWS S3, Azure Data Lake Storage, or Google Cloud Storage).

File format

Depending on the downstream workloads, the data is stored as files in either structured (e.g., Parquet, ORC), semi-structured (e.g., JSON), or sometimes even unstructured formats (e.g., text, images, audio, video).

Processing engine(s)

Again, depending on the kinds of analytical workloads to be performed, a processing engine is chosen. This can either be a batch processing engine (e.g., Spark, Presto, Apache Hive), a stream processing engine (e.g., Spark, Apache Flink), or a machine learning library (e.g., Spark MLlib, scikit-learn, R).

This flexibility—the ability to choose the storage system, open data format, and processing engine that are best suited to the workload at hand—is the biggest advantage of data lakes over databases. On the whole, for the same performance characteristics, data lakes often provide a much cheaper solution than databases. This key advantage has led to the explosive growth of the big data ecosystem. In the next section, we will discuss how you can use Apache Spark to read and write common file formats on any storage system.

Reading from and Writing to Data Lakes using Apache Spark

Apache Spark is one of the best processing engines to use when building your own data lake, because it provides all the key features they require:

Support for diverse workloads

Spark provides all the necessary tools to handle a diverse range of workloads, including batch processing, ETL operations, SQL workloads using Spark SQL, stream processing using Structured Streaming (discussed in Chapter 8), and machine learning using MLlib (discussed in Chapter 10), among many others.

Support for diverse file formats

In Chapter 4, we explored in detail how Spark has built-in support for unstructured, semi-structured, and structured file formats.

Support for diverse filesystems

Spark supports accessing data from any storage system that supports Hadoop's FileSystem APIs. Since this API has become the de facto standard in the big data ecosystem, most cloud and on-premises storage systems provide implementations for it—which means Spark can read from and write to most storage systems.

However, for many filesystems (especially those based on cloud storage, like AWS S3), you have to configure Spark such that it can access the filesystem in a secure manner. Furthermore, cloud storage systems often do not have the same file operation semantics expected from a standard filesystem (e.g., eventual consistency in S3), which can lead to inconsistent results if you do not configure Spark accordingly. See the documentation on cloud integration for details.

Limitations of Data Lakes

Data lakes are not without their share of flaws, the most egregious of which is the lack of transactional guarantees. Specifically, data lakes fail to provide ACID guarantees on:

Atomicity and isolation

Processing engines write data in data lakes as many files in a distributed manner. If the operation fails, there is no mechanism to roll back the files already written, thus leaving behind potentially corrupted data (the problem is exacerbated when concurrent workloads modify the data because it is very difficult to provide isolation across files without higher-level mechanisms).

Consistency

Lack of atomicity on failed writes further causes readers to get an inconsistent view of the data. In fact, it is hard to ensure data quality even in successfully written data. For example, a very common issue with data lakes is accidentally writing out data files in a format and schema inconsistent with existing data.

To work around these limitations of data lakes, developers employ all sorts of tricks. Here are a few examples:

- Large collections of data files in data lakes are often "partitioned" by subdirectories based on a column's value (e.g., a large Parquet-formatted Hive table partitioned by date). To achieve atomic modifications of existing data, often entire subdirectories are rewritten (i.e.,

written to a temporary directory, then references swapped) just to update or delete a few records.
- The schedules of data update jobs (e.g., daily ETL jobs) and data querying jobs (e.g., daily reporting jobs) are often staggered to avoid concurrent access to the data and any inconsistencies caused by it.

Attempts to eliminate such practical issues have led to the development of new systems, such as lakehouses.

Lakehouses: The Next Step in the Evolution of Storage Solutions

The *lakehouse* is a new paradigm that combines the best elements of data lakes and data warehouses for OLAP workloads. Lakehouses are enabled by a new system design that provides data management features similar to databases directly on the low-cost, scalable storage used for data lakes. More specifically, they provide the following features:

Transaction support

Similar to databases, lakehouses provide ACID guarantees in the presence of concurrent workloads.

Schema enforcement and governance

Lakehouses prevent data with an incorrect schema being inserted into a table, and when needed, the table schema can be explicitly evolved to accommodate ever-changing data. The system should be able to reason about data integrity, and it should have robust governance and auditing mechanisms.

Support for diverse data types in open formats

Unlike databases, but similar to data lakes, lakehouses can store, refine, analyze, and access all types of data needed for many new data applications, be it structured, semi-structured, or unstructured. To enable a wide variety of tools to access it directly and efficiently, the data must be stored in open formats with standardized APIs to read and write them.

Support for diverse workloads

Powered by the variety of tools reading data using open APIs, lakehouses enable diverse workloads to operate on data in a single repository. Breaking down isolated data silos (i.e., multiple repositories for different categories of data) enables developers to more easily build diverse and complex data solutions, from traditional SQL and streaming analytics to machine learning.

Support for upserts and deletes

Complex use cases like change-data-capture (CDC) and slowly changing dimension (SCD) operations require data in tables to be continuously updated. Lakehouses allow data to be concurrently deleted and updated with transactional guarantees.

Data governance

Lakehouses provide the tools with which you can reason about data integrity and audit all the data changes for policy compliance.

Currently, there are a few open source systems, such as Apache Hudi, Apache Iceberg, and Delta Lake, that can be used to build lakehouses with these properties. At a very high level, all three projects have a similar architecture inspired by well-known database principles. They are all open data storage formats that do the following:

- Store large volumes of data in structured file formats on scalable filesystems.
- Maintain a transaction log to record a timeline of atomic changes to the data (much like databases).
- Use the log to define versions of the table data and provide snapshot isolation guarantees between readers and writers.
- Support reading and writing to tables using Apache Spark.

Within these broad strokes, each project has unique characteristics in terms of APIs, performance, and the level of integration with Apache Spark's data source APIs. We will explore them next. Note that all of these projects are evolving fast, and therefore some of the descriptions may be outdated at the time you are reading them. Refer to the online documentation for each project for the most up-to-date information.

Apache Hudi

Initially built by Uber Engineering, Apache Hudi—an acronym for Hadoop Update Delete and Incremental—is a data storage format that is designed for incremental upserts and deletes over key/value-style data. The data is stored as a combination of columnar formats (e.g., Parquet files) and row-based formats (e.g., Avro files for recording incremental changes over Parquet files). Besides the common features mentioned earlier, it supports:

- Upserting with fast, pluggable indexing
- Atomic publishing of data with rollback support
- Reading incremental changes to a table
- Savepoints for data recovery
- File size and layout management using statistics
- Async compaction of row and columnar data

Apache Iceberg

Originally built at Netflix, Apache Iceberg is another open storage format for huge data sets. However, unlike Hudi, which focuses on upserting key/value data, Iceberg focuses more on general-purpose data storage that scales to petabytes in a single table and has schema evolution properties. Specifically, it provides the following additional features (besides the common ones):

- Schema evolution by adding, dropping, updating, renaming, and reordering of columns, fields, and/or nested structures
- Hidden partitioning, which under the covers creates the partition values for rows in a table
- Partition evolution, where it automatically performs a metadata operation to update the table layout as data volume or query patterns change
- Time travel, which allows you to query a specific table snapshot by ID or by timestamp
- Rollback to previous versions to correct errors
- Serializable isolation, even between multiple concurrent writers

Delta Lake

Delta Lake is an open source project hosted by the Linux Foundation, built by the original creators of Apache Spark. Similar to the others, it is an open data storage format that provides transactional guarantees and enables schema enforcement and evolution. It also provides several other interesting features, some of which are unique. Delta Lake supports:

- Streaming reading from and writing to tables using Structured Streaming sources and sinks

- Update, delete, and merge (for upserts) operations, even in Java, Scala, and Python APIs
- Schema evolution either by explicitly altering the table schema or by implicitly merging a DataFrame's schema to the table's during the DataFrame's write. (In fact, the merge operation in Delta Lake supports advanced syntax for conditional updates/inserts/deletes, updating all columns together, etc., as you'll see later in the chapter.)
- Time travel, which allows you to query a specific table snapshot by ID or by timestamp
- Rollback to previous versions to correct errors
- Serializable isolation between multiple concurrent writers performing any SQL, batch, or streaming operations

In the rest of this chapter, we are going to explore how such a system, along with Apache Spark, can be used to build a lakehouse that provides the aforementioned properties. Of these three systems, so far Delta Lake has the tightest integration with Apache Spark data sources (both for batch and streaming workloads) and SQL operations (e.g., `MERGE`). Hence, we will use Delta Lake as the vehicle for further exploration.

NOTE

This project is called Delta Lake because of its analogy to streaming. Streams flow into the sea to create deltas—this is where all of the sediments accumulate, and thus where the valuable crops are grown. Jules S. Damji (one of our coauthors) came up with this!

Building Lakehouses with Apache Spark and Delta Lake

In this section, we are going to take a quick look at how Delta Lake and Apache Spark can be used to build lakehouses. Specifically, we will explore the following:

- Reading and writing Delta Lake tables using Apache Spark
- How Delta Lake allows concurrent batch and streaming writes with ACID guarantees

- How Delta Lake ensures better data quality by enforcing schema on all writes, while allowing for explicit schema evolution
- Building complex data pipelines using update, delete, and merge operations, all of which ensure ACID guarantees
- Auditing the history of operations that modified a Delta Lake table and traveling back in time by querying earlier versions of the table

The data we will use in this section is a modified version (a subset of columns in Parquet format) of the public Lending Club Loan Data.[1] It includes all funded loans from 2012 to 2017. Each loan record includes applicant information provided by the applicant as well as the current loan status (current, late, fully paid, etc.) and latest payment information.

Configuring Apache Spark with Delta Lake

You can configure Apache Spark to link to the Delta Lake library in one of the following ways:

Set up an interactive shell

If you're using Apache Spark 3.0, you can start a PySpark or Scala shell with Delta Lake by using the following command-line argument:

```
--packages io.delta:delta-core_2.12:0.7.0
```

For example:

```
pyspark --packages io.delta:delta-core_2.12:0.7.0
```

If you are running Spark 2.4, you have to use Delta Lake 0.6.0.

Set up a standalone Scala/Java project using Maven coordinates

If you want to build a project using Delta Lake binaries from the Maven Central repository, you can add the following Maven coordinates to the project dependencies:

```
    <dependency>
    <groupId>io.delta</groupId>
    <artifactId>delta-core_2.12</artifactId>
    <version>0.7.0</version>
  </dependency>
```

Again, if you are running Spark 2.4 you have to use Delta Lake 0.6.0.

NOTE

See the Delta Lake documentation for the most up-to-date information.

Loading Data into a Delta Lake Table

If you are used to building data lakes with Apache Spark and any of the structured data formats—say, Parquet—then it is very easy to migrate existing workloads to use the Delta Lake format. All you have to do is change all the DataFrame read and write operations to use `format("delta")` instead of `format("parquet")`. Let's try this out with some of the aforementioned loan data, which is available as a Parquet file. First let's read this data and save it as a Delta Lake table:

```
// In Scala
// Configure source data path
val sourcePath = "/databricks-datasets/learning-spark-v2/loans/
  loan-risks.snappy.parquet"

// Configure Delta Lake path
val deltaPath = "/tmp/loans_delta"

// Create the Delta table with the same loans data
spark
  .read
  .format("parquet")
  .load(sourcePath)
  .write
  .format("delta")
  .save(deltaPath)
```

```
// Create a view on the data called loans_delta
spark
 .read
 .format("delta")
 .load(deltaPath)
 .createOrReplaceTempView("loans_delta")


# In Python
# Configure source data path
sourcePath = "/databricks-datasets/learning-spark-v2/loans/
  loan-risks.snappy.parquet"

# Configure Delta Lake path
deltaPath = "/tmp/loans_delta"

# Create the Delta Lake table with the same loans data
(spark.read.format("parquet").load(sourcePath)
  .write.format("delta").save(deltaPath))

# Create a view on the data called loans_delta
spark.read.format("delta").load(deltaPath).createOrReplaceTempView("loans_delta"
```

Now we can read and explore the data as easily as any other table:

```
// In Scala/Python

// Loans row count
spark.sql("SELECT count(*) FROM loans_delta").show()

+--------+
|count(1)|
+--------+
|   14705|
+--------+

// First 5 rows of loans table
spark.sql("SELECT * FROM loans_delta LIMIT 5").show()

+-------+-----------+---------+----------+
|loan_id|funded_amnt|paid_amnt|addr_state|
+-------+-----------+---------+----------+
|      0|       1000|   182.22|        CA|
|      1|       1000|   361.19|        WA|
```

```
|      2|       1000|   176.26|        TX|
|      3|       1000|   1000.0|        OK|
|      4|       1000|   249.98|        PA|
+-------+-----------+---------+----------+
```

Loading Data Streams into a Delta Lake Table

As with static DataFrames, you can easily modify your existing Structured Streaming jobs to write to and read from a Delta Lake table by setting the format to `"delta"`. Say you have a stream of new loan data as a DataFrame named `newLoanStreamDF`, which has the same schema as the table. You can append to the table as follows:

```
// In Scala
import org.apache.spark.sql.streaming._

val newLoanStreamDF = ...   // Streaming DataFrame with new loans data
val checkpointDir = ...     // Directory for streaming checkpoints
val streamingQuery = newLoanStreamDF.writeStream
  .format("delta")
  .option("checkpointLocation", checkpointDir)
  .trigger(Trigger.ProcessingTime("10 seconds"))
  .start(deltaPath)
```

```
# In Python
newLoanStreamDF = ...   # Streaming DataFrame with new loans data
checkpointDir = ...     # Directory for streaming checkpoints
streamingQuery = (newLoanStreamDF.writeStream
    .format("delta")
    .option("checkpointLocation", checkpointDir)
    .trigger(processingTime = "10 seconds")
    .start(deltaPath))
```

With this format, just like any other, Structured Streaming offers end-to-end exactly-once guarantees. However, Delta Lake has a few additional advantages over traditional formats like JSON, Parquet, or ORC:

It allows writes from both batch and streaming jobs into the same table

> With other formats, data written into a table from a Structured Streaming job will overwrite any existing data in the table. This is because the metadata

maintained in the table to ensure exactly-once guarantees for streaming writes does not account for other nonstreaming writes. Delta Lake's advanced metadata management allows both batch and streaming data to be written.

It allows multiple streaming jobs to append data to the same table

The same limitation of metadata with other formats also prevents multiple Structured Streaming queries from appending to the same table. Delta Lake's metadata maintains transaction information for each streaming query, thus enabling any number of streaming queries to concurrently write into a table with exactly-once guarantees.

It provides ACID guarantees even under concurrent writes

Unlike built-in formats, Delta Lake allows concurrent batch and streaming operations to write data with ACID guarantees.

Enforcing Schema on Write to Prevent Data Corruption

A common problem with managing data with Spark using common formats like JSON, Parquet, and ORC is accidental data corruption caused by writing incorrectly formatted data. Since these formats define the data layout of individual files and not of the entire table, there is no mechanism to prevent any Spark job from writing files with different schemas into existing tables. This means there are no guarantees of consistency for the entire table of many Parquet files.

The Delta Lake format records the schema as table-level metadata. Hence, all writes to a Delta Lake table can verify whether the data being written has a schema compatible with that of the table. If it is not compatible, Spark will throw an error before any data is written and committed to the table, thus preventing such accidental data corruption. Let's test this by trying to write some data with an additional column, `closed`, that signifies whether the loan has been terminated. Note that this column does not exist in the table:

```
// In Scala
val loanUpdates = Seq(
    (1111111L, 1000, 1000.0, "TX", false),
    (2222222L, 2000, 0.0, "CA", true))
  .toDF("loan_id", "funded_amnt", "paid_amnt", "addr_state", "closed")
```

```
loanUpdates.write.format("delta").mode("append").save(deltaPath)


# In Python
from pyspark.sql.functions import *

cols = ['loan_id', 'funded_amnt', 'paid_amnt', 'addr_state', 'closed']
items = [
(1111111, 1000, 1000.0, 'TX', True),
(2222222, 2000, 0.0, 'CA', False)
]

loanUpdates = (spark.createDataFrame(items, cols)
  .withColumn("funded_amnt", col("funded_amnt").cast("int")))
loanUpdates.write.format("delta").mode("append").save(deltaPath)
```

This write will fail with the following error message:

```
org.apache.spark.sql.AnalysisException: A schema mismatch detected when writing
  to the Delta table (Table ID: 48bfa949-5a09-49ce-96cb-34090ab7d695).
To enable schema migration, please set:
'.option("mergeSchema", "true")'.

Table schema:
root
-- loan_id: long (nullable = true)
-- funded_amnt: integer (nullable = true)
-- paid_amnt: double (nullable = true)
-- addr_state: string (nullable = true)


Data schema:
root
-- loan_id: long (nullable = true)
-- funded_amnt: integer (nullable = true)
-- paid_amnt: double (nullable = true)
-- addr_state: string (nullable = true)
-- closed: boolean (nullable = true)
```

This illustrates how Delta Lake blocks writes that do not match the schema of the table. However, it also gives a hint about how to actually

evolve the schema of the table using the option `mergeSchema`, as discussed next.

Evolving Schemas to Accommodate Changing Data

In our world of ever-changing data, it is possible that we might want to add this new column to the table. This new column can be explicitly added by setting the option `"mergeSchema"` to `"true"`:

```
// In Scala
loanUpdates.write.format("delta").mode("append")
  .option("mergeSchema", "true")
  .save(deltaPath)
```

```
# In Python
(loanUpdates.write.format("delta").mode("append")
  .option("mergeSchema", "true")
  .save(deltaPath))
```

With this, the column `closed` will be added to the table schema, and new data will be appended. When existing rows are read, the value of the new column is considered as `NULL`. In Spark 3.0, you can also use the SQL DDL command `ALTER TABLE` to add and modify columns.

Transforming Existing Data

Delta Lake supports the DML commands `UPDATE`, `DELETE`, and `MERGE`, which allow you to build complex data pipelines. These commands can be invoked using Java, Scala, Python, and SQL, giving users the flexibility of using the commands with any APIs they are familiar with, using either DataFrames or tables. Furthermore, each of these data modification operations ensures ACID guarantees.

Let's explore this with a few examples of real-world use cases.

Updating data to fix errors

A common use case when managing data is fixing errors in the data. Suppose, upon reviewing the data, we realized that all of the loans assigned to `addr_state = 'OR'` should have been assigned to `addr_state = 'WA'`. If the loan table were a Parquet table, then to do such an update we would need to:

1. Copy all of the rows that are not affected into a new table.
2. Copy all of the rows that are affected into a DataFrame, then perform the data modification.
3. Insert the previously noted DataFrame's rows into the new table.
4. Remove the old table and rename the new table to the old table name.

In Spark 3.0, which added direct support for DML SQL operations like `UPDATE`, `DELETE`, and `MERGE`, instead of manually performing all these steps you can simply run the SQL `UPDATE` command. However, with a Delta Lake table, users can run this operation too, by using Delta Lake's programmatic APIs as follows:

```
// In Scala
import io.delta.tables.DeltaTable
import org.apache.spark.sql.functions._

val deltaTable = DeltaTable.forPath(spark, deltaPath)
deltaTable.update(
  col("addr_state") === "OR",
  Map("addr_state" -> lit("WA")))
```

```
# In Python
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, deltaPath)
deltaTable.update("addr_state = 'OR'",  {"addr_state": "'WA'"})
```

Deleting user-related data

With data protection policies like the EU's General Data Protection Regulation (GDPR) coming into force, it is more important now than ever to be able to delete user data from all your tables. Say it is mandated that you have to delete the data on all loans that have been fully paid off. With Delta Lake, you can do the following:

```
// In Scala
val deltaTable = DeltaTable.forPath(spark, deltaPath)
deltaTable.delete("funded_amnt >= paid_amnt")
```

```
# In Python
deltaTable = DeltaTable.forPath(spark, deltaPath)
deltaTable.delete("funded_amnt >= paid_amnt")
```

Similar to updates, with Delta Lake and Apache Spark 3.0 you can directly run the `DELETE` SQL command on the table.

Upserting change data to a table using merge()

A common use case is change data capture, where you have to replicate row changes made in an OLTP table to another table for OLAP workloads. To continue with our loan data example, say we have another table of new loan information, some of which are new loans and others of which are updates to existing loans. In addition, let's say this `changes` table has the same schema as the `loan_delta` table. You can upsert these changes into the table using the `DeltaTable.merge()` operation, which is based on the `MERGE` SQL command:

```
// In Scala
deltaTable
  .alias("t")
  .merge(loanUpdates.alias("s"), "t.loan_id = s.loan_id")
  .whenMatched.updateAll()
  .whenNotMatched.insertAll()
  .execute()
```

```
# In Python
(deltaTable
  .alias("t")
  .merge(loanUpdates.alias("s"), "t.loan_id = s.loan_id")
  .whenMatchedUpdateAll()
  .whenNotMatchedInsertAll()
  .execute())
```

As a reminder, you can run this as a SQL `MERGE` command starting with Spark 3.0. Furthermore, if you have a stream of such captured changes, you can continuously apply those changes using a Structured Streaming query. The query can read the changes in micro-batches (see Chapter 8) from any streaming source, and use `foreachBatch()` to apply the changes in each micro-batch to the Delta Lake table.

Deduplicating data while inserting using insert-only merge

The merge operation in Delta Lake supports an extended syntax beyond that specified by the ANSI standard, including advanced features like the following:

Delete actions

For example, `MERGE ... WHEN MATCHED THEN DELETE`.

Clause conditions

For example, `MERGE ... WHEN MATCHED AND <condition> THEN ...`.

Optional actions

All the `MATCHED` and `NOT MATCHED` clauses are optional.

Star syntax

For example, `UPDATE *` and `INSERT *` to update/insert all the columns in the target table with matching columns from the source data set. The equivalent Delta Lake APIs are `updateAll()` and `insertAll()`, which we saw in the previous section.

This allows you to express many more complex use cases with little code. For example, say you want to backfill the `loan_delta` table with historical data on past loans. But some of the historical data may already have been inserted in the table, and you don't want to update those records because they may contain more up-to-date information. You can dedupli-

cate by the `loan_id` while inserting by running the following merge operation with only the `INSERT` action (since the `UPDATE` action is optional):

```
// In Scala
deltaTable
  .alias("t")
  .merge(historicalUpdates.alias("s"), "t.loan_id = s.loan_id")
  .whenNotMatched.insertAll()
  .execute()


# In Python
(deltaTable
  .alias("t")
  .merge(historicalUpdates.alias("s"), "t.loan_id = s.loan_id")
  .whenNotMatchedInsertAll()
  .execute())
```

There are even more complex use cases, like CDC with deletes and SCD tables, that are made simple with the extended merge syntax. Refer to the documentation for more details and examples.

Auditing Data Changes with Operation History

All of the changes to your Delta Lake table are recorded as commits in the table's transaction log. As you write into a Delta Lake table or directory, every operation is automatically versioned. You can query the table's operation history as noted in the following code snippet:

```
// In Scala/Python
deltaTable.history().show()
```

By default this will show a huge table with many versions and a lot of columns. Let's instead print some of the key columns of the last three operations:

```
// In Scala
deltaTable
```

```
  .history(3)
  .select("version", "timestamp", "operation", "operationParameters")
  .show(false)


# In Python
(deltaTable
  .history(3)
  .select("version", "timestamp", "operation", "operationParameters")
  .show(truncate=False))
```

This will generate the following output:

```
+-------+-----------+---------+-----------------------------------------+
|version|timestamp  |operation|operationParameters                      |
+-------+-----------+---------+-----------------------------------------+
|5      |2020-04-07 |MERGE    |[predicate -> (t.`loan_id` = s.`loan_id`)] |
|4      |2020-04-07 |MERGE    |[predicate -> (t.`loan_id` = s.`loan_id`)] |
|3      |2020-04-07 |DELETE   |[predicate -> ["(CAST(`funded_amnt` ...    |
+-------+-----------+---------+-----------------------------------------+
```

Note the `operation` and `operationParameters` that are useful for auditing the changes.

Querying Previous Snapshots of a Table with Time Travel

You can query previous versioned snapshots of a table by using the `DataFrameReader` options `"versionAsOf"` and `"timestampAsOf"`. Here are a few examples:

```
// In Scala
spark.read
  .format("delta")
  .option("timestampAsOf", "2020-01-01")  // timestamp after table creation
  .load(deltaPath)

spark.read.format("delta")
  .option("versionAsOf", "4")
  .load(deltaPath)
```

```
# In Python
(spark.read
  .format("delta")
  .option("timestampAsOf", "2020-01-01")  # timestamp after table creation
  .load(deltaPath))

(spark.read.format("delta")
  .option("versionAsOf", "4")
  .load(deltaPath))
```

This is useful in a variety of situations, such as:

- Reproducing machine learning experiments and reports by rerunning the job on a specific table version
- Comparing the data changes between different versions for auditing
- Rolling back incorrect changes by reading a previous snapshot as a DataFrame and overwriting the table with it

Summary

This chapter examined the possibilities for building reliable data lakes using Apache Spark. To recap, databases have solved data problems for a long time, but they fail to fulfill the diverse requirements of modern use cases and workloads. Data lakes were built to alleviate some of the limitations of databases, and Apache Spark is one of the best tools to build them with. However, data lakes still lack some of the key features provided by databases (e.g., ACID guarantees). Lakehouses are the next generation of data solutions, which aim to provide the best features of databases and data lakes and meet all the requirements of diverse use cases and workloads.

We briefly explored a couple of open source systems (Apache Hudi and Apache Iceberg) that can be used to build lakehouses, then took a closer look at Delta Lake, a file-based open source storage format that, along with Apache Spark, is a great building block for lakehouses. As you saw, it provides the following:

- Transactional guarantees and schema management, like databases

- Scalability and openness, like data lakes
- Support for concurrent batch and streaming workloads with ACID guarantees
- Support for transformation of existing data using update, delete, and merge operations that ensure ACID guarantees
- Support for versioning, auditing of operation history, and querying of previous versions

In the next chapter, we'll explore how to begin building ML models using Spark's MLlib.

1 A full view of the data is available at this Excel file.