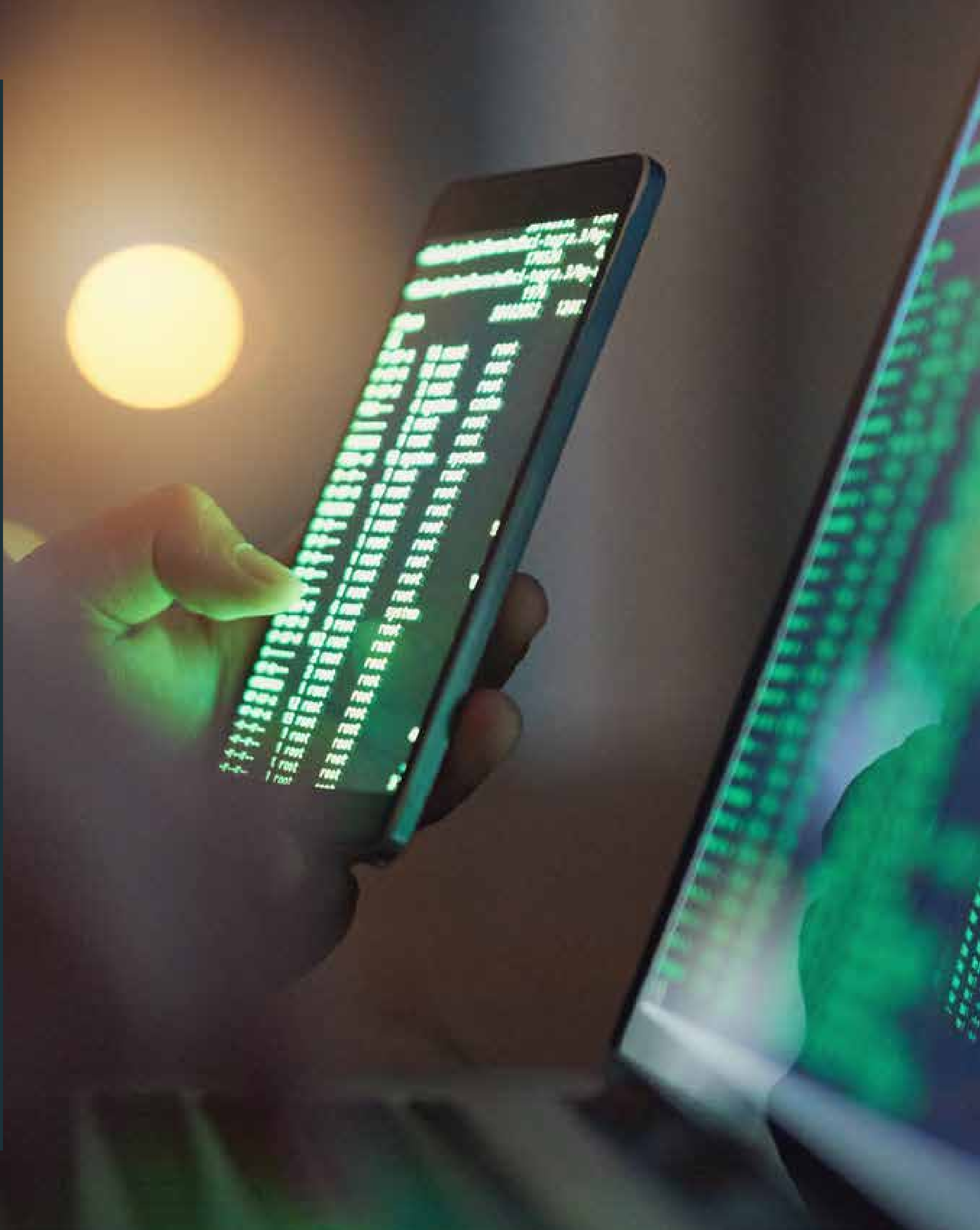
The Delta Lake Series Features

Use Delta Lake's robust features to reliably manage your data

databricks



What's inside?

The Delta Lake Series of eBooks is published by Databricks to help leaders and practitioners understand the full capabilities of Delta Lake as well as the landscape it resides in. This eBook, **The Delta Lake Series — Features**, focuses on Delta Lake's robust features so you can use them to your benefit.

What's next?

After reading this eBook, you'll not only understand what Delta Lake offers, but you'll also understand how its features result in substantial performance improvements.

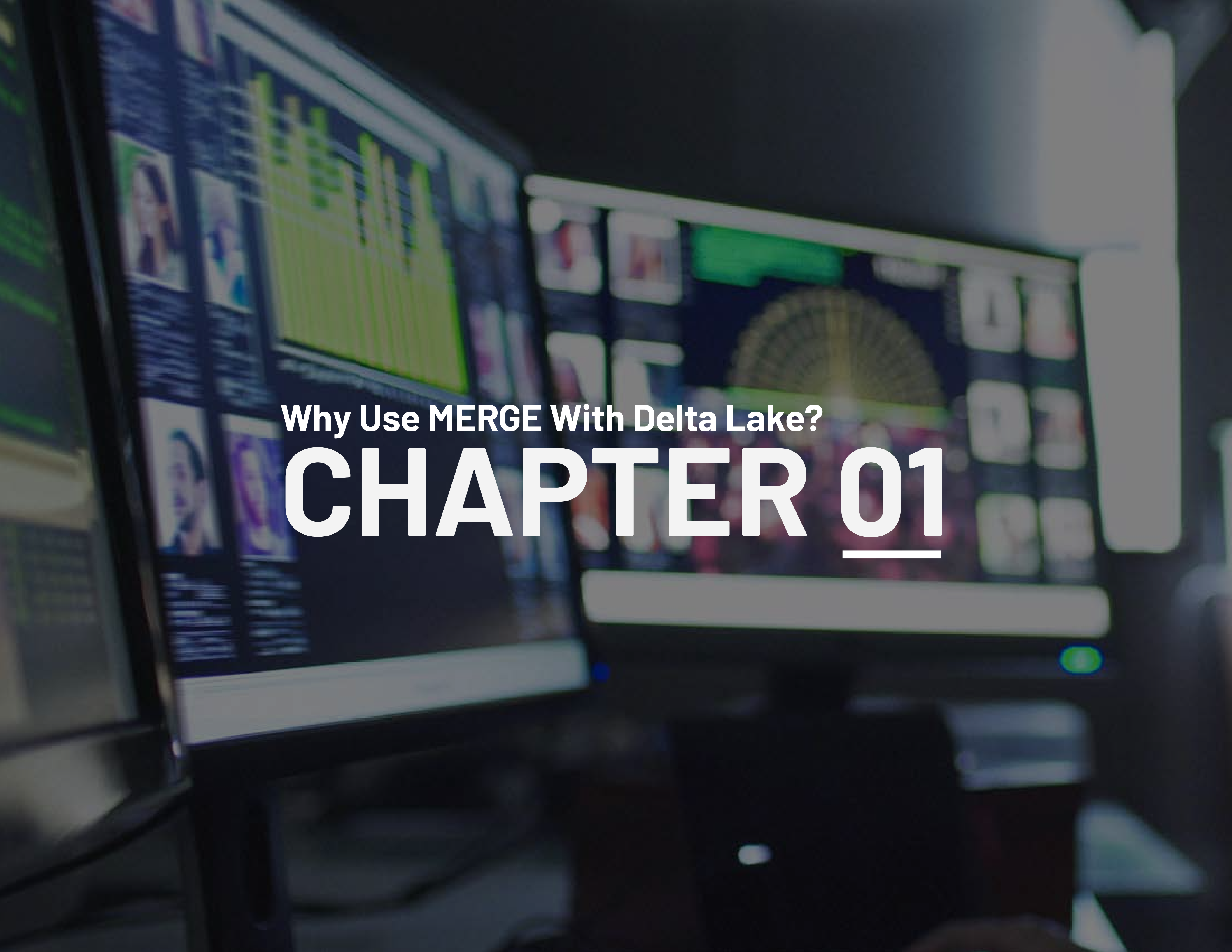
Here's what you'll find inside

What is Delta Lake?

Delta Lake is a unified data management system that brings data reliability and fast analytics to cloud data lakes. Delta Lake runs on top of existing data lakes and is fully compatible with Apache Spark™ APIs.

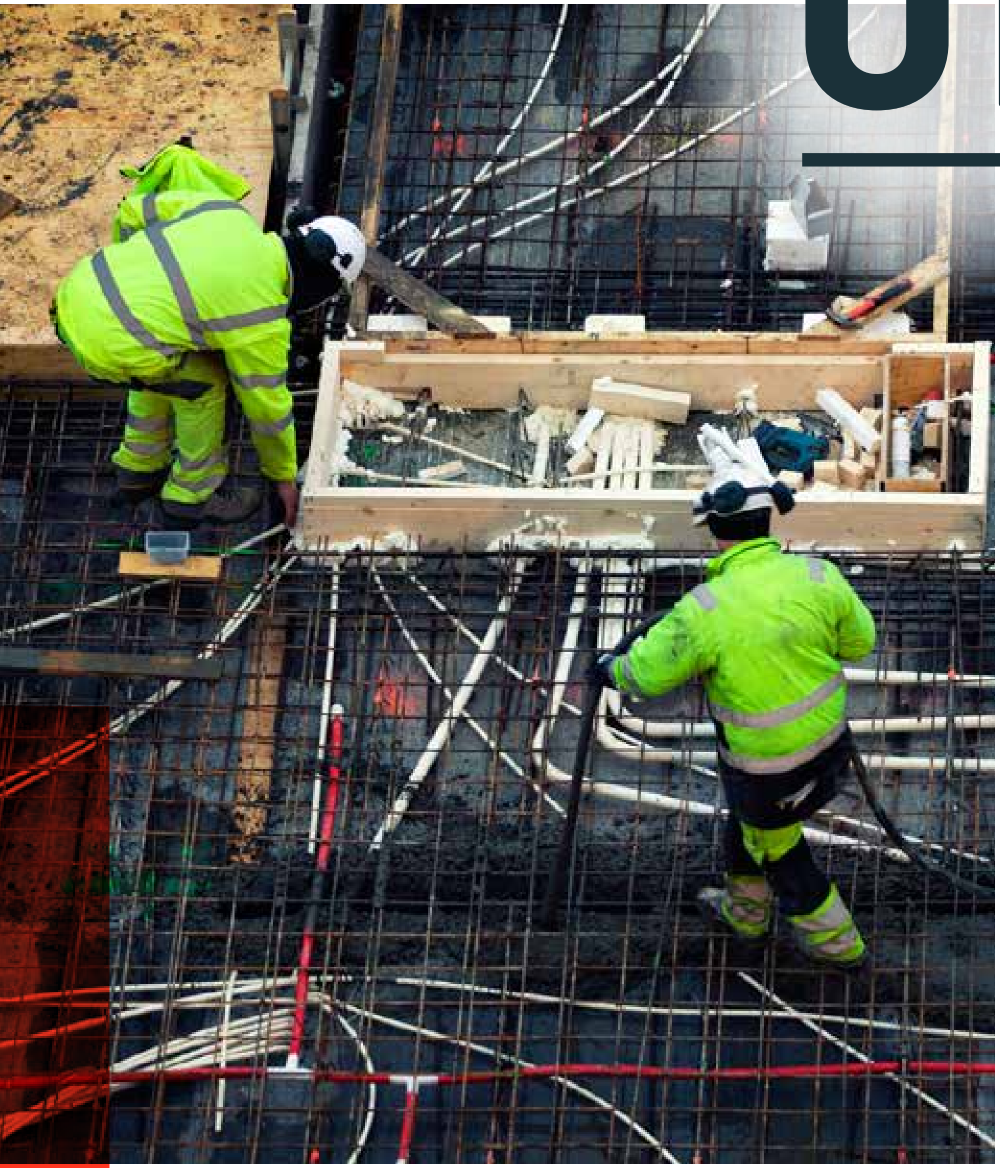
At Databricks, we've seen how Delta Lake can bring reliability, performance and lifecycle management to data lakes. Our customers have found that Delta Lake solves for challenges around malformed data ingestion, difficulties deleting data for compliance, or issues modifying data for data capture.

With Delta Lake, you can accelerate the velocity that high-quality data can get into your data lake and the rate that teams can leverage that data with a secure and scalable cloud service.



Why Use MERGE With Delta Lake?

CHAPTER 01



01 Why Use MERGE With Delta Lake?

Delta Lake, the next-generation engine built on top of Apache Spark, supports the MERGE command, which allows you to efficiently upsert and delete records in your data lakes.

MERGE dramatically simplifies how a number of common data pipelines can be built — all the complicated multi-hop processes that inefficiently rewrote entire partitions can now be replaced by simple MERGE queries.

This finer-grained update capability simplifies how you build your big data pipelines for various use cases ranging from change data capture to GDPR. You no longer need to write complicated logic to overwrite tables and overcome a lack of snapshot isolation.

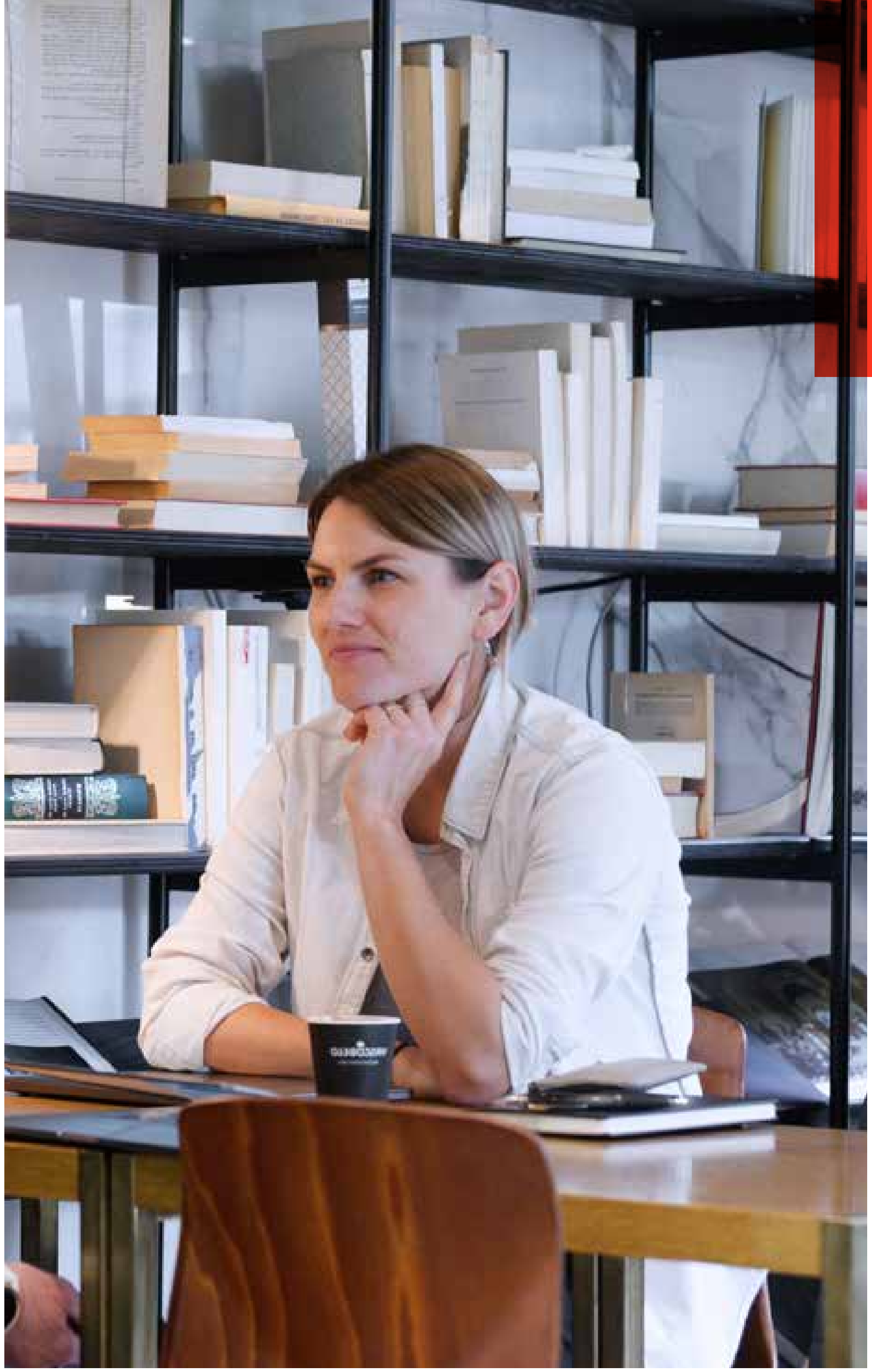
With changing data, another critical capability required is the ability to roll back, in case of bad writes. Delta Lake also offers rollback capabilities with the Time Travel feature, so that if you do a bad merge, you can easily roll back to an earlier version.

In this chapter, we'll discuss common use cases where existing data might need to be updated or deleted. We'll also explore the challenges inherent to upserts and explain how MERGE can address them.

When are upserts necessary?

There are a number of common use cases where existing data in a data lake needs to be updated or deleted:

- **General Data Protection Regulation (GDPR) compliance:** With the introduction of the right to be forgotten (also known as data erasure) in GDPR, organizations must remove a user's information upon request. This data erasure includes deleting user information in the data lake as well.
- **Change data capture from traditional databases:** In a service-oriented architecture, typically web and mobile applications are served by microservices built on traditional SQL/NoSQL databases that are optimized for low latency. One of the biggest challenges organizations face is joining data across these various siloed data systems, and hence data engineers build pipelines to consolidate all data sources into a central data lake to facilitate analytics. These pipelines often have to periodically read changes made on a traditional SQL/NoSQL table and apply them to corresponding tables in the data lake. Such changes can take various forms: Tables with slowly changing dimensions, change data capture of all inserted/updated/deleted rows, etc.
- **Sessionization:** Grouping multiple events into a single session is a common use case in many areas ranging from product analytics to targeted advertising to predictive maintenance. Building continuous applications to track sessions and recording the results that write into data lakes is difficult because data lakes have always been optimized for appending data.
- **De-duplication:** A common data pipeline use case is to collect system logs into a Delta Lake table by appending data to the table. However, often the sources can generate duplicate records and downstream de-duplication steps are needed to take care of them.



Why upserts into data lakes have traditionally been challenging

Since data lakes are fundamentally based on files, they have always been optimized for appending data rather than for changing existing data. Hence, building the above use case has always been challenging.

Users typically read the entire table (or a subset of partitions) and then overwrite them. Therefore, every organization tries to reinvent the wheel for their requirement by handwriting complicated queries in SQL, Spark, etc. This approach is:

- **Inefficient:** Reading and rewriting entire partitions (or entire tables) to update a few records causes pipelines to be slow and costly. Hand-tuning the table layout and query optimization is tedious and requires deep domain knowledge.
- **Possibly incorrect:** Handwritten code modifying data is very prone to logical and human errors. For example, multiple pipelines concurrently modifying the same table without any transactional support can lead to unpredictable data inconsistencies and in the worst case, data losses. Often, even a single handwritten pipeline can easily cause data corruptions due to errors in encoding the business logic.
- **Hard to maintain:** Fundamentally such handwritten code is hard to understand, keep track of and maintain. In the long term, this alone can significantly increase the organizational and infrastructural costs.

Introducing MERGE in Delta Lake

With Delta Lake, you can easily address the use cases above without any of the aforementioned problems using the following MERGE command:

```
MERGE INTO
USING
ON
[ WHEN MATCHED [ AND ] THEN ]
[ WHEN MATCHED [ AND ] THEN ]
[ WHEN NOT MATCHED [ AND ] THEN ]
```

```
where


=
DELETE |
UPDATE SET * |
UPDATE SET column1 = value1 [, column2 = value2 ...]


=
INSERT * |
INSERT (column1 [, column2 ...]) VALUES (value1 [, value2 ...])
```

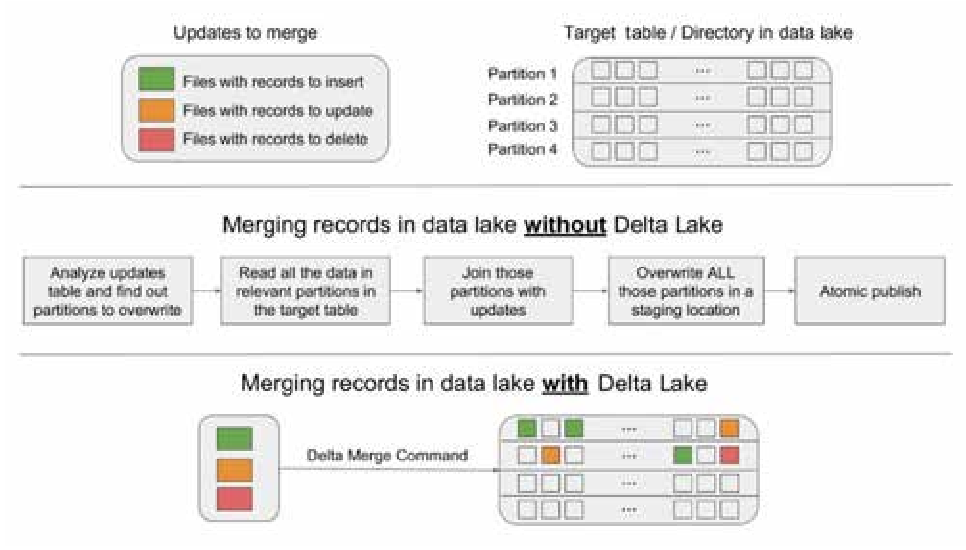
Let's understand how to use MERGE with a simple example. Suppose you have a slowly changing dimension table that maintains user information like addresses. Furthermore, you have a table of new addresses for both existing and new users. To merge all the new addresses to the main user table, you can run the following:

```
MERGE INTO users
USING updates
ON users.userId = updates.userId
WHEN MATCHED THEN
      UPDATE SET address = updates.addresses
WHEN NOT MATCHED THEN
       INSERT (userId, address) VALUES (updates.userId, updates.address)
```

This will perform exactly what the syntax says — for existing users (i.e., MATCHED clause), it will update the address column, and for new users (i.e., NOT MATCHED clause) it will insert all the columns. For large tables with TBs of data, this Delta Lake MERGE operation can be orders of magnitude faster than overwriting entire partitions or tables since Delta Lake reads only relevant files and updates them. Specifically, Delta Lake's MERGE has the following advantages:

- **Fine-grained:** The operation rewrites data at the granularity of files and not partitions. This eliminates all the complications of rewriting partitions, updating the Hive metastore with MSCK and so on.
- **Efficient:** Delta Lake's data skipping makes the MERGE efficient at finding files to rewrite, thus eliminating the need to hand-optimize your pipeline. Furthermore, Delta Lake with all its I/O and processing optimizations makes all the reading and writing data by MERGE significantly faster than similar operations in Apache Spark.
- **Transactional:** Delta Lake uses optimistic concurrency control to ensure that concurrent writers update the data correctly with ACID transactions, and concurrent readers always see a consistent snapshot of the data.

Here is a visual explanation of how MERGE compares with handwritten pipelines.


Updates to merge
Files with records to insert
Files with records to update
Files with records to delete
Target table / Directory in data lake
Partition 1
Partition 2
Partition 3
Partition 4
Merging records in data lake without Delta Lake
Analyze updates table and find out partitions to overwrite
Read all the data in relevant partitions in the target table
Join those partitions with updates
Overwrite ALL those partitions in a staging location
Atomic publish
Merging records in data lake with Delta Lake
Delta Merge Command

Simplifying use cases with MERGE

Deleting data due to GDPR

Complying with the "right to be forgotten" clause of GDPR for data in data lakes cannot get any easier. You can set up a simple scheduled job with an example code, like below, to delete all the users who have opted out of your service.

```
MERGE INTO users
USING opted_out_users
ON opted_out_users.userId = users.userId
WHEN MATCHED THEN DELETE
```

Applying change data from databases

You can easily apply all data changes — updates, deletes, inserts — generated from an external database into a Delta Lake table with the MERGE syntax as follows:

```
MERGE INTO users
USING (
SELECT userId, latest.address AS address, latest.deleted AS deleted FROM
(
SELECT userId, MAX(struct(TIME, address, deleted)) AS latest
FROM changes GROUP BY userId
)
) latestChange
ON latestChange.userId = users.userId
WHEN MATCHED AND latestChange.deleted = TRUE THEN
DELETE
WHEN MATCHED THEN
UPDATE SET address = latestChange.address
WHEN NOT MATCHED AND latestChange.deleted = FALSE THEN
INSERT (userId, address) VALUES (userId, address)
```

Updating session information from streaming pipelines

If you have streaming event data flowing in and if you want to sessionize the streaming event data and incrementally update and store sessions in a Delta Lake table, you can accomplish this using the foreachBatch in Structured Streaming and MERGE. For example, suppose you have a Structured Streaming DataFrame that computes updated session information for each user. You can start a streaming query that applies all the sessions update to a Delta Lake table as follows (Scala).

```
streamingSessionUpdatesDF.writeStream
.foreachBatch { (microBatchOutputDF: DataFrame, batchId: Long) =>
microBatchOutputDF.createOrReplaceTempView("updates")
microBatchOutputDF.sparkSession.sql(s"""
MERGE INTO sessions
USING updates
ON sessions.sessionId = updates.sessionId
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT * """)
}.start()
```

For a complete working example of each Batch and MERGE, see this notebook (Azure | AWS).

Additional resources

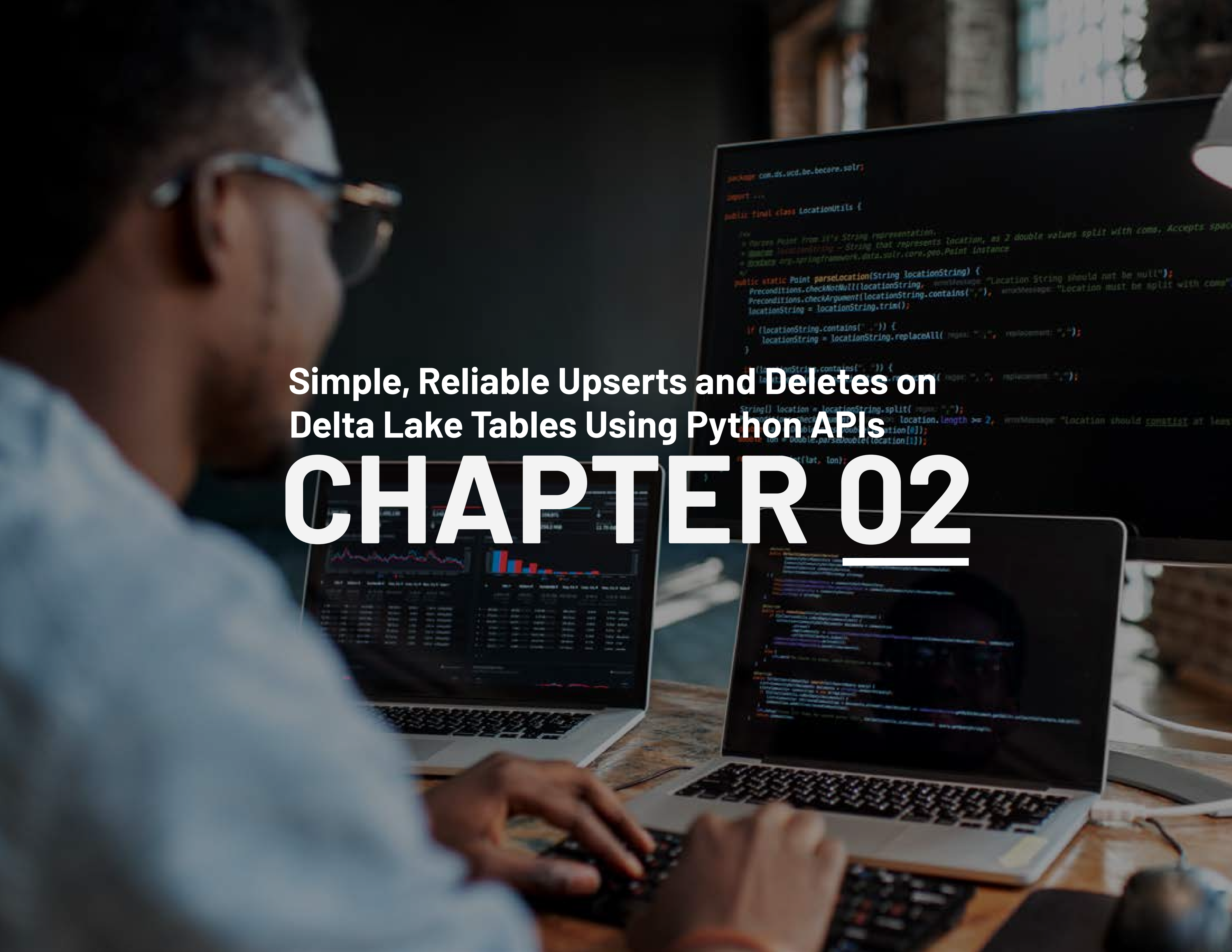
Tech Talk | Addressing GDPR and CCPA Scenarios With Delta Lake and Apache Spark

Tech Talk | Using Delta as a Change Data Capture Source

Simplifying Change Data Capture With Databricks Delta

Building Sessionization Pipeline at Scale With Databricks Delta

Tech Chat | Slowly Changing Dimensions (SCD) Type 2



Simple, Reliable Upserts and Deletes on Delta Lake Tables Using Python APIs

CHAPTER 02

02 Simple, Reliable Upserts and Deletes on Delta Lake Tables Using Python APIs

In this chapter, we will demonstrate how to use Python and the new Python APIs in Delta Lake within the context of an on-time flight performance scenario. We will show how to upsert and delete data, query old versions of data with time travel, and vacuum older versions for cleanup.

How to start using Delta Lake

The Delta Lake package is installable through PySpark by using the `--packages` option. In our example, we will also demonstrate the ability to VACUUM files and execute Delta Lake SQL commands within Apache Spark. As this is a short demonstration, we will also enable the following configurations:

```
spark.databricks.delta.retentionDurationCheck.enabled=false
```

to allow us to vacuum files shorter than the default retention duration of seven days. Note, this is only required for the SQL command VACUUM

```
spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension
```

to enable Delta Lake SQL commands within Apache Spark; this is not required for Python or Scala API calls.

```
# Using Spark Packages
./bin/pyspark --packages io.delta:delta-core_2.11:0.4.0 --conf “spark.
databricks.delta.retentionDurationCheck.enabled=false” --conf “spark.
sql.extensions=io.delta.sql.DeltaSparkSessionExtension”
```

Loading and saving our Delta Lake data

This scenario will be using the On-time flight performance or Departure Delays data set generated from the RITA BTS Flight Departure Statistics; some examples of this data in action include the 2014 Flight Departure Performance via d3.js Crossfilter and On-Time Flight Performance with GraphFrames for Apache Spark™. Within PySpark, start by reading the data set.

```
# Location variables
tripdelaysFilePath = "/root/data/departuredelays.csv"
pathToEventsTable = "/root/deltalake/departureDelays.delta"

# Read flight delay data
departureDelays = spark.read \
.option("header", "true") \
.option("inferSchema", "true") \
.csv(tripdelaysFilePath)
```

Next, let's save our departureDelays data set to a Delta Lake table. By saving this table to Delta Lake storage, we will be able to take advantage of its features including ACID transactions, unified batch and streaming and time travel.

```
# Save flight delay data into Delta Lake format
departureDelays \
.write \
.format("delta") \
.mode("overwrite") \
.save("departureDelays.delta")
```

Note, this approach is similar to how you would normally save Parquet data; instead of specifying `format("parquet")`, you will now specify `format("delta")`. If you were to take a look at the underlying file system, you will notice four files created for the departureDelays Delta Lake table.

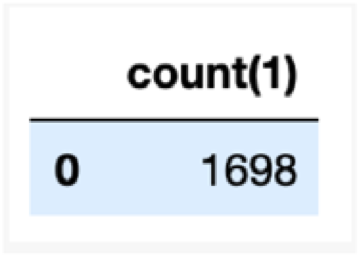
```
/departureDelays.delta$ ls -l
.
..
_delta_log
part-00000-df6f69ea-e6aa-424b-bc0e-f3674c4f1906-c000.snappy.parquet
part-00001-711bcce3-fe9e-466e-a22c-8256f8b54930-c000.snappy.parquet
part-00002-778ba97d-89b8-4942-a495-5f6238830b68-c000.snappy.parquet
Part-00003-1a791c4a-6f11-49a8-8837-8093a3220581-c000.snappy.parquet
```

Now, let's reload the data, but this time our DataFrame will be backed by Delta Lake.

```
# Load flight delay data in Delta Lake format
delays_delta = spark \
.read \
.format("delta") \
.load("departureDelays.delta")

# Create temporary view
delays_delta.createOrReplaceTempView("delays_delta")

# How many flights are between Seattle and San Francisco
spark.sql("select count(1) from delays_delta where origin = 'SEA' and
destination = 'SFO'").show()
```



	count(1)
0	1698

Finally, let's determine the number of flights originating from Seattle to San Francisco; in this data set, there are 1698 flights.

In-place conversion to Delta Lake

If you have existing Parquet tables, you have the ability to convert them to Delta Lake format in place, thus not needing to rewrite your table. To convert the table, you can run the following commands.

```
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")
```

Delete our flight data

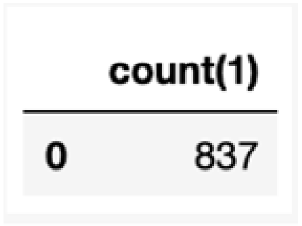
To delete data from a traditional data lake table, you will need to:

1. Select all of the data from your table not including the rows you want to delete
2. Create a new table based on the previous query
3. Delete the original table
4. Rename the new table to the original table name for downstream dependencies

Instead of performing all of these steps, with Delta Lake, we can simplify this process by running a DELETE statement. To show this, let's delete all of the flights that had arrived early or on-time (i.e., delay < 0).

```
from delta.tables import *
from pyspark.sql.functions import *
# Access the Delta Lake table
deltaTable = DeltaTable.forPath(spark, pathToEventsTable
)
# Delete all on-time and early flights
deltaTable.delete("delay < 0")

# How many flights are between Seattle and San Francisco
spark.sql("select count(1) from delays_delta where origin = 'SEA' and
destination = 'SFO'").show()
```



```
   count(1)
0       837
```

After we delete (more on this below) all of the on-time and early flights, as you can see from the preceding query there are 837 late flights originating from Seattle to San Francisco. If you review the file system, you will notice there are more files even though you deleted data.

```
/departureDelays.delta$ ls -l
_delta_log
part-00000-a2a19ba4-17e9-4931-9bbf-3c9d4997780b-c000.snappy.parquet
part-00000-df6f69ea-e6aa-424b-bc0e-f3674c4f1906-c000.snappy.parquet
part-00001-711bcce3-fe9e-466e-a22c-8256f8b54930-c000.snappy.parquet
part-00001-a0423a18-62eb-46b3-a82f-ca9aac1f1e93-c000.snappy.parquet
part-00002-778ba97d-89b8-4942-a495-5f6238830b68-c000.snappy.parquet
part-00002-bfaa0a2a-0a31-4abf-aa63-162402f802cc-c000.snappy.parquet
part-00003-1a791c4a-6f11-49a8-8837-8093a3220581-c000.snappy.parquet
part-00003-b0247e1d-f5ce-4b45-91cd-16413c784a66-c000.snappy.parquet
```

In traditional data lakes, deletes are performed by rewriting the entire table excluding the values to be deleted. With Delta Lake, deletes are instead performed by selectively writing new versions of the files containing the data to be deleted and only marks the previous files as deleted. This is because Delta Lake uses multiversion concurrency control (MVCC) to do atomic operations on the table: For example, while one user is deleting data, another user may be querying the previous version of the table. This multiversion model also enables us to travel back in time (i.e., time travel) and query previous versions as we will see later.

Update our flight data

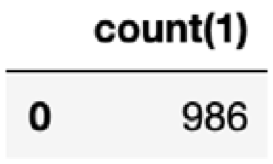
To update data from your traditional Data Lake table, you will need to:

1. Select all of the data from your table not including the rows you want to modify
2. Modify the rows that need to be updated/changed
3. Merge these two tables to create a new table
4. Delete the original table
5. Rename the new table to the original table name for downstream dependencies

Instead of performing all of these steps, with Delta Lake, we can simplify this process by running an UPDATE statement. To show this, let's update all of the flights originating from Detroit to Seattle.

```
# Update all flights originating from Detroit to now be
originating from Seattle
deltaTable.update(“origin = ‘DTW’”, { “origin”: “’SEA’” } )

# How many flights are between Seattle and San Francisco
spark.sql(“select count(1) from delays_delta where origin = ‘SEA’
and destination = ‘SFO’”).show()
```



	count(1)
0	986

With the Detroit flights now tagged as Seattle flights, we now have 986 flights originating from Seattle to San Francisco. If you were to list the file system for your departureDelays folder (i.e., `$../departureDelays/ls -l`), you will notice there are now 11 files (instead of the 8 right after deleting the files and the four files after creating the table).

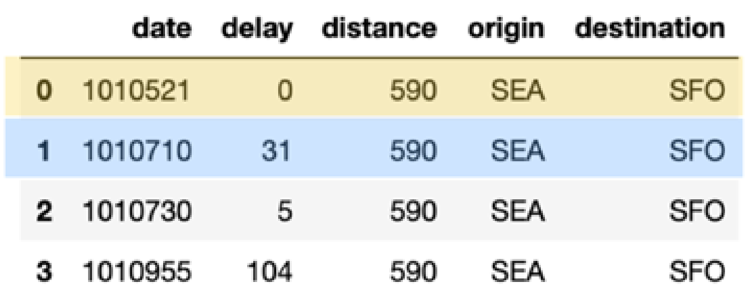
Merge our flight data

A common scenario when working with a data lake is to continuously append data to your table. This often results in duplicate data (rows you do not want to be inserted into your table again), new rows that need to be inserted, and some rows that need to be updated. With Delta Lake, all of this can be achieved by using the merge operation (similar to the SQL MERGE statement).

Let's start with a sample data set that you will want to be updated, inserted or de-duplicated with the following query.

```
# What flights between SEA and SFO for these date periods
spark.sql(“select * from delays_delta where origin = ‘SEA’ and
destination = ‘SFO’ and date like ‘1010%’ limit 10”).show()
```

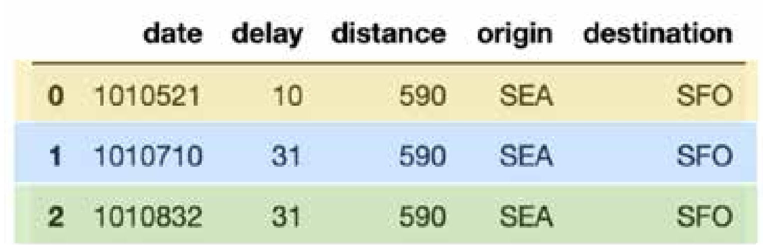
The output of this query looks like the following table. Note, the color-coding has been added to clearly identify which rows are de-duplicated (blue), updated (yellow) and inserted (green).



	date	delay	distance	origin	destination
0	1010521	0	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010730	5	590	SEA	SFO
3	1010955	104	590	SEA	SFO

Next, let's generate our own `merge_table` that contains data we will insert, update or de-duplicate with the following code snippet.

```
items = [(1010710, 31, 590, ‘SEA’, ‘SFO’), (1010521, 10, 590,
‘SEA’, ‘SFO’),
(1010822, 31, 590, ‘SEA’, ‘SFO’)]
cols = [‘date’, ‘delay’, ‘distance’, ‘origin’, ‘destination’]
merge_table = spark.createDataFrame(items, cols)
merge_table.toPandas()
```



	date	delay	distance	origin	destination
0	1010521	10	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010832	31	590	SEA	SFO

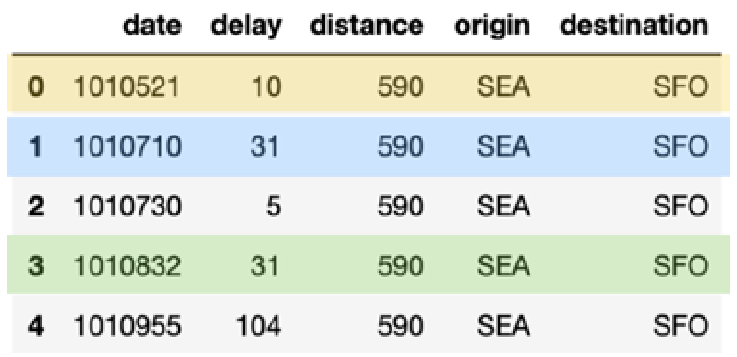
In the preceding table (merge_table), there are three rows with a unique date value:
1. 1010521: This row needs to *update* the *flights* table with a new delay value (yellow)
2. 1010710: This row is a *duplicate* (blue)
3. 1010832: This is a new row to be *inserted* (green)

With Delta Lake, this can be easily achieved via a merge statement as noted in the following code snippet.

```
# Merge merge_table with flights
deltaTable.alias(“flights”) \
    .merge(merge_table.alias(“updates”),”flights.date =
     updates.date”) \
    .whenMatchedUpdate(set = { “delay” : “updates.delay” } ) \
    .whenNotMatchedInsertAll() \
    .execute()

# What flights between SEA and SFO for these date periods
spark.sql(“select * from delays_delta where origin = ‘SEA’ and
destination = ‘SFO’ and date like ‘1010%’ limit 10”).show()
```

All three actions of de-duplication, update and insert were efficiently completed with one statement.



	date	delay	distance	origin	destination
0	1010521	10	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010730	5	590	SEA	SFO
3	1010832	31	590	SEA	SFO
4	1010955	104	590	SEA	SFO

View table history

As previously noted, after each of our transactions (delete, update), there were more files created within the file system. This is because for each transaction, there are different versions of the Delta Lake table.

This can be seen by using the DeltaTable.history() method as noted below.

```
deltaTable.history().show()
+-------+-------------------+------+--------+---------+--------------------+---
-+--------+---------+-----------+--------------+-------------+
|version|          timestamp|userId|userName|operation| operationParameters|
job|notebook|clusterId|readVersion|isolationLevel|isBlindAppend|
+-------+-------------------+------+--------+---------+--------------------+---
-+--------+---------+-----------+--------------+-------------+

|      2|2019-09-29 15:41:22|  null|    null|   UPDATE|[predicate ->
(or...|null|    null|     null|          1|          null|        false|
|      1|2019-09-29 15:40:45|  null|    null|   DELETE|[predicate ->
["(...|null|    null|     null|          0|          null|        false|
|      0|2019-09-29 15:40:14|  null|    null|    WRITE|[mode ->
Overwrit...|null|    null|     null|          null|          null|        false|
+-------+-------------------+------+--------+---------+--------------------+---
-+--------+---------+-----------+--------------+-------------+
```

Note, you can also perform the same task with SQL:

```
spark.sql("DESCRIBE HISTORY '" + pathToEventsTable + "'").show()
```

As you can see, there are three rows representing the different versions of the table (below is an abridged version to help make it easier to read) for each of the operations (create table, delete and update):

version	timestamp	operation	operationParameters
2	2019-09-29 15:41:22	UPDATE	[predicate -> (or…
1	2019-09-29 15:40:45	DELETE	[predicate -> ["(…
0	2019-09-29 15:40:14	WRITE	[mode -> Overwrit…

Travel back in time with table history

With Time Travel, you can review the Delta Lake table as of the version or timestamp. To view historical data, specify the version or timestamp option; in the following code snippet, we will specify the version option.

```
# Load DataFrames for each version
dfv0 = spark.read.format("delta").option("versionAsOf",
0).load("departureDelays.delta")
dfv1 = spark.read.format("delta").option("versionAsOf",
1).load("departureDelays.delta")
dfv2 = spark.read.format("delta").option("versionAsOf",
2).load("departureDelays.delta")

# Calculate the SEA to SFO flight counts for each version of history
cnt0 = dfv0.where("origin = 'SEA'").where("destination = 'SFO'").count()
cnt1 = dfv1.where("origin = 'SEA'").where("destination = 'SFO'").count()
cnt2 = dfv2.where("origin = 'SEA'").where("destination = 'SFO'").count()

# Print out the value
print("SEA -> SFO Counts: Create Table: %s, Delete: %s, Update: %s" %
(cnt0, cnt1, cnt2))

## Output
SEA -> SFO Counts: Create Table: 1698, Delete: 837, Update: 986
```

Whether for governance, risk management and compliance (GRC) or rolling back errors, the Delta Lake table contains both the metadata (e.g., recording the fact that a delete had occurred with these operators) and data (e.g., the actual rows deleted). But how do we remove the data files either for compliance or size reasons?

Clean up old table versions with vacuum

The Delta Lake vacuum method will delete all of the rows (and files) by default that are older than seven days' reference. If you were to view the file system, you'll notice the 11 files for your table.

```
/departureDelays.delta$ ls -l
_delta_log
```

```
part-00000-5e52736b-0e63-48f3-8d56-50f7cfa0494d-c000.snappy.parquet
part-00000-69eb53d5-34b4-408f-a7e4-86e000428c37-c000.snappy.parquet
part-00000-f8edaf04-712e-4ac4-8b42-368d0bbdb95b-c000.snappy.parquet
part-00001-20893eed-9d4f-4c1f-b619-3e6ea1fdd05f-c000.snappy.parquet
part-00001-9b68b9f6-bad3-434f-9498-f92dc4f503e3-c000.snappy.parquet
part-00001-d4823d2e-8f9d-42e3-918d-4060969e5844-c000.snappy.parquet
part-00002-24da7f4e-7e8d-40d1-b664-95bf93ffeadb-c000.snappy.parquet
part-00002-3027786c-20a9-4b19-868d-dc7586c275d4-c000.snappy.parquet
part-00002-f2609f27-3478-4bf9-aeb7-2c78a05e6ec1-c000.snappy.parquet
part-00003-850436a6-c4dd-4535-a1c0-5dc0f01d3d55-c000.snappy.parquet
Part-00003-b9292122-99a7-4223-aaa9-8646c281f199-c000.snappy.parquet
```

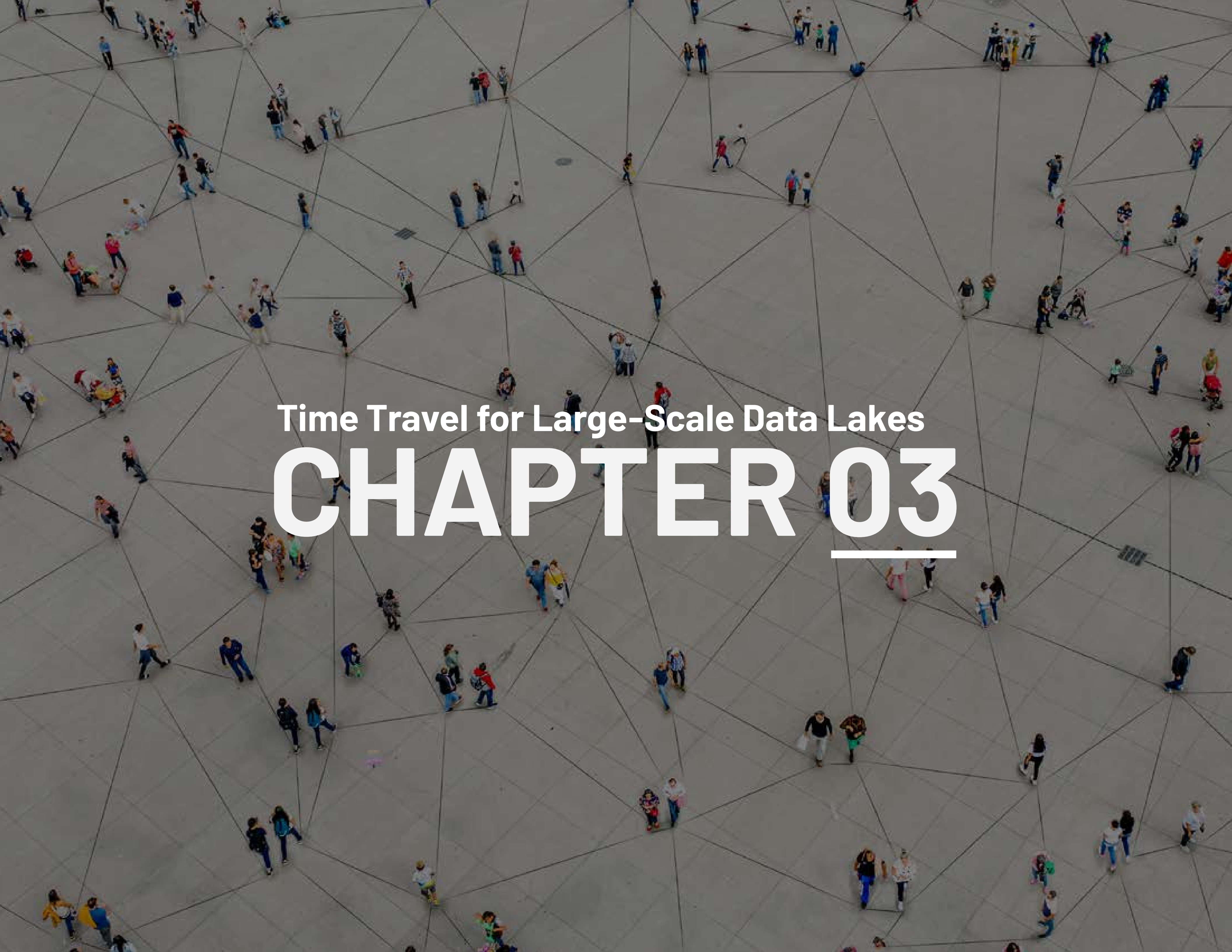
To delete all of the files so that you only keep the current snapshot of data, you will specify a small value for the vacuum method (instead of the default retention of 7 days).

```
# Remove all files older than 0 hours old.
deltaTable.vacuum(0)
Note, you perform the same task via SQL syntax:¸
# Remove all files older than 0 hours old
spark.sql(“VACUUM ‘” + pathToEventsTable + “‘ RETAIN 0 HOURS”)
```

Once the vacuum has completed, when you review the file system you will notice fewer files as the historical data has been removed.

```
/departureDelays.delta$ ls -l
_delta_log
part-00000-f8edaf04-712e-4ac4-8b42-368d0bbdb95b-c000.snappy.parquet
part-00001-9b68b9f6-bad3-434f-9498-f92dc4f503e3-c000.snappy.parquet
part-00002-24da7f4e-7e8d-40d1-b664-95bf93ffeadb-c000.snappy.parquet
part-00003-b9292122-99a7-4223-aaa9-8646c281f199-c000.snappy.parquet
```

Note, the ability to time travel back to a version older than the retention period is lost after running vacuum.



Time Travel for Large-Scale Data Lakes

CHAPTER 03

03 Time Travel for Large-Scale Data Lakes

Time travel capabilities are available in Delta Lake. Delta Lake is an open-source storage layer that brings reliability to data lakes. Delta Lake provides ACID transactions, scalable metadata handling, and unifies streaming and batch data processing. Delta Lake runs on top of your existing data lake and is fully compatible with Apache Spark APIs.

With this feature, Delta Lake 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.

Your organization can finally standardize on a clean, centralized, versioned big data repository in your own cloud storage for your analytics.

Common challenges with changing data

- **Audit data changes:** Auditing data changes is critical 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 and 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. A 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 Lake's time travel capabilities simplify building data pipelines for the above use cases. Time Travel in Delta Lake improves developer productivity tremendously. It helps:

- Data scientists manage their experiments better
- Data engineers simplify their pipelines and roll back bad writes
- Data analysts do easy reporting

Organizations can finally standardize on a clean, centralized, versioned big data repository in their own cloud storage for analytics. We are thrilled to see what you will be able to accomplish with this feature.

As you write into a Delta Lake table or directory, every operation is automatically versioned. You can access the different versions of the data two different ways:

1. Using a timestamp

Scala syntax

You can provide the timestamp or date string as an option to DataFrame reader:

```
val df = spark.read
  .format("delta")
  .option("timestampAsOf", "2019-01-01")
  .load("/path/to/my/table")
```

Python syntax

```
df = spark.read \
  .format("delta") \
  .option("timestampAsOf", "2019-01-01") \
  .load("/path/to/my/table")
```

SQL syntax

```
SELECT count(*) FROM my_table TIMESTAMP AS OF "2019-01-01"
SELECT count(*) FROM my_table TIMESTAMP AS OF date_sub(current_date(), 1)
SELECT count(*) FROM my_table TIMESTAMP AS OF "2019-01-01 01:30:00.000"
```

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:

```
val inputPath = "/path/to/my/table@20190101000000000"
val df = loadData(inputPath)
// Function in a library that you don't have access to
def loadData(inputPath : String) : DataFrame = {
  spark.read
    .format("delta")
    .load(inputPath)
}
inputPath = "/path/to/my/table@20190101000000000"
df = loadData(inputPath)

# Function in a library that you don't have access to
def loadData(inputPath):
  return spark.read \
    .format("delta") \
    .load(inputPath)
}
```

databricks

2. Using a version number

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

Scala syntax

```
val df = spark.read
  .format(“delta”)
  .option(“versionAsOf”, “5238”)
  .load(“/path/to/my/table”)

val df = spark.read
  .format(“delta”)
  .load(“/path/to/my/table@v5238”)
```

Python syntax

```
df = spark.read \
  .format(“delta”) \
  .option(“versionAsOf”, “5238”) \
  .load(“/path/to/my/table”)

df = spark.read \
  .format(“delta”) \
  .load(“/path/to/my/table@v5238”)
```

SQL syntax

```
SELECT count(*) FROM my_table VERSION AS OF 5238
```

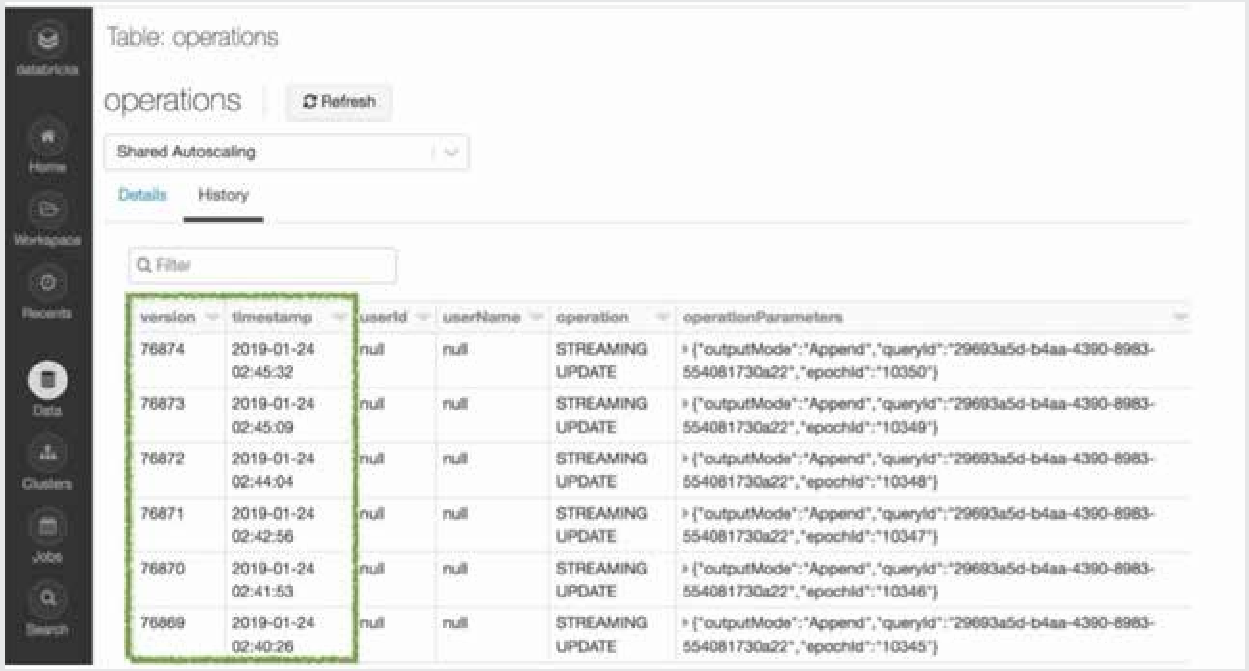


Table: operations

operations

Shared Autoscaling

version	timestamp	userId	userName	operation	operationParameters
76874	2019-01-24 02:45:32	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10350"}
76873	2019-01-24 02:45:09	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10349"}
76872	2019-01-24 02:44:04	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10348"}
76871	2019-01-24 02:42:56	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10347"}
76870	2019-01-24 02:41:53	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10346"}
76869	2019-01-24 02:40:26	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10345"}

Audit data changes

You can look at the history of table changes using the DESCRIBE HISTORY command or through the UI.

Reproduce experiments and 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 hundreds of models before they put one into production, and in that time-consuming process would like to go back to earlier models. However, because data management is often separate from data science tools, this is really hard to accomplish.

Databricks solves this reproducibility problem by integrating Delta Lake's Time Travel capabilities with MLflow, an open-source platform for the machine learning lifecycle. For reproducible machine learning training, you can simply log a timestamped URL to the path as an MLflow parameter to track which version of the data was used for each training job.

This enables you to go back to earlier settings and data sets to reproduce earlier models. You neither need to coordinate with upstream teams on the data nor worry about cloning data for different experiments. This is the power of unified analytics, whereby data science is closely married with data engineering.

Rollbacks

Time travel also makes it easy to do rollbacks in case of bad writes. For example, if your GDPR pipeline job had a bug that accidentally deleted user information, you can easily fix the pipeline:

```
INSERT INTO my_table
SELECT * FROM my_table TIMESTAMP AS OF date_sub(current_date(), 1)
```

```
WHERE userId = 111
You can also fix incorrect updates as follows:
MERGE INTO my_table target
USING my_table TIMESTAMP AS OF date_sub(current_date(), 1) source
ON source.userId = target.userId
WHEN MATCHED THEN UPDATE SET *
```

If you simply want to roll back to a previous version of your table, you can do so with either of the following commands:

```
RESTORE TABLE my_table VERSION AS OF [version_number]
RESTORE TABLE my_table TIMESTAMP AS OF [timestamp]
```

Pinned view of a continuously updating Delta Lake table across multiple downstream jobs

With AS OF queries, you can now pin the snapshot of a continuously updating Delta Lake table for multiple downstream jobs. Consider a situation where a Delta Lake table is being continuously updated, say every 15 seconds, and there is a downstream job that periodically reads from this Delta Lake table and updates different destinations. In such scenarios, typically you want a consistent view of the source Delta Lake table so that all destination tables reflect the same state.

You can now easily handle such scenarios as follows:

```
version = spark.sql("SELECT max(version) FROM (DESCRIBE HISTORY
my_table)").collect()

# Will use the latest version of the table for all operations below

data = spark.table("my_table@v%s" % version[0][0]data.where
("event_type = e1").write.jdbc("table1")
```

```
data.where("event_type = e2").write.jdbc("table2")
...
data.where("event_type = e10").write.jdbc("table10")
```

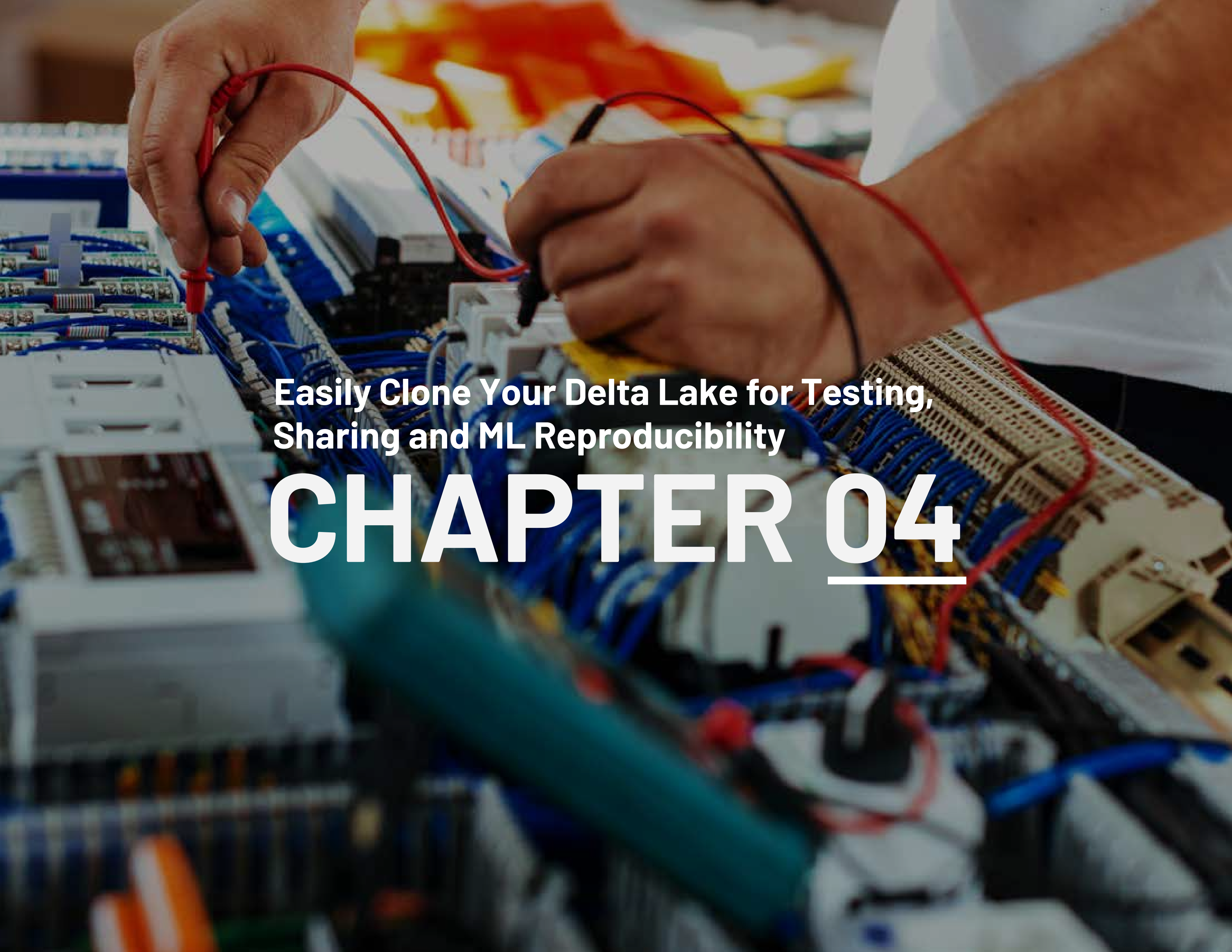
Queries for time series analytics made simple

Time travel also simplifies time series analytics. For example, if you want to find out how many new customers you added over the last week, your query could be a very simple one like this:

```
SELECT count(distinct userId) - (
SELECT count(distinct userId)
FROM my_table TIMESTAMP AS OF date_sub(current_date(), 7))
FROM my_table
```

Additional resources

Tech Talk | Diving Into Delta Lake: Unpacking the Transaction Log
Tech Talk | Getting Data Ready for Data Science with Delta Lake and MLflow
Data + AI Summit Europe 2020 | Data Time Travel by Delta Time Machine
Spark + AI Summit NA 2020 | Machine Learning Data Lineage With MLflow and Delta Lake
Productionizing Machine Learning With Delta Lake



Easily Clone Your Delta Lake for Testing, Sharing and ML Reproducibility

CHAPTER 04

04 Easily Clone Your Delta Lake for Testing, Sharing and ML Reproducibility

Delta Lake has a feature called **Table Cloning**, which makes it easy to test, share and recreate tables for ML reproducibility. Creating copies of tables in a data lake or data warehouse has several practical uses. However, given the volume of data in tables in a data lake and the rate of its growth, making physical copies of tables is an expensive operation.

Delta Lake now makes the process simpler and cost-effective with the help of table clones.

What are clones?

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. In Delta Lake we have two types of clones: shallow or deep.

Shallow clones

A shallow (also known as a 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 the 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?

Sometimes I wish I had a clone to help with my chores or magic tricks. However, we're not talking about human clones here. There are many scenarios where you need a copy of your data sets — for exploring, sharing or testing ML models or analytical queries. Below are some examples of customer 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 data sets 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
CREATE TABLE delta.`/some/test/location` SHALLOW CLONE prod.events

# Python
DeltaTable.forName("spark", "prod.events").clone("/some/test/location",
isShallow=True)

// Scala
DeltaTable.forName("spark", "prod.events").clone("/some/test/location",
isShallow=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 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 that you're making until you're done with all of your work. A shallow clone can help you out here:

```
-- SQL
CREATE TABLE temp.staged_changes SHALLOW CLONE prod.events;
DELETE FROM temp.staged_changes WHERE event_id is null;
UPDATE temp.staged_changes SET change_date = current_date()
WHERE change_date is null;
...
-- Perform your verifications
```

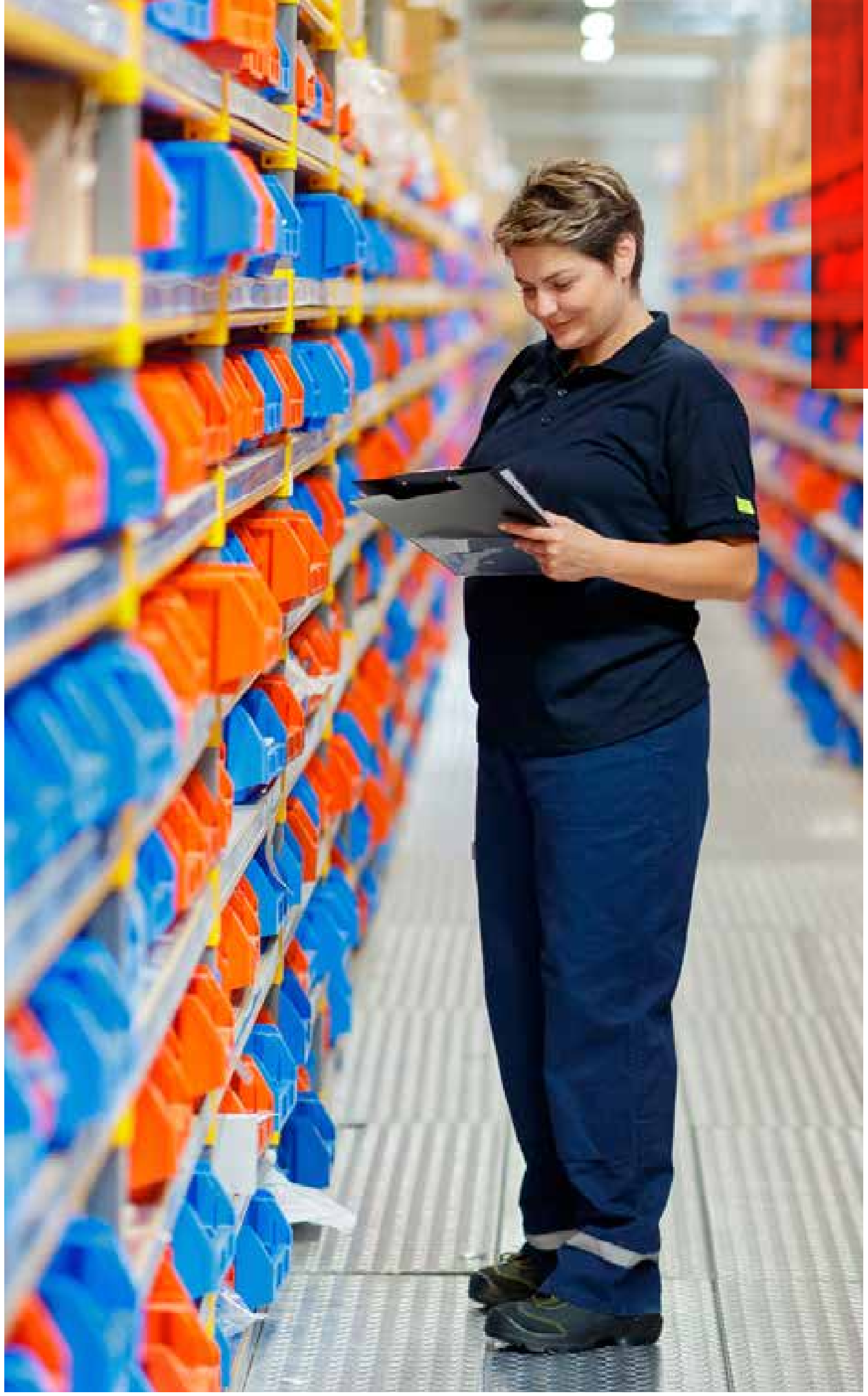
Once you're happy with the results, you have two options. If no other change has been made to your source table, you can replace your source table with the clone. If changes have been made to your source table, you can merge the changes into your source table.

```
-- 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

Coming up with an effective ML model is an iterative process. Throughout this process of tweaking the different parts of the model, data scientists need to assess the accuracy of the model against a fixed data set.

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 autologging 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, “feature_
store_bf2020”)
```

Data migration

A massive table may need to be moved to a new, dedicated bucket or storage system for performance 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
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

In an organization, it is often the case that users from different departments are looking for data sets that they can use to enrich their analysis or models. You may want to share your data with other users across the organization. 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 again come to the rescue.

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

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

Looks awesome! Any gotchas?

Just to reiterate some of the gotchas mentioned above as a single list, here's what you should be wary of:

- Clones are executed on a snapshot of your data. Any changes that are made to the source table after the cloning process starts will not be reflected in the clone.
- Shallow clones are not self-contained tables like deep clones. If the data is deleted in the source table (for example through VACUUM), your shallow clone may not be usable.
- Clones have a separate, independent history from the source table. Time travel queries on your source table and clone may not return the same result.
- Shallow clones do not copy stream transactions or COPY INTO metadata. Use deep clones to migrate your tables and continue your ETL processes from where it left off.

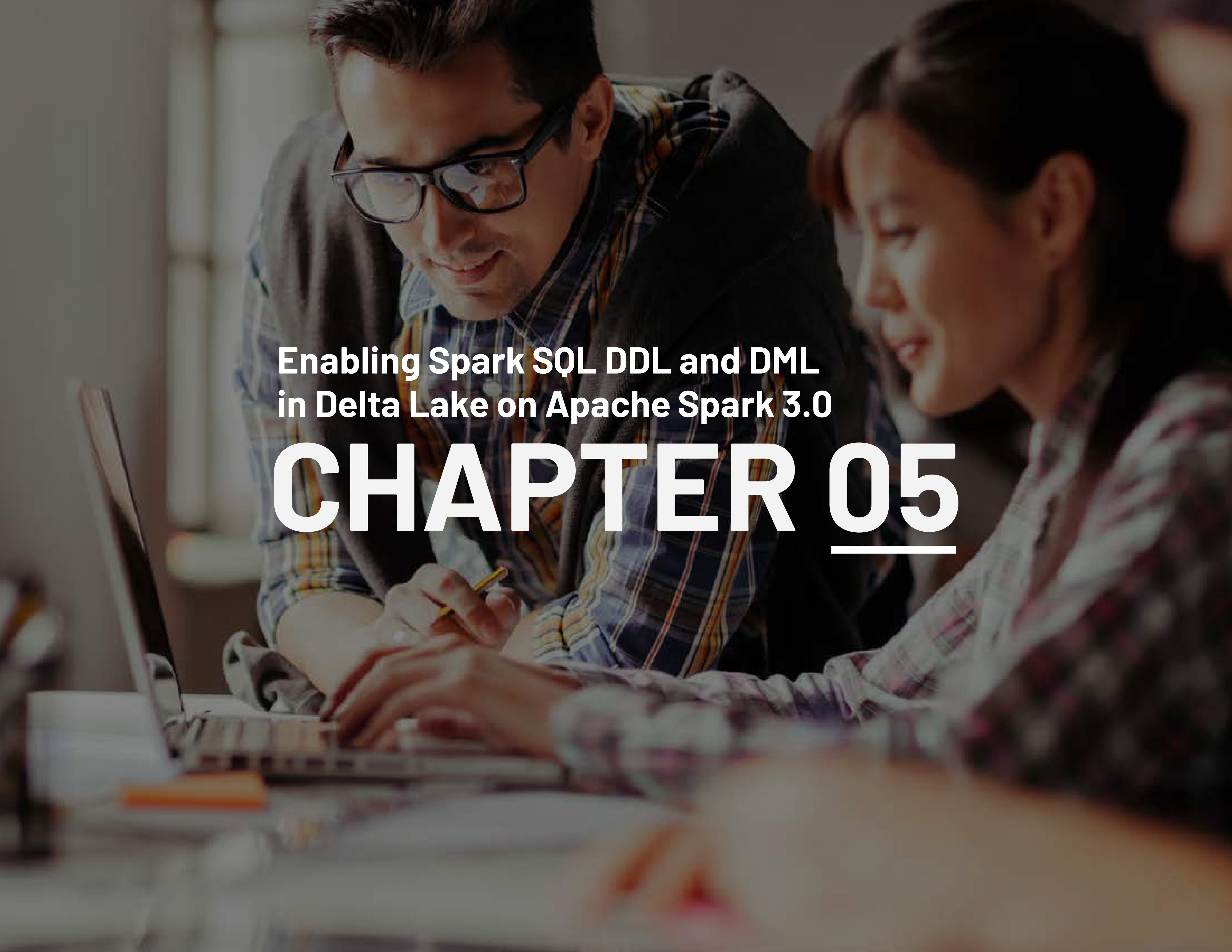
How can I use it?

Shallow and deep clones support new advances in how data teams test and manage their modern cloud data lakes and warehouses. Table clones can help your team implement production-level testing of their pipelines, fine-tune their indexing for optimal query performance, create table copies for sharing — all with minimal overhead and expense. If this is a need in your organization, we hope you will take table cloning for a spin and give us your feedback — we look forward to hearing about new use cases and extensions you would like to see in the future.

Additional resources

Using Deep Clone for Disaster Recovery With Delta Lake on Databricks

Simplifying Disaster Recovery With Delta Lake



Enabling Spark SQL DDL and DML in Delta Lake on Apache Spark 3.0

CHAPTER 05

05 Enabling Spark SQL DDL and DML in Delta Lake on Apache Spark 3.0

The release of Delta Lake 0.7.0 coincided with the release of Apache Spark 3.0, thus enabling a new set of features that were simplified using Delta Lake from SQL. Here are some of the key features.

Support for SQL DDL commands to define tables in the Hive metastore

You can now define Delta tables in the Hive metastore and use the table name in all SQL operations when creating (or replacing) tables.

Create or replace tables

```
-- Create table in the metastore
CREATE TABLE events (
    date DATE,
    eventId STRING,
    eventType STRING,
    data STRING)
USING DELTA
PARTITIONED BY (date)
LOCATION '/delta/events'
-- If a table with the same name already exists, the table is replaced
with
the new configuration, else it is created
CREATE OR REPLACE TABLE events (
      date DATE,
    eventId STRING,
      eventType STRING,
      data STRING)
  USING DELTA
  PARTITIONED BY (date)
  LOCATION '/delta/events'
```

Explicitly alter the table schema

```
-- Alter table and schema
ALTER TABLE table_name ADD COLUMNS (
    col_name data_type
        [COMMENT col_comment]
        [FIRST|AFTER colA_name],
    ...)
```

You can also use the Scala/Java/Python APIs:

- `DataFrame.saveAsTable(tableName)` and `DataFrameWriterV2` APIs (#307).
- `DeltaTable.forName(tableName)` API to create instances of `io.delta.tables .DeltaTable` which is useful for executing Update/Delete/Merge operations in Scala/Java/Python.

Support for SQL Insert, Delete, Update and Merge

One of the most frequent questions through our Delta Lake Tech Talks was when would DML operations such as delete, update and merge be available in Spark SQL? Wait no more, these operations are now available in SQL! Below are examples of how you can write delete, update and merge (insert, update, delete and de-duplication operations using Spark SQL).

```
-- Using append mode, you can atomically add new data to an existing
Delta table
INSERT INTO events SELECT * FROM newEvents
-- To atomically replace all of the data in a table, you can use
overwrite mode
INSERT OVERWRITE events SELECT * FROM newEvents

-- Delete events
DELETE FROM events WHERE date < '2017-01-01'

-- Update events
UPDATE events SET eventType = 'click' WHERE eventType = 'click'

-- Upsert data to a target Delta
-- table using merge
MERGE INTO events
USING updates
   ON events.eventId = updates.eventId
 WHEN MATCHED THEN UPDATE
      SET events.data = updates.data
 WHEN NOT MATCHED THEN INSERT (date, eventId, data)
      VALUES (date, eventId, data)
```

It is worth noting that the merge operation in Delta Lake supports more advanced syntax than standard ANSI SQL syntax. For example, merge supports

- Delete actions — Delete a target when matched with a source row. For example, "... WHEN MATCHED THEN DELETE ..."
- Multiple matched actions with clause conditions — Greater flexibility when target and source rows match. For example:

```
...
WHEN MATCHED AND events.shouldDelete THEN DELETE
WHEN MATCHED THEN UPDATE SET events.data = updates.data
```

- Star syntax — Shorthand for setting target column value with the similarly-named sources column. For example:

```
WHEN MATCHED THEN SET *
WHEN NOT MATCHED THEN INSERT *
-- equivalent to updating/inserting with event.date = updates.date,
    events.eventId = updates.eventId, event.data = updates.data
```

Automatic and incremental Presto/Athena manifest generation

As noted in Query Delta Lake Tables From Presto and Athena, Improved Operations Concurrency, and Merge Performance, Delta Lake supports other processing engines to read Delta Lake by using manifest files; the manifest files contain the list of the most current version of files as of manifest generation. As described in the preceding chapter, you will need to:

- Generate a Delta Lake manifest file
- Configure Presto or Athena to read the generated manifests
- Manually re-generate (update) the manifest file

New for Delta Lake 0.7.0 is the capability to update the manifest file automatically with the following command:

```
ALTER TABLE delta.`pathToDeltaTable`
SET TBLPROPERTIES(
    delta.compatibility.symlinkFormatManifest.enabled=true
)
```

Configuring your table through table properties

With the ability to set table properties on your table by using ALTER TABLE SET TBLPROPERTIES, you can enable, disable or configure many features of Delta Lake such as automated manifest generation. For example, with table properties, you can block deletes and updates in a Delta table using `delta.appendOnly=true`.

You can also easily control the history of your Delta Lake table retention by the following properties:

`delta.logRetentionDuration`: Controls how long the history for a table (i.e., transaction log history) is kept. By default, 30 days of history is kept, but you may want to alter this value based on your requirements (e.g., GDPR historical context)

`delta.deletedFileRetentionDuration`: Controls how long ago a file must have been deleted before being a candidate for VACUUM. By default, data files older than seven days are deleted.

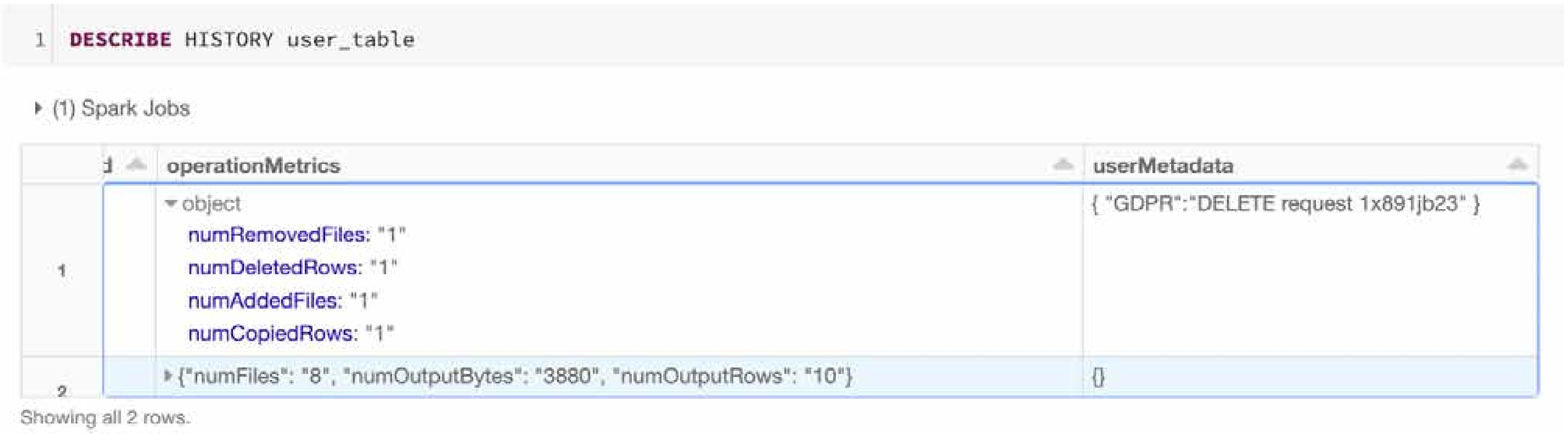
As of Delta Lake 0.7.0, you can use ALTER TABLE SET TBLPROPERTIES to configure these properties.

```
ALTER TABLE delta.`pathToDeltaTable`
SET TBLPROPERTIES(
    delta.logRetentionDuration = “interval “
    delta.deletedFileRetentionDuration = “interval “
)
```

Support for adding user-defined metadata in Delta Lake table commits

You can specify user-defined strings as metadata in commits made by Delta Lake table operations, either using the DataFrameWriter option userMetadata or the SparkSession configuration `spark.databricks.delta.commitInfo.userMetadata`.

In the following example, we are deleting a user (1xsdf1) from our data lake per user request. To ensure we associate the user's request with the deletion, we have also added the DELETE request ID into the userMetadata.



```
DESCRIBE HISTORY user_table
```

▸ (1) Spark Jobs

	operationMetrics	userMetadata
1	▾ object numRemovedFiles: "1" numDeletedRows: "1" numAddedFiles: "1" numCopiedRows: "1"	{ "GDPR":"DELETE request 1x891jb23" }
2	▸ {"numFiles": "8", "numOutputBytes": "3880", "numOutputRows": "10"}	{}

Showing all 2 rows.

```
SET spark.databricks.delta.commitInfo.userMetadata={
    “GDPR”:”DELETE Request 1x891jb23”
};
DELETE FROM user_table WHERE user_id = ‘1xsdf1’
```

When reviewing the history operations of the user table (user_table), you can easily identify the associated deletion request within the transaction log.

Other highlights

Other highlights for the Delta Lake 0.7.0 release include:

- Support for Azure Data Lake Storage Gen2 — Spark 3.0 has support for Hadoop 3.2 libraries which enables support for Azure Data Lake Storage Gen2.
- Improved support for streaming one-time triggers — With Spark 3.0, we now ensure that a one-time trigger (`Trigger.Once`) processes all outstanding data in a Delta Lake table in a single micro-batch even if rate limits are set with the DataStreamReader option maxFilesPerTrigger.

There were a lot of great questions during the AMA concerning structured streaming and using `trigger.once`.

For more information, some good resources explaining this concept include:

- Running Streaming Jobs Once a Day for 10x Cost Savings
- Beyond Lambda: Introducing Delta Architecture: Specifically the cost vs. latency trade-off discussed here.

Additional resources

Tech Talk | Delta Lake 0.7.0 + Spark 3.0 AMA

Tech Talks | Apache Spark 3.0 + Delta Lake

Enabling Spark SQL DDL and DML in Delta Lake on Apache Spark 3.0

What's next?

Now that you understand Delta Lake and how its features can improve performance, it may be time to take a look at some additional resources.

Explore subsequent eBooks in the collection >

- The Delta Lake Series — Fundamentals and Performance
- The Delta Lake Series — Streaming
- The Delta Lake Series — Lakehouse
- The Delta Lake Series — Customer Use Cases

Do a deep dive into Delta Lake >

Getting Started With Delta Lake Tech Talk Series

Diving Into Delta Lake Tech Talk Series

Try Databricks for free >

Learn more >

databricks