The Delta Lake Series Complete Collection

databricks

What is Delta Lake?

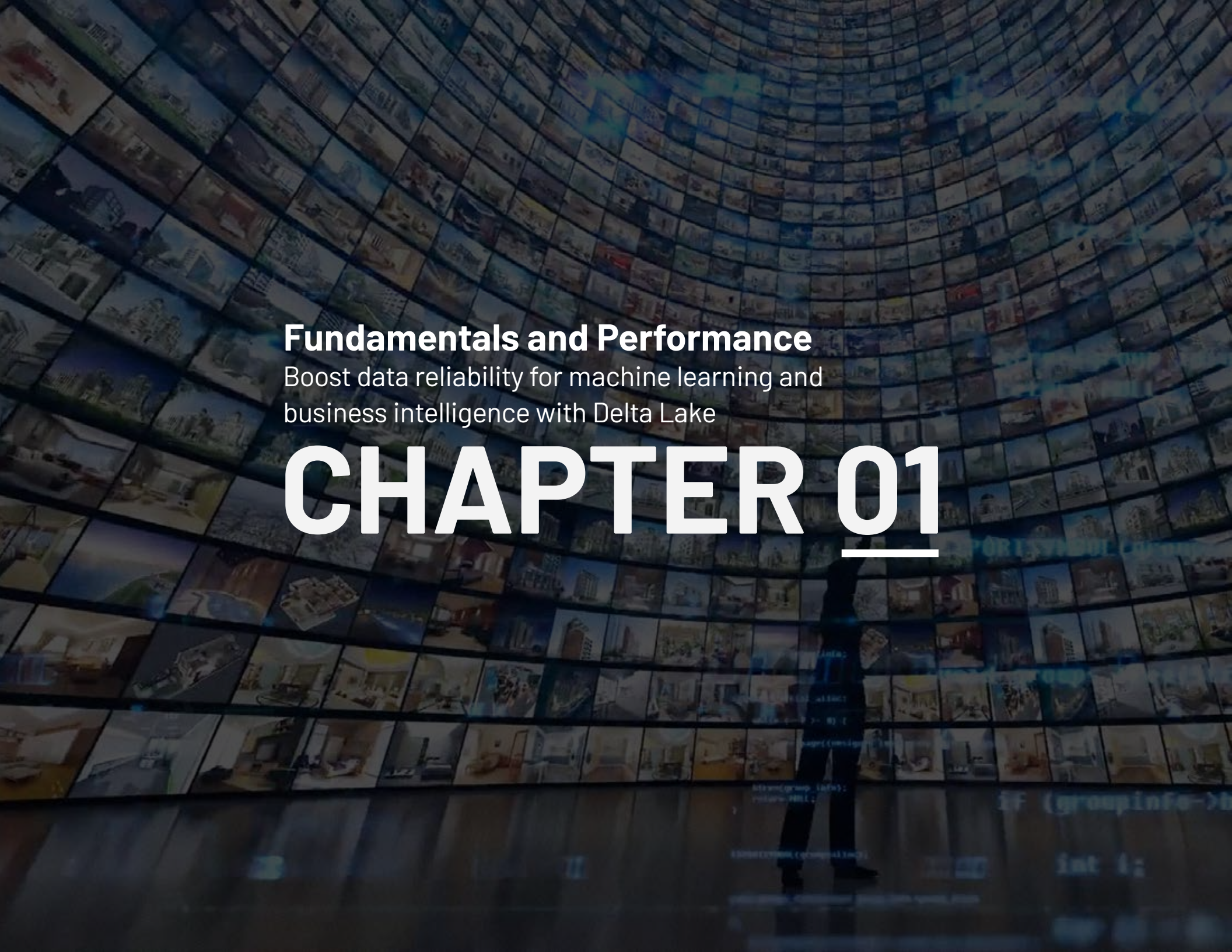
Delta Lake is a unified data management system that brings data reliability and fast analytics to cloud data lakes. Delta Lake runs on top of existing data lakes and is fully compatible with Apache Spark™ APIs.

At Databricks, we've seen how Delta Lake can bring reliability, performance and lifecycle management to data lakes. With Delta Lake, there will be no more 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.

In this eBook, the Databricks team has compiled all of their insights into a comprehensive format so that you can gain a full understanding of Delta Lake and its capabilities.

Here's what you'll find inside



Fundamentals and Performance

Boost data reliability for machine learning and business intelligence with Delta Lake

CHAPTER 01

The Fundamentals of Delta Lake: Why Reliability and Performance Matter

When it comes to data reliability, performance — the speed at which your programs run — is of utmost importance. Because of the ACID transactional protections that Delta Lake provides, you're able to get the reliability and performance you need.

With Delta Lake, you can stream and batch concurrently, perform CRUD operations, and save money because you're now using fewer VMs. It's easier to maintain your data engineering pipelines by taking advantage of streaming, even for batch jobs.

Delta Lake is a storage layer that brings reliability to your data lakes built on HDFS and cloud object storage by providing ACID transactions through optimistic concurrency control between writes and snapshot isolation for consistent reads during writes. Delta Lake also provides built-in data versioning for easy rollbacks and reproducing reports.

In this chapter, we'll share some of the common challenges with data lakes as well as the Delta Lake features that address them.

Challenges with data lakes

Data lakes are a common element within modern data architectures. They serve as a central ingestion point for the plethora of data that organizations seek to gather and mine. While a good step forward in getting to grips with the range of data, they run into the following common problems:

1. **Reading and writing into data lakes is not reliable.** Data engineers often run into the problem of unsafe writes into data lakes that cause readers to see garbage data during writes. They have to build workarounds to ensure readers always see consistent data during writes.

2. **The data quality in data lakes is low.** Dumping unstructured data into a data lake is easy, but this comes at the cost of data quality. Without any mechanisms for validating schema and the data, data lakes suffer from poor data quality. As a consequence, analytics projects that strive to mine this data also fail.

3. **Poor performance with increasing amounts of data.** As the amount of data that gets dumped into a data lake increases, the number of files and directories also increases. Big data jobs and query engines that process the data spend a significant amount of time handling the metadata operations. This problem is more pronounced in the case of streaming jobs or handling many concurrent batch jobs.

4. **Modifying, updating or deleting records in data lakes is hard.** Engineers need to build complicated pipelines to read entire partitions or tables, modify the data and write them back. Such pipelines are inefficient and hard to maintain.

Because of these challenges, many big data projects fail to deliver on their vision or sometimes just fail altogether. We need a solution that enables data practitioners to make use of their existing data lakes, while ensuring data quality.

Delta Lake's key functionalities

Delta Lake addresses the above problems to simplify how you build your data lakes. Delta Lake offers the following key functionalities:

- **ACID transactions:** Delta Lake provides ACID transactions between multiple writes. Every write is a transaction, and there is a serial order for writes recorded in a transaction log. The transaction log tracks writes at file level and uses optimistic

concurrency control, which is ideally suited for data lakes since multiple writes trying to modify the same files don't happen that often. In scenarios where there is a conflict, Delta Lake throws a concurrent modification exception for users to handle them and retry their jobs. Delta Lake also offers the highest level of isolation possible (serializable isolation) that allows engineers to continuously keep writing to a directory or table and consumers to keep reading from the same directory or table. Readers will see the latest snapshot that existed at the time the reading started.

- **Schema management:** Delta Lake automatically validates that the schema of the DataFrame being written is compatible with the schema of the table. Columns that are present in the table but not in the DataFrame are set to null. If there are extra columns in the DataFrame that are not present in the table, this operation throws an exception. Delta Lake has DDL to add new columns explicitly and the ability to update the schema automatically.

- **Scalable metadata handling:** Delta Lake stores the metadata information of a table or directory in the transaction log instead of the metastore. This allows Delta Lake to list files in large directories in constant time and be efficient while reading data.

- **Data versioning and time travel:** Delta Lake allows users to read a previous snapshot of the table or directory. When files are modified during writes, Delta Lake creates newer versions of the files and preserves the older versions. When users want to read the older versions of the table or directory, they can provide a timestamp or a version number to Apache Spark's read APIs, and Delta Lake constructs the full snapshot as of that timestamp or version based on the information in the transaction log. This allows users to reproduce experiments and reports and also revert a table to its older versions, if needed.

- **Unified batch and streaming sink:** Apart from batch writes, Delta Lake can also be used as an efficient streaming sink with Apache Spark's structured streaming. Combined with ACID transactions and scalable metadata handling, the efficient streaming sink enables lots of near real-time analytics use cases without having to maintain a complicated streaming and batch pipeline.

- **Record update and deletion:** Delta Lake will support merge, update and delete DML commands. This allows engineers to easily upsert and delete records in data lakes and simplify their change data capture and GDPR use cases. Since Delta Lake tracks and modifies data at file-level granularity, it is much more efficient than reading and overwriting entire partitions or tables.

- **Data expectations (coming soon):** Delta Lake will also support a new API to set data expectations on tables or directories. Engineers will be able to specify a boolean condition and tune the severity to handle data expectations. When Apache Spark jobs write to the table or directory, Delta Lake will automatically validate the records and when there is a violation, it will handle the records based on the severity provided.

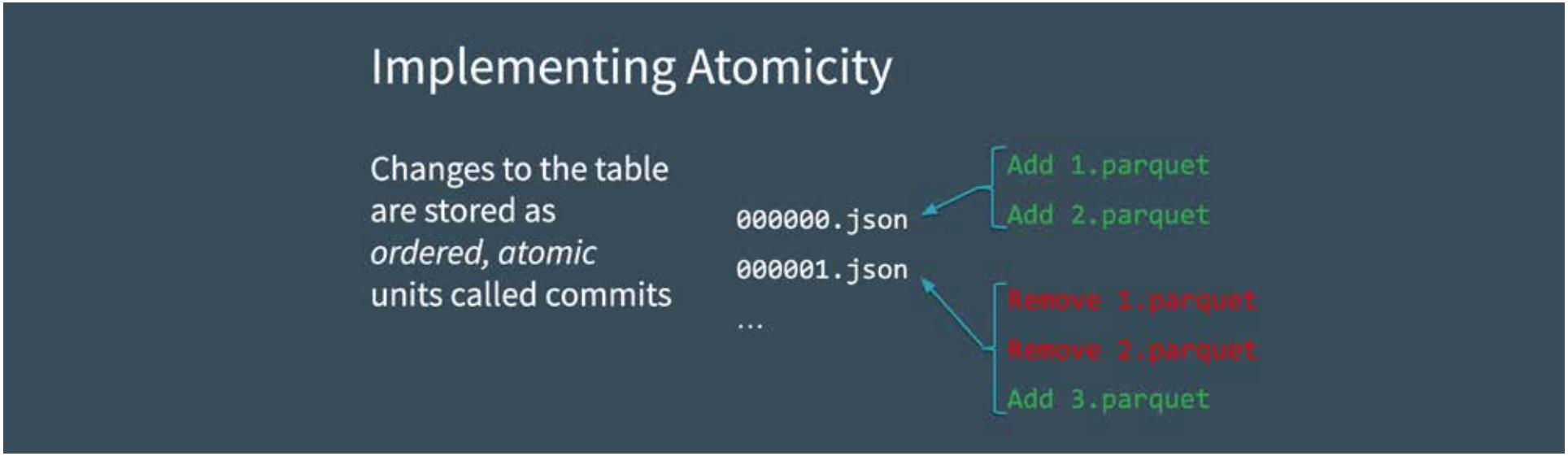
Unpacking the Transaction Log

The transaction log is key to understanding Delta Lake because it is the common thread that runs through many of its most important features, including ACID transactions, scalable metadata handling, time travel and more. The Delta Lake transaction log is an ordered record of every transaction that has ever been performed on a Delta Lake table since its inception.

Delta Lake is built on top of Apache Spark to allow multiple readers and writers of a given table to work on the table at the same time. To show users correct views of the data at all times, the transaction log serves as a single source of truth: the central repository that tracks all changes that users make to the table.

When a user reads a Delta Lake table for the first time or runs a new query on an open table that has been modified since the last time it was read, Spark checks the transaction log to see what new transactions are posted to the table. Then, Spark updates the end user's table with those new changes. This ensures that a user's version of a table is always synchronized with the master record as of the most recent query and that users cannot make divergent, conflicting changes to a table.

In this chapter, we'll explore how the Delta Lake transaction log offers an elegant solution to the problem of multiple concurrent reads and writes.



Implementing atomicity to ensure operations complete fully

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

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

Dealing with multiple concurrent reads and writes

But how does Delta Lake deal with multiple concurrent reads and writes? Since Delta Lake is powered by Apache Spark, it's not only possible for multiple users to modify a table at once — it's expected. To handle these situations, Delta Lake employs **optimistic concurrency control**.

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

Of course, even with optimistic concurrency control, sometimes users do try to modify the same parts of the data at the same time. Luckily, Delta Lake has a protocol for that. Delta Lake handles these cases by implementing a rule of mutual exclusion, then it attempts to solve any conflict optimistically.

This protocol allows Delta Lake to deliver on the ACID principle of isolation, which ensures that the resulting state of the table after multiple, concurrent writes is the same as if those writes had occurred serially, in isolation from one another.

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

Time travel, data lineage and debugging

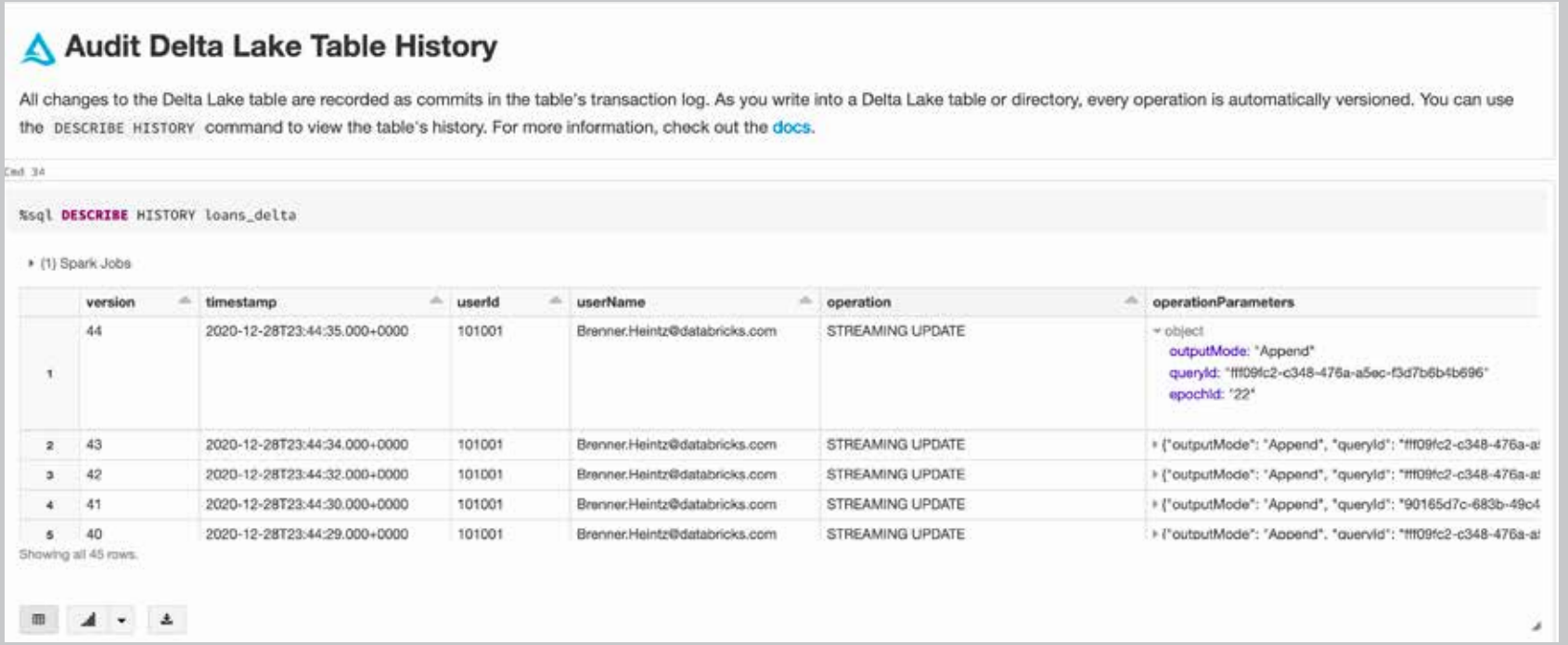
Every table is the result of the sum total of all the commits recorded in the Delta Lake transaction log — no more and no less. The transaction log provides a step-by-step instruction guide, detailing exactly how to get from the table's original state to its current state.

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

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

Want to learn more about Delta Lake's transaction log?

Read our blog post > Watch our tech talk >



How to Use Schema Enforcement and Evolution

As business problems and requirements evolve over time, so does the structure of your data. With Delta Lake, incorporating new columns or objects is easy; users have access to simple semantics to control the schema of their tables. At the same time, it is important to call out the importance of schema enforcement to prevent users from accidentally polluting their tables with mistakes or garbage data in addition to schema evolution, which enables them to automatically add new columns of rich data when those columns belong.

Schema enforcement rejects any new columns or other schema changes that aren't compatible with your table. By setting and upholding these high standards, analysts and engineers can trust that their data has the highest levels of integrity and can reason about it with clarity, allowing them to make better business decisions.

On the flip side of the coin, schema evolution complements enforcement by making it easy for intended schema changes to take place automatically. After all, it shouldn't be hard to add a column.

Schema enforcement is the yin to schema evolution's yang. When used together, these features make it easier than ever to block out the noise and tune in to the signal.

Understanding table schemas

Every DataFrame in Apache Spark contains a schema, a blueprint that defines the shape of the data, such as data types and columns, and metadata. With Delta Lake, the table's schema is saved in JSON format inside the transaction log.

What is schema enforcement?

Schema enforcement, or schema validation, is a safeguard in Delta Lake that ensures data quality by rejecting writes to a table that don't match the table's schema.

Like the front-desk manager at a busy restaurant who only accepts reservations, it checks to see whether each column of data inserted into the table is on its list of expected columns (in other words, whether each one has a "reservation"), and rejects any writes with columns that aren't on the list.

How does schema enforcement work?

Delta Lake uses **schema validation on write,** which means that all new writes to a table are checked for compatibility with the target table's schema at write time. If the schema is not compatible, Delta Lake cancels the transaction altogether (no data is written), and raises an exception to let the user know about the mismatch.

To determine whether a write to a table is compatible, Delta Lake uses the following rules. The DataFrame to be written cannot contain:

- **Any additional columns that are not present in the target table's schema.** Conversely, it's OK if the incoming data doesn't contain every column in the table — those columns will simply be assigned null values.

- **Column data types that differ from the column data types in the target table.** If a target table's column contains StringType data, but the corresponding column in the DataFrame contains IntegerType data, schema enforcement will raise an exception and prevent the write operation from taking place.

- **Column names that differ only by case.** This means that you cannot have columns such as "Foo" and "foo" defined in the same table. While Spark can be used in case sensitive or insensitive (default) mode, Delta Lake is case-preserving but insensitive when storing the schema. Parquet is case sensitive when storing and returning column information. To avoid potential mistakes, data corruption or loss issues (which we've personally experienced at Databricks), we decided to add this restriction.

Rather than automatically adding the new columns, Delta Lake enforces the schema, and stops the write from occurring. To help identify which column(s) caused the mismatch, Spark prints out both schemas in the stack trace for comparison.

How is schema enforcement useful?

Because it's such a stringent check, schema enforcement is an excellent tool to use as a gatekeeper for a clean, fully transformed data set that is ready for production or consumption. It's typically enforced on tables that directly feed:

- Machine learning algorithms
- BI dashboards
- Data analytics and visualization tools
- Any production system requiring highly structured, strongly typed, semantic schemas

In order to prepare their data for this final hurdle, many users employ a simple multi-hop architecture that progressively adds structure to their tables. To learn more, take a look at Productionizing Machine Learning With Delta Lake.

What is schema evolution?

Schema evolution is a feature that allows users to easily change a table's current schema to accommodate data that is changing over time. Most commonly, it's used when performing an append or overwrite operation, to automatically adapt the schema to include one or more new columns.

How does schema evolution work?

Following up on the example from the previous section, developers can easily use schema evolution to add the new columns that were previously rejected due to a schema mismatch. Schema evolution is activated by adding `.option('mergeSchema', 'true')` to your `.write` or `.writeStream` Spark command, as shown in the following example.

```
#Add the mergeSchema option
loans.write.format("delta") \
.option("mergeSchema", "true") \
.mode("append") \
.save(DELTALAKE_SILVER_PATH)
```

By including the `mergeSchema` option in your query, any columns that are present in the DataFrame but not in the target table are automatically added to the end of the schema as part of a write transaction. Nested fields can also be added, and these fields will get added to the end of their respective struct columns as well.

Data engineers and scientists can use this option to add new columns (perhaps a newly tracked metric, or a column of this month's sales figures) to their existing ML production tables without breaking existing models that rely on the old columns.

The following types of schema changes are eligible for schema evolution during table appends or overwrites:

- Adding new columns (this is the most common scenario)
- Changing of data types from NullType → any other type, or upcasts from ByteType → ShortType → IntegerType

Other changes, not eligible for schema evolution, require that the schema and data are overwritten by adding `.option("overwriteSchema","true")`. Those changes include:

- Dropping a column
- Changing an existing column's data typeC (in place)
- Renaming column names that differ onlyC by case (e.g., "Foo" and "foo")

Finally, with the release of Spark 3.0, explicit DDL (using `ALTER TABLE`) is fully supported, allowing users to perform the following actions on table schemas:

- Adding columns
- Changing column comments
- Setting table properties that define the behavior of the table, such as setting the retention duration of the transaction log

How is schema evolution useful?

Schema evolution can be used anytime you *intend* to change the schema of your table (as opposed to where you accidentally added columns to your DataFrame that shouldn't be there). It's the easiest way to migrate your schema because it automatically adds the correct column names and data types, without having to declare them explicitly.

Summary

Schema enforcement rejects any new columns or other schema changes that aren't compatible with your table. By setting and upholding these high standards, analysts and engineers can trust that their data has the highest levels of integrity and can reason about it with clarity, allowing them to make better business decisions. On the flip side of the coin, schema evolution complements enforcement by making it easy for intended schema changes to take place automatically. After all, it shouldn't be hard to add a column.

Schema enforcement is the yin to schema evolution's yang. When used together, these features make it easier than ever to block out the noise and tune in to the signal.

Want to learn more about schema enforcement and evolution?

Read our blog post > Watch our tech talk >

Delta Lake DML Internals

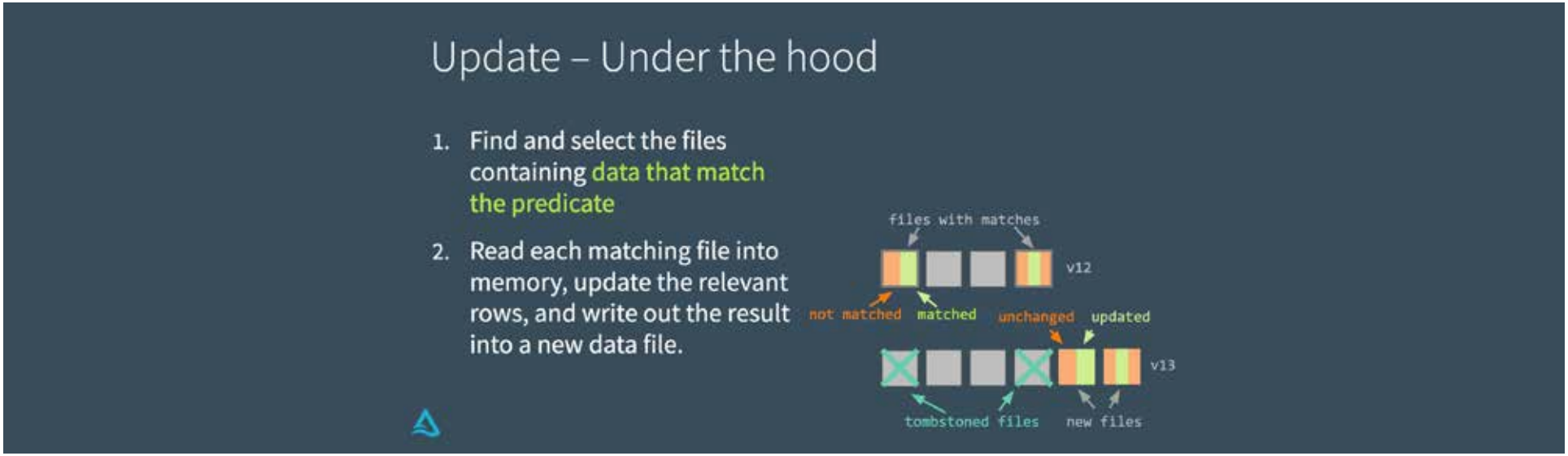
Delta Lake supports data manipulation language (DML) commands including UPDATE, DELETE and MERGE. These commands simplify change data capture (CDC), audit and governance, and GDPR/CCPA workflows, among others.

In this chapter, we will demonstrate how to use each of these DML commands, describe what Delta Lake is doing behind the scenes, and offer some performance tuning tips for each one.

Delta Lake DML: UPDATE

You can use the UPDATE operation to selectively update any rows that match a filtering condition, also known as a predicate. The code below demonstrates how to use each type of predicate as part of an UPDATE statement. Note that Delta Lake offers APIs for Python, Scala and SQL, but for the purposes of this eBook, we'll include only the SQL code.

```
-- Update events
UPDATE events SET eventType='click' WHERE buttonPress = 1
```



UPDATE: Under the hood

Delta Lake performs an UPDATE on a table in two steps:

1. Find and select the files containing data that match the predicate and, therefore, need to be updated. Delta Lake uses data skipping whenever possible to speed up this process.

2. Read each matching file into memory, update the relevant rows, and write out the result into a new data file.

Once Delta Lake has executed the UPDATE successfully, it adds a commit in the transaction log indicating that the new data file will be used in place of the old one from now on. The old data file is not deleted, though. Instead, it's simply "tombstoned" — recorded as a data file that applied to an older version of the table, but not the current version. Delta Lake is able to use it to provide data versioning and time travel.

UPDATE + Delta Lake time travel = Easy debugging

Keeping the old data files turns out to be very useful for debugging because you can use Delta Lake "time travel" to go back and query previous versions of a table at any time. In the event that you update your table incorrectly and want to figure out what happened, you can easily compare two versions of a table to one another to see what has changed.

```
SELECT * FROM events VERSION AS OF 11 EXCEPT ALL SELECT
* FROM mytable VERSION AS OF 12
```

UPDATE: Performance tuning tips

The main way to improve the performance of the UPDATE command on Delta Lake is to add more predicates to narrow down the search space. The more specific the search, the fewer files Delta Lake needs to scan and/or modify.

Delta Lake DML: DELETE

You can use the DELETE command to selectively delete rows based upon a predicate (filtering condition).

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

In the event that you want to revert an accidental DELETE operation, you can use time travel to roll back your table to the way it was.

DELETE: Under the hood

DELETE works just like UPDATE under the hood. Delta Lake makes two scans of the data: The first scan is to identify any data files that contain rows matching the predicate condition. The second scan reads the matching data files into memory, at which point Delta Lake deletes the rows in question before writing out the newly clean data to disk.

After Delta Lake completes a DELETE operation successfully, the old data files are not deleted entirely — they're still retained on disk, but recorded as "tombstoned" (no longer part of the active table) in the Delta Lake transaction log. Remember, those old files aren't deleted immediately because you might still need them to time travel back to an earlier version of the table. If you want to delete files older than a certain time period, you can use the VACUUM command.

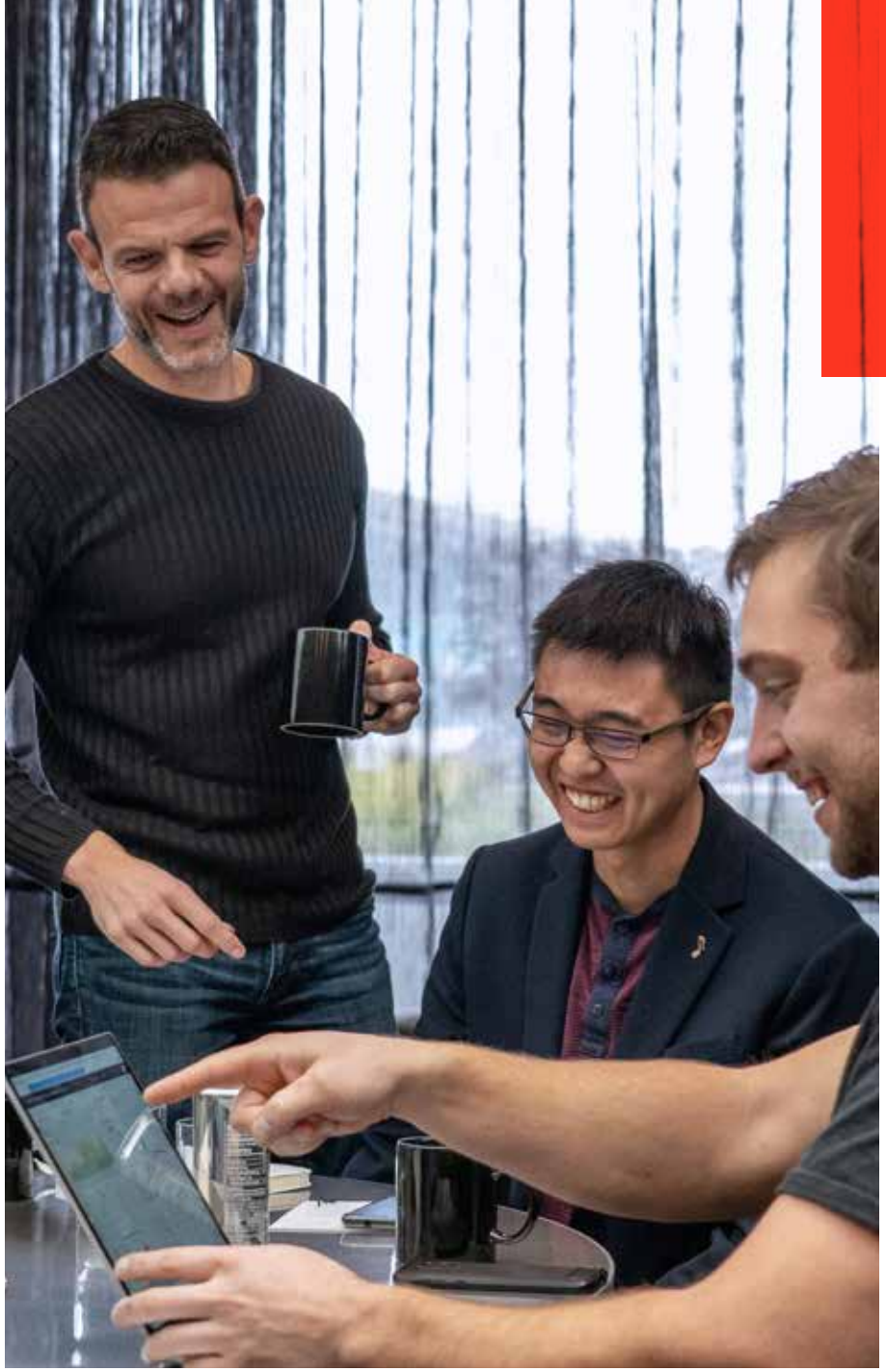
DELETE + VACUUM: Cleaning up old data files

Running the VACUUM command permanently deletes all data files that are:

1. No longer part of the active table and
2. Older than the retention threshold, which is seven days by default

Delta Lake does not automatically VACUUM old files — you must run the command yourself, as shown below. If you want to specify a retention period that is different from the default of seven days, you can provide it as a parameter.

```
from delta.tables import * deltaTable.
# vacuum files older than 30 days(720 hours)
deltaTable.vacuum(720)
```



DELETE: Performance tuning tips

Just like with the UPDATE command, the main way to improve the performance of a DELETE operation on Delta Lake is to add more predicates to narrow down the search space. The Databricks managed version of Delta Lake also features other performance enhancements like improved data skipping, the use of bloom filters, and Z-Order Optimize (multi-dimensional clustering). Read more about Z-Order Optimize on Databricks.

Delta Lake DML: MERGE

The Delta Lake MERGE command allows you to perform upserts, which are a mix of an UPDATE and an INSERT. To understand upserts, imagine that you have an existing table (aka a target table), and a source table that contains a mix of new records and updates to existing records.

Here's how an upsert works:

- When a record from the source table matches a preexisting record in the target table, Delta Lake updates the record.
- When there is no such match, Delta Lake inserts the new record.

The Delta Lake MERGE command greatly simplifies workflows that can be complex and cumbersome with other traditional data formats like Parquet. Common scenarios where merges/upserts come in handy include change data capture, GDPR/CCPA compliance, sessionization, and deduplication of records.

For more information about upserts, read:

Efficient Upserts Into Data Lakes With Databricks Delta

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

Schema Evolution in Merge Operations and Operational Metrics in Delta Lake

MERGE: Under the hood

Delta Lake completes a MERGE in two steps:

1. Perform an inner join between the target table and source table to select all files that have matches.
2. Perform an outer join between the selected files in the target and source tables and write out the updated/deleted/inserted data.

The main way that this differs from an UPDATE or a DELETE under the hood is that Delta Lake uses joins to complete a MERGE. This fact allows us to utilize some unique strategies when seeking to improve performance.

MERGE: Performance tuning tips

To improve performance of the MERGE command, you need to determine which of the two joins that make up the merge is limiting your speed.

If the inner join is the bottleneck (i.e., finding the files that Delta Lake needs to rewrite takes too long), try the following strategies:

- Add more predicates to narrow down the search space.
- Adjust shuffle partitions.
- Adjust broadcast join thresholds.
- Compact the small files in the table if there are lots of them, but don't compact them into files that are too large, since Delta Lake has to copy the entire file to rewrite it.

On Databricks' managed Delta Lake, use Z-Order optimize to exploit the locality of updates.

On the other hand, if the outer join is the bottleneck (i.e., rewriting the actual files themselves takes too long), try the strategies below.

- **Adjust shuffle partitions:** Reduce files by enabling automatic repartitioning before writes (with Optimized Writes in Databricks Delta Lake).
- **Adjust broadcast thresholds:** If you're doing a full outer join, Spark cannot do a broadcast join, but if you're doing a right outer join, Spark can do one, and you can adjust the broadcast thresholds as needed.
- **Cache the source table / DataFrame:** Caching the source table can speed up the second scan, but be sure not to cache the target table, as this can lead to cache coherency issues.

Delta Lake supports DML commands including UPDATE, DELETE and MERGE INTO, which greatly simplify the workflow for many common big data operations. In this chapter, we demonstrated how to use these commands in Delta Lake, shared information about how each one works under the hood, and offered some performance tuning tips.

Want a deeper dive into DML internals, including snippets of code?

Read our blog post >

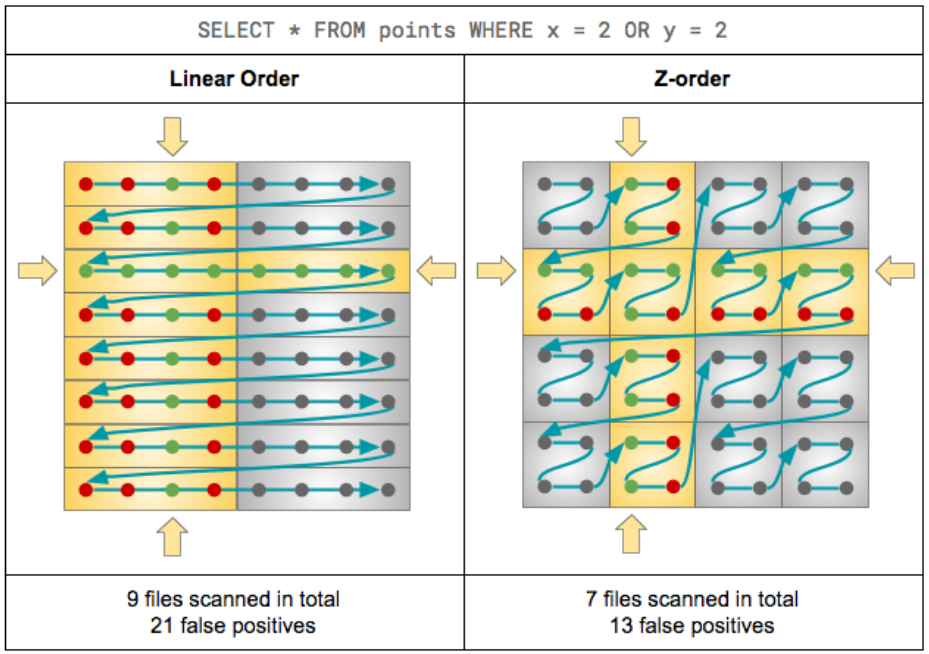
How Delta Lake Quickly Processes Petabytes With Data Skipping and Z-Ordering

Delta Lake is capable of sifting through petabytes of data within seconds. Much of this speed is owed to two features: (1) data skipping and (2) Z-Ordering.

Combining these features helps the Databricks Runtime to dramatically reduce the amount of data that needs to be scanned to answer selective queries against large Delta tables, which typically translates into substantial runtime improvements and cost savings.

Using Delta Lake's built-in data skipping and ZORDER clustering features, large cloud data lakes can be queried in a matter of seconds by skipping files not relevant to the query. For example, 93.2% of the records in a 504 TB data set were skipped for a typical query in a real-world cybersecurity analysis use case, reducing query times by up to two orders of magnitude. In other words, Delta Lake can speed up your queries by as much as 100x.

Want to see data skipping and Z-Ordering in action?
Apple's Dominique Brezinski and Databricks' Michael Armbrust demonstrated how to use Delta Lake as a unified solution for data engineering and data science in the context of cybersecurity monitoring and threat response. Watch their keynote speech, Threat Detection and Response at Scale.



Using data skipping and Z-Order clustering

Data skipping and Z-Ordering are used to improve the performance of needle-in-the-haystack queries against huge data sets. Data skipping is an automatic feature of Delta Lake, kicking in whenever your SQL queries or data set operations include filters of the form "column op literal," where:

- `column` is an attribute of some Delta Lake table, be it top-level or nested, whose data type is string / numeric / date/ timestamp
- `op` is a binary comparison operator, `StartsWith / LIKE pattern%', or IN <list_of_values>`
- `literal` is an explicit (list of) value(s) of the same data type as a column

`AND / OR / NOT` are also supported as well as "literal op column" predicates.

Even though data skipping kicks in when the above conditions are met, it may not always be effective. But, if there are a few columns that you frequently filter by and want to make sure that's fast, then you can explicitly optimize your data layout with respect to skipping effectiveness by running the following command:

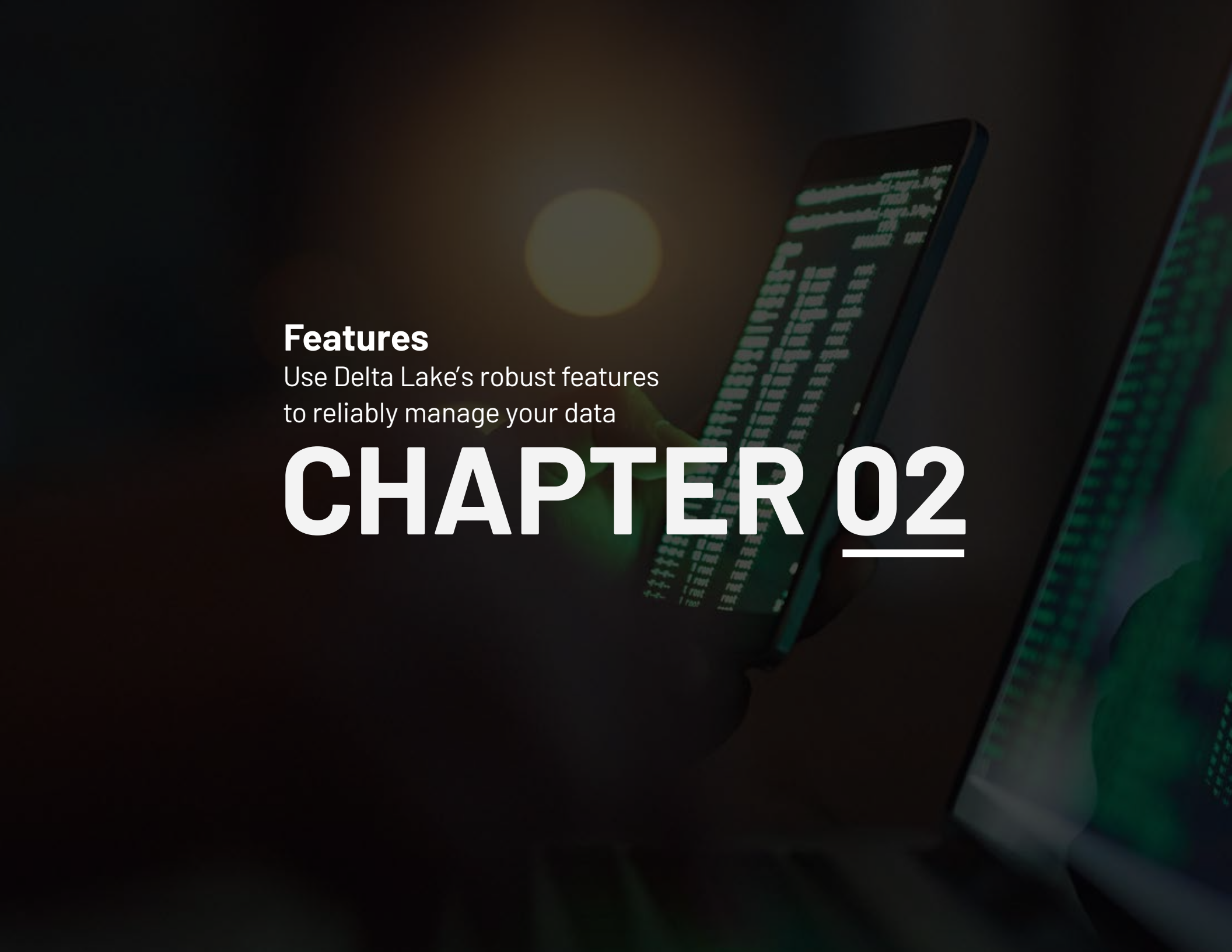
```
OPTIMIZE <table> [WHERE <partition_filter>]
ZORDER BY (<column> [, …])
```

Exploring the details

Apart from partition pruning, another common technique that's used in the data warehousing world, but which Spark currently lacks, is I/O pruning based on small materialized aggregates. In short, the idea is to keep track of simple statistics such as minimum and maximum values at a certain granularity that are correlated with I/O granularity. And we want to leverage those statistics at query planning time in order to avoid unnecessary I/O.

This is exactly what Delta Lake's data skipping feature is about. As new data is inserted into a Delta Lake table, file-level min/max statistics are collected for all columns (including nested ones) of supported types. Then, when there's a lookup query against the table, Delta Lake first consults these statistics in order to determine which files can safely be skipped.

Want to learn more about data skipping and Z-Ordering, including how to apply it within a cybersecurity analysis?
Read our blog post >



Features

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

CHAPTER 02

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?

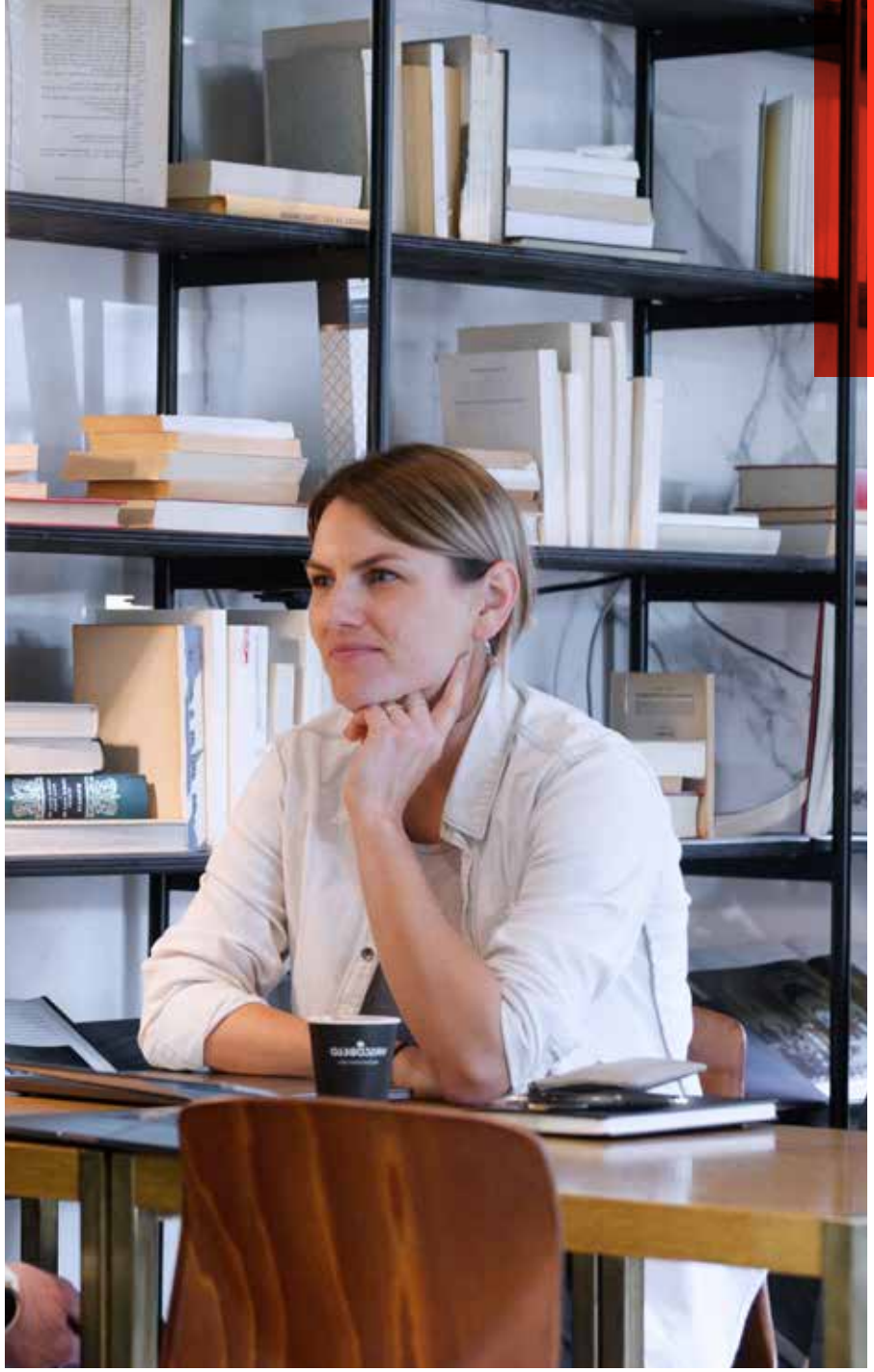
There are a number of common use cases where existing data in a data lake needs to be updated or deleted:

- **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)
```

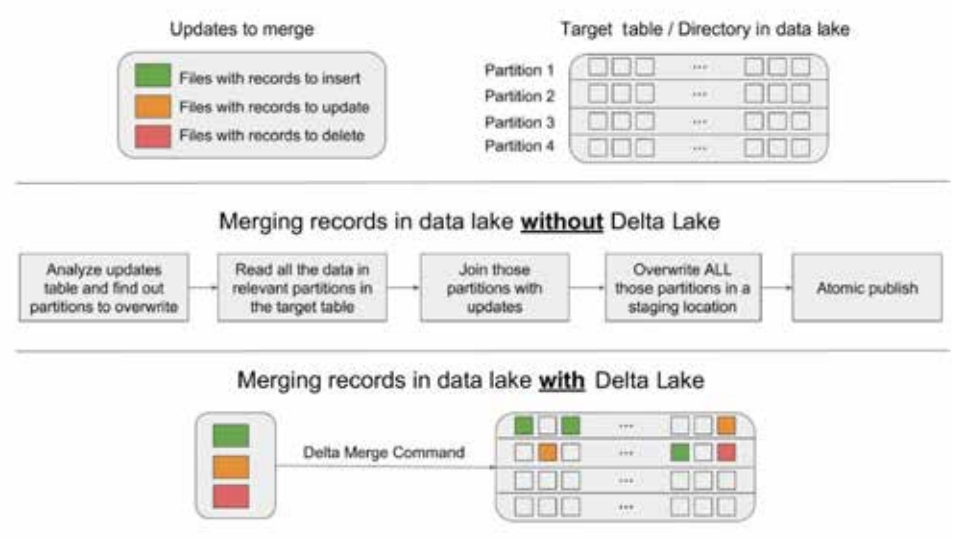
This will perform exactly what the syntax says — for existing users (i.e., MATCHED clause), it will update the address column, and for new users (i.e., NOT MATCHED clause) it will insert all the columns. For large tables with TBs of data, this Delta Lake MERGE operation can be orders of magnitude faster 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.



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

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

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)
}
```

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

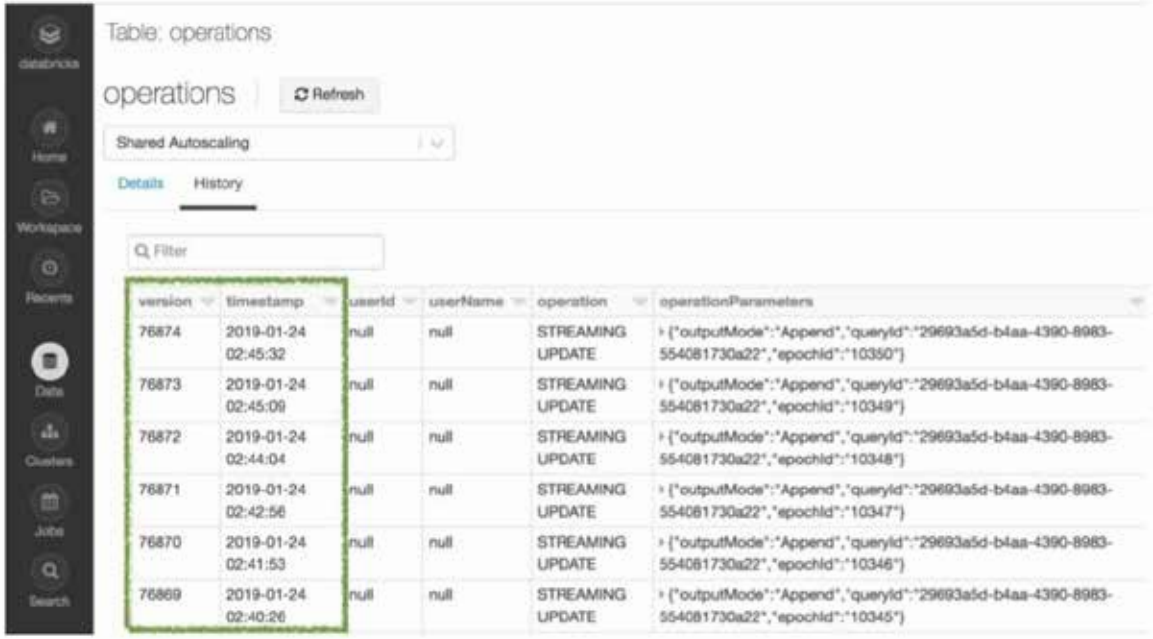
```
val df = spark.read
  .format("delta")
  .option("versionAsOf", "5238")
  .load("/path/to/my/table")

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
```



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

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
```

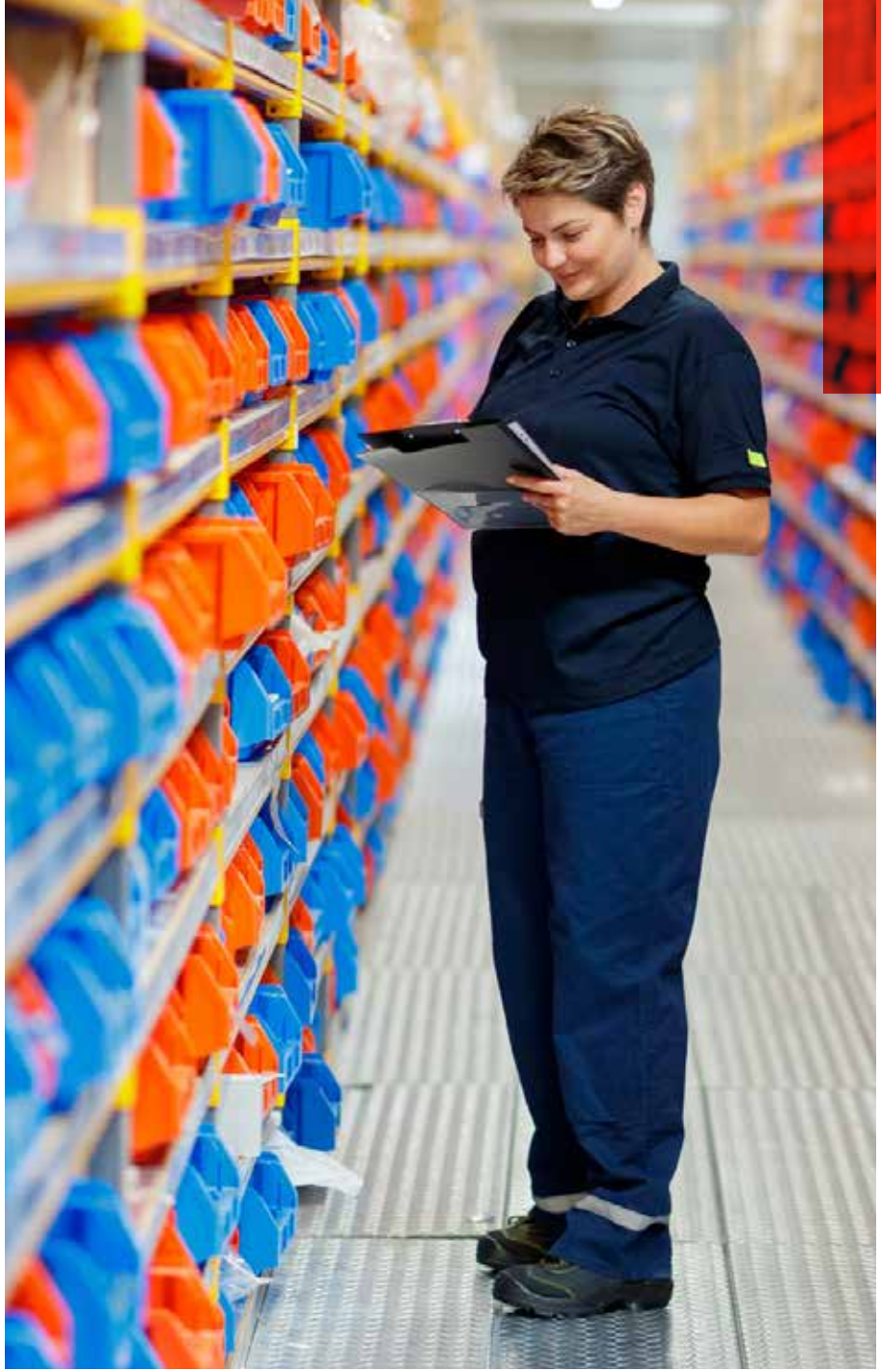
Once you're happy with the results, you have two options. If no other change has been made to your source table, you can replace your source table with the clone. If changes have been made to your source table, you can merge the changes into your source table.

```
-- 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 resource

Simplifying Disaster Recovery With Delta Lake

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.

```
1 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

Lakehouse

Combining the best elements of data lakes and data warehouses

CHAPTER 03

What Is a Lakehouse?

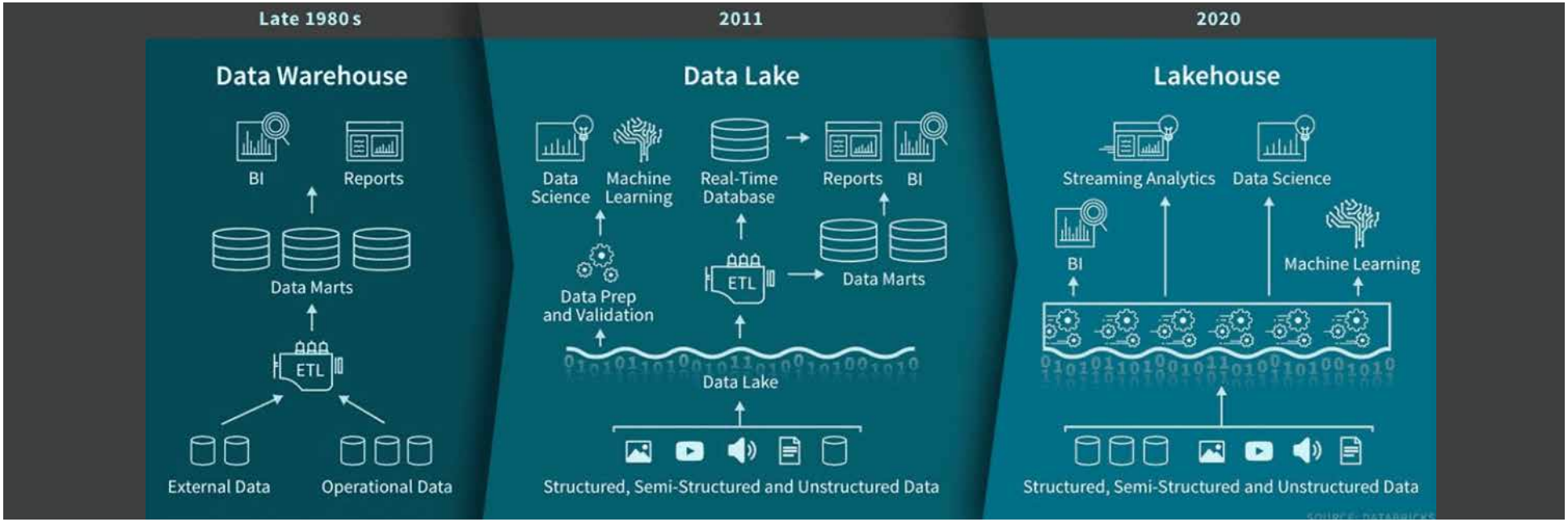
Over the past few years at Databricks, we've seen a new data management architecture that emerged independently across many customers and use cases: the **lakehouse.** In this chapter, we'll describe this new architecture and its advantages over previous approaches.

Data warehouses have a long history of decision support and business intelligence applications. Since its inception in the late 1980s, data warehouse technology continued to evolve and MPP architectures led to systems that were able to handle larger data sizes.

But while warehouses were great for structured data, a lot of modern enterprises have to deal with unstructured data, semi-structured data, and data with high variety, velocity and volume. Data warehouses are not suited for many of these use cases, and they are certainly not the most cost-efficient.

As companies began to collect large amounts of data from many different sources, architects began envisioning a single system to house data for many different analytic products and workloads.

About a decade ago, companies began building data lakes — repositories for raw data in a variety of formats. While suitable for storing data, data lakes lack some critical features: They do not support transactions, they do not enforce data quality, and their lack of consistency / isolation makes it almost impossible to mix appends and reads,



and batch and streaming jobs. For these reasons, many of the promises of data lakes have not materialized and, in many cases, lead to a loss of many of the benefits of data warehouses.

The need for a flexible, high-performance system hasn't abated. Companies require systems for diverse data applications including SQL analytics, real-time monitoring, data science and machine learning. Most of the recent advances in AI have been in better models to process unstructured data (text, images, video, audio), but these are precisely the types of data that a data warehouse is not optimized for.

A common approach is to use multiple systems — a data lake, several data warehouses, and other specialized systems such as streaming, time-series, graph and image databases. Having a multitude of systems introduces complexity and, more importantly, introduces delay as data professionals invariably need to move or copy data between different systems.

A lakehouse combines the best elements of data lakes and data warehouses

A lakehouse is a new data architecture that combines the best elements of data lakes and data warehouses.

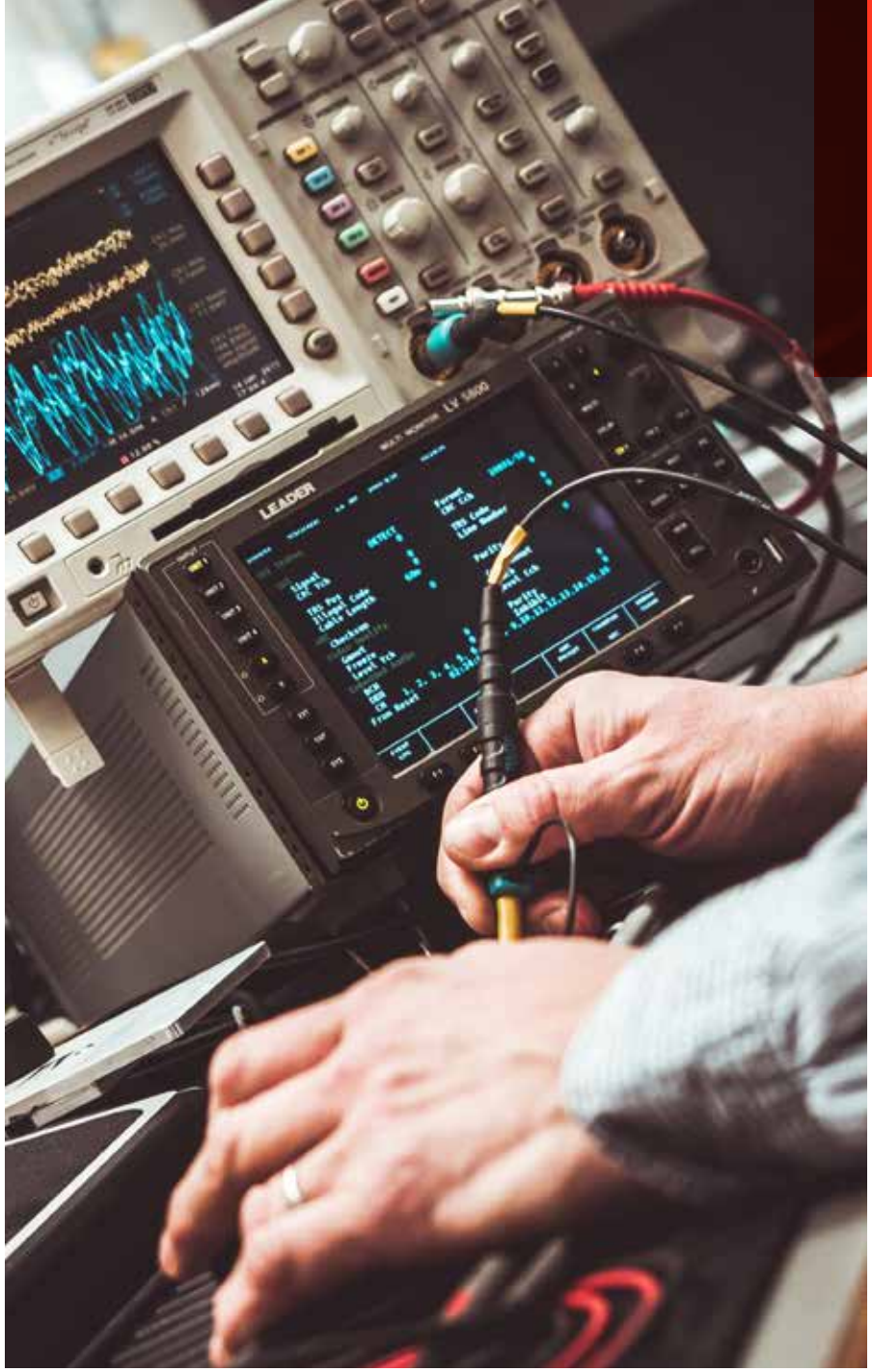
Lakehouses are enabled by a new system design: implementing similar data structures and data management features to those in a data warehouse, directly on the kind of low-cost storage used for data lakes. They are what you would get if you had to redesign data warehouses in the modern world, now that cheap and highly reliable storage (in the form of object stores) are available.

A lakehouse has the following key features:

- **Transaction support:** In an enterprise lakehouse, many data pipelines will often be reading and writing data concurrently. Support for ACID transactions ensures consistency as multiple parties concurrently read or write data, typically using SQL.

- **Schema enforcement and governance:** The lakehouse should have a way to support schema enforcement and evolution, supporting DW schema paradigms such as star/snowflake-schemas. The system should be able to reason about data integrity, and it should have robust governance and auditing mechanisms.
- **BI support:** Lakehouses enable using BI tools directly on the source data. This reduces staleness and improves recency, reduces latency and lowers the cost of having to operationalize two copies of the data in both a data lake and a warehouse.
- **Storage is decoupled from compute:** In practice, this means storage and compute use separate clusters, thus these systems are able to scale to many more concurrent users and larger data sizes. Some modern data warehouses also have this property.
- **Openness:** The storage formats they use are open and standardized, such as Parquet, and they provide an API so a variety of tools and engines, including machine learning and Python/R libraries, can efficiently access the data directly.
- **Support for diverse data types ranging from unstructured to structured data:** The lakehouse can be used to store, refine, analyze and access data types needed for many new data applications, including images, video, audio, semi-structured data, and text.
- **Support for diverse workloads:** Including data science, machine learning and SQL analytics. Multiple tools might be needed to support all these workloads, but they all rely on the same data repository.
- **End-to-end streaming:** Real-time reports are the norm in many enterprises. Support for streaming eliminates the need for separate systems dedicated to serving real-time data applications.

These are the key attributes of lakehouses. Enterprise-grade systems require additional features. Tools for security and access control are basic requirements. Data governance capabilities including auditing, retention and lineage have become essential particularly in light of recent privacy regulations. Tools that enable data discovery such as data catalogs and data usage metrics are also needed. With a lakehouse, such enterprise features only need to be implemented, tested and administered for a single system.



Read the research

Delta Lake: High-Performance ACID Table Storage Over Cloud Object Stores

Abstract

Cloud object stores such as Amazon S3 are some of the largest and most cost-effective storage systems on the planet, making the main attractive target to store large data warehouses and data lakes. Unfortunately, their implementation as key-value stores makes it difficult to achieve ACID transactions and high performance: Metadata operations, such as listing objects, are expensive, and consistency guarantees are limited. In this paper, we present Delta Lake, an open source ACID table storage layer over cloud object stores initially developed at Databricks. Delta Lake uses a transaction log that is compacted into Apache Parquet format to provide ACID properties, time travel, and significantly faster metadata operations for large tabular data sets (e.g., the ability to quickly search billions of table partitions for those relevant to a query). It also leverages this design to provide high-level features such as automatic data layout optimization, upserts, caching, and audit logs. Delta Lake tables can be accessed from Apache Spark, Hive, Presto, Redshift, and other systems. Delta Lake is deployed at thousands of Databricks customers that process exabytes of data per day, with the largest instances managing exabyte-scale data sets and billions of objects.

Authors: Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hövell, Adrian Ionescu, Alicja Łuszczak, Michał Szafra´nski, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, Matei Zaharia

Read the full research paper on the inner workings of the lakehouse.

Some early examples

The Databricks Unified Data Platform has the architectural features of a lakehouse. Microsoft's Azure Synapse Analytics service, which integrates with Azure Databricks, enables a similar lakehouse pattern. Other managed services such as BigQuery and Redshift Spectrum have some of the lakehouse features listed above, but they are examples that focus primarily on BI and other SQL applications.

Companies that want to build and implement their own systems have access to open source file formats (Delta Lake, Apache Iceberg, Apache Hudi) that are suitable for building a lakehouse.

Merging data lakes and data warehouses into a single system means that data teams can move faster as they are able to use data without needing to access multiple systems. The level of SQL support and integration with BI tools among these early lakehouses is generally sufficient for most enterprise data warehouses. Materialized views and stored procedures are available, but users may need to employ other mechanisms that aren't equivalent to those found in traditional data warehouses. The latter is particularly important for "lift and shift scenarios," which require systems that achieve semantics that are almost identical to those of older, commercial data warehouses.

What about support for other types of data applications? Users of a lakehouse have access to a variety of standard tools (Apache Spark, Python, R, machine learning libraries) for non-BI workloads like data science and machine learning. Data exploration and refinement are standard for many analytic and data science applications. Delta Lake is designed to let users incrementally improve the quality of data in their lakehouse until it is ready for consumption.

A note about technical building blocks. While distributed file systems can be used for the storage layer, object stores are more commonly used in lakehouses. Object stores provide low-cost, highly available storage that excels at massively parallel reads — an essential requirement for modern data warehouses.

From BI to AI

The lakehouse is a new data management architecture that radically simplifies enterprise data infrastructure and accelerates innovation in an age when machine learning is poised to disrupt every industry. In the past, most of the data that went into a company's products or decision-making was structured data from operational systems, whereas today, many products incorporate AI in the form of computer vision and speech models, text mining and others. Why use a lakehouse instead of a data lake for AI? A lakehouse gives you data versioning, governance, security and ACID properties that are needed even for unstructured data.

Current lakehouses reduce cost, but their performance can still lag specialized systems (such as data warehouses) that have years of investments and real-world deployments behind them. Users may favor certain tools (BI tools, IDEs, notebooks) over others so lakehouses will also need to improve their UX and their connectors to popular tools so they can appeal to a variety of personas. These and other issues will be addressed as the technology continues to mature and develop. Over time, lakehouses will close these gaps while retaining the core properties of being simpler, more cost-efficient and more capable of serving diverse data applications.

Diving Deep Into the Inner Workings of the Lakehouse and Delta Lake

Databricks wrote a blog article that outlined how more and more enterprises are adopting the lakehouse pattern. The blog created a massive amount of interest from technology enthusiasts. While lots of people praised it as the next-generation data architecture, some people thought the lakehouse is the same thing as the data lake. Recently, several of our engineers and founders wrote a research paper that describes some of the core technological challenges and solutions that set the lakehouse architecture apart from the data lake, and it was accepted and published at the International Conference on Very Large Databases (VLDB) 2020. You can read the paper, "Delta Lake: High-Performance ACID Table Storage Over Cloud Object Stores," here.

Henry Ford is often credited with having said, "If I had asked people what they wanted, they would have said faster horses." The crux of this statement is that people often envision a better solution to a problem as an evolution of what they already know rather than rethinking the approach to the problem altogether. In the world of data storage, this pattern has been playing out for years. Vendors continue to try to reinvent the old horses of data warehouses and data lakes rather than seek a new solution.

More than a decade ago, the cloud opened a new frontier for data storage. Cloud object stores like Amazon S3 have become some of the largest and most cost-effective storage systems in the world, which makes them an attractive platform to store data warehouses and data lakes. However, their nature as key-value stores makes it difficult to achieve ACID transactions that many organizations require. Also, performance is hampered by expensive metadata operations (e.g., listing objects) and limited consistency guarantees.

Based on the characteristics of cloud object stores, three approaches have emerged.

1. Data lakes

The first is directories of files (i.e., data lakes) that store the table as a collection of objects, typically in columnar format such as Apache Parquet. It's an attractive approach because the table is just a group of objects that can be accessed from a wide variety of tools without a lot of additional data stores or systems. However, both performance and consistency problems are common. Hidden data corruption is common due to failed transactions, eventual consistency leads to inconsistent queries, latency is high, and basic management capabilities like table versioning and audit logs are unavailable.

2. Custom storage engines

The second approach is custom storage engines, such as proprietary systems built for the cloud like the Snowflake data warehouse. These systems can bypass the consistency challenges of data lakes by managing the metadata in a separate, strongly consistent service that's able to provide a single source of truth. However, all I/O operations need to connect to this metadata service, which can increase cloud resource costs and reduce performance and availability. Additionally, it takes a lot of engineering work to implement connectors to existing computing engines like Apache Spark, TensorFlow and PyTorch, which can be challenging for data teams that use a variety of computing engines on their data. Engineering challenges can be exacerbated by unstructured data because these systems are generally optimized for traditional structured

data types. Finally, and most egregiously, the proprietary metadata service locks customers into a specific service provider, leaving customers to contend with consistently high prices and expensive, time-consuming migrations if they decide to adopt a new approach later.

3. Lakehouse

With Delta Lake, an open source ACID table storage layer atop cloud object stores, we sought to build a car instead of a faster horse with not just a better data store, but a fundamental change in how data is stored and used via the lakehouse. A lakehouse is a new architecture that combines the best elements of data lakes and data warehouses. Lakehouses are enabled by a new system design: implementing similar data structures and data management features to those in a data warehouse, directly on the kind of low-cost storage used for data lakes. They are what you would get if you had to redesign storage engines in the modern world, now that cheap and highly reliable storage (in the form of object stores) are available.

Delta Lake maintains information about which objects are part of a Delta table in an ACID manner, using a write-ahead log, compacted into Parquet, that is also stored in the cloud object store. This design allows clients to update multiple objects at once, replace a subset of the objects with another, etc., in a serializable manner that still achieves high parallel read/write performance from the objects. The log also provides significantly faster metadata operations for large tabular data sets. Additionally, Delta Lake offers advanced capabilities like time travel (i.e., the ability to query point-in-time snapshots or roll back erroneous updates), automatic data layout optimization, upserts, caching, and audit logs. Together, these features improve both the manageability and performance of working with data in cloud object stores, ultimately opening the door to the lakehouse architecture that combines the key features of data warehouses and data lakes to create a better, simpler data architecture.

Today, Delta Lake is used across thousands of Databricks customers, processing exabytes of structured and unstructured data each day, as well as many organizations in the open source community. These use cases span a variety of data sources and applications. The data types stored include Change Data Capture (CDC) logs from enterprise OLTP systems, application logs, time-series data, graphs, aggregate tables for reporting, and image or feature data for machine learning. The applications include SQL workloads (most commonly), business intelligence, streaming, data science, machine learning and graph analytics. Overall, Delta Lake has proven itself to be a good fit for most data lake applications that would have used structured storage formats like Parquet or ORC, and many traditional data warehousing workloads.

Across these use cases, we found that customers often use Delta Lake to significantly simplify their data architecture by running more workloads directly against cloud object stores, and increasingly, by creating a lakehouse with both data lake and transactional features to replace some or all of the functionality provided by message queues (e.g., Apache Kafka), data lakes or cloud data warehouses (e.g., Snowflake, Amazon Redshift).

In the research paper, the authors explain:

- The characteristics and challenges of object stores
- The Delta Lake storage format and access protocols
- The current features, benefits and limitations of Delta Lake
- Both the core and specialized use cases commonly employed today
- Performance experiments, including TPC-DS performance

Through the paper, you'll gain a better understanding of Delta Lake and how it enables a wide range of DBMS-like performance and management features for data held in low-cost cloud storage. As well as how the Delta Lake storage format and access protocols make it simple to operate, highly available, and able to deliver high-bandwidth access to the object store.

Understanding Delta Engine

The Delta Engine ties together a 100% Apache Spark-compatible vectorized query engine to take advantage of modern CPU architecture with optimizations to Spark 3.0's query optimizer and caching capabilities that were launched as part of Databricks Runtime 7.0. Together, these features significantly accelerate query performance on data lakes, especially those enabled by Delta Lake, to make it easier for customers to adopt and scale a lakehouse architecture.

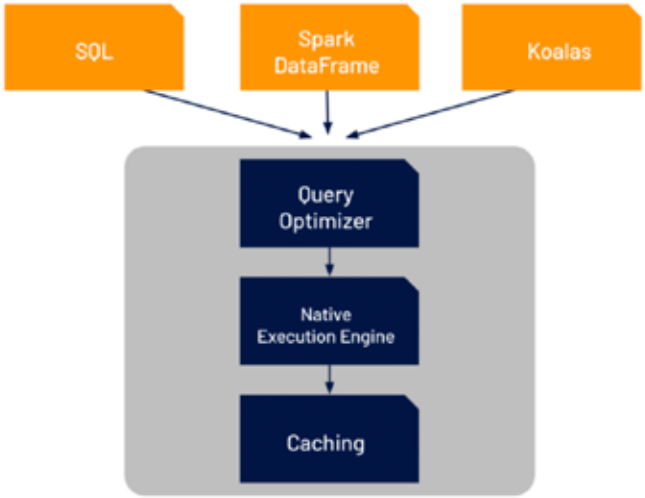
Scaling execution performance

One of the big hardware trends over the last several years is that CPU clock speeds have plateaued. The reasons are outside the scope of this chapter, but the takeaway is that we have to find new ways to process data faster beyond raw compute power. One of the most impactful methods has been to improve the amount of data that can be processed in parallel. However, data processing engines need to be specifically architected to take advantage of this parallelism.

In addition, data teams are being given less and less time to properly model data as the pace of business increases. Poorer modeling in the interest of better business agility drives poorer query performance. Naturally, this is not a desired state, and organizations want to find ways to maximize both agility and performance.

Announcing Delta Engine for high-performance query execution

Delta Engine accelerates the performance of Delta Lake for SQL and DataFrame workloads through three components: an improved query optimizer, a caching layer that sits between the execution layer and the cloud object storage, and a native vectorized execution engine that's written in C++.



The improved query optimizer extends the functionality already in Spark 3.0 (cost-based optimizer, adaptive query execution, and dynamic runtime filters) with more advanced statistics to deliver up to 18x increased performance in star schema workloads.

Delta Engine's caching layer automatically chooses which input data to cache for the user, transcoding it along the way in a more CPU-efficient format to better leverage the increased storage speeds of NVMe SSDs. This delivers up to 5x faster scan performance for virtually all workloads.

However, the biggest innovation in Delta Engine to tackle the challenges facing data teams today is the native execution engine, which we call Photon. (We know. It's in an engine within the engine...). This completely rewritten execution engine for

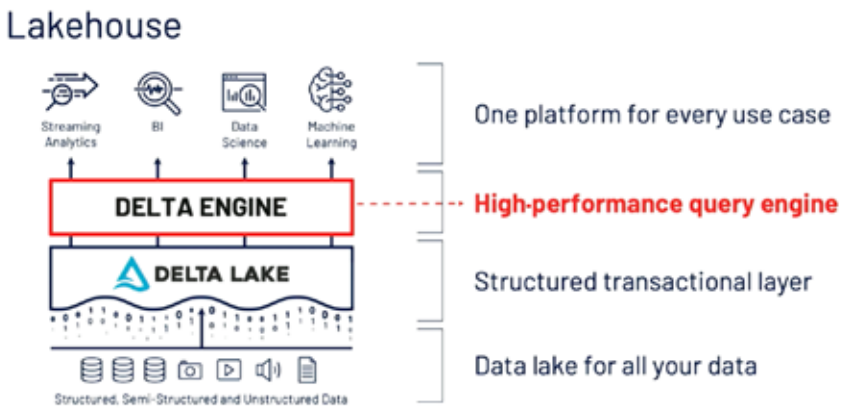
Databricks has been built to maximize the performance from the new changes in modern cloud hardware. It brings performance improvements to all workload types while remaining fully compatible with open Spark APIs.

Getting started with Delta Engine

By linking these three components together, we think it will be easier for customers to understand how improvements in multiple places within the Databricks code aggregate into significantly faster performance for analytics workloads on data lakes.

We're excited about the value that Delta Engine delivers to our customers. While the time and cost savings are already valuable, its role in the lakehouse pattern supports new advances in how data teams design their data architectures for increased unification and simplicity.

For more information on the Delta Engine, watch this keynote address from Spark + AI Summit 2020: Delta Engine: High-Performance Query Engine for Delta Lake.



Streaming

Using Delta Lake to express computation on streaming data

CHAPTER 04

How Delta Lake Solves Common Pain Points in Streaming

The pain points of a traditional streaming and data warehousing solution can be broken into two groups: data lake and data warehouse pains.

Data lake pain points

While data lakes allow you to flexibly store an immense amount of data in a file system, there are many pain points including (but not limited to):

- Consolidation of streaming data from many disparate systems is difficult.
- Updating data in a data lake is nearly impossible, and much of the streaming data needs to be updated as changes are made. This is especially important in scenarios involving financial reconciliation and subsequent adjustments.
- Query speeds for a data lake are typically very slow.
- Optimizing storage and file sizes is very difficult and often requires complicated logic.

Data warehouse pain points

The power of a data warehouse is that you have a persistent performant store of your data. But the pain points for building modern continuous applications include (but are not limited to):

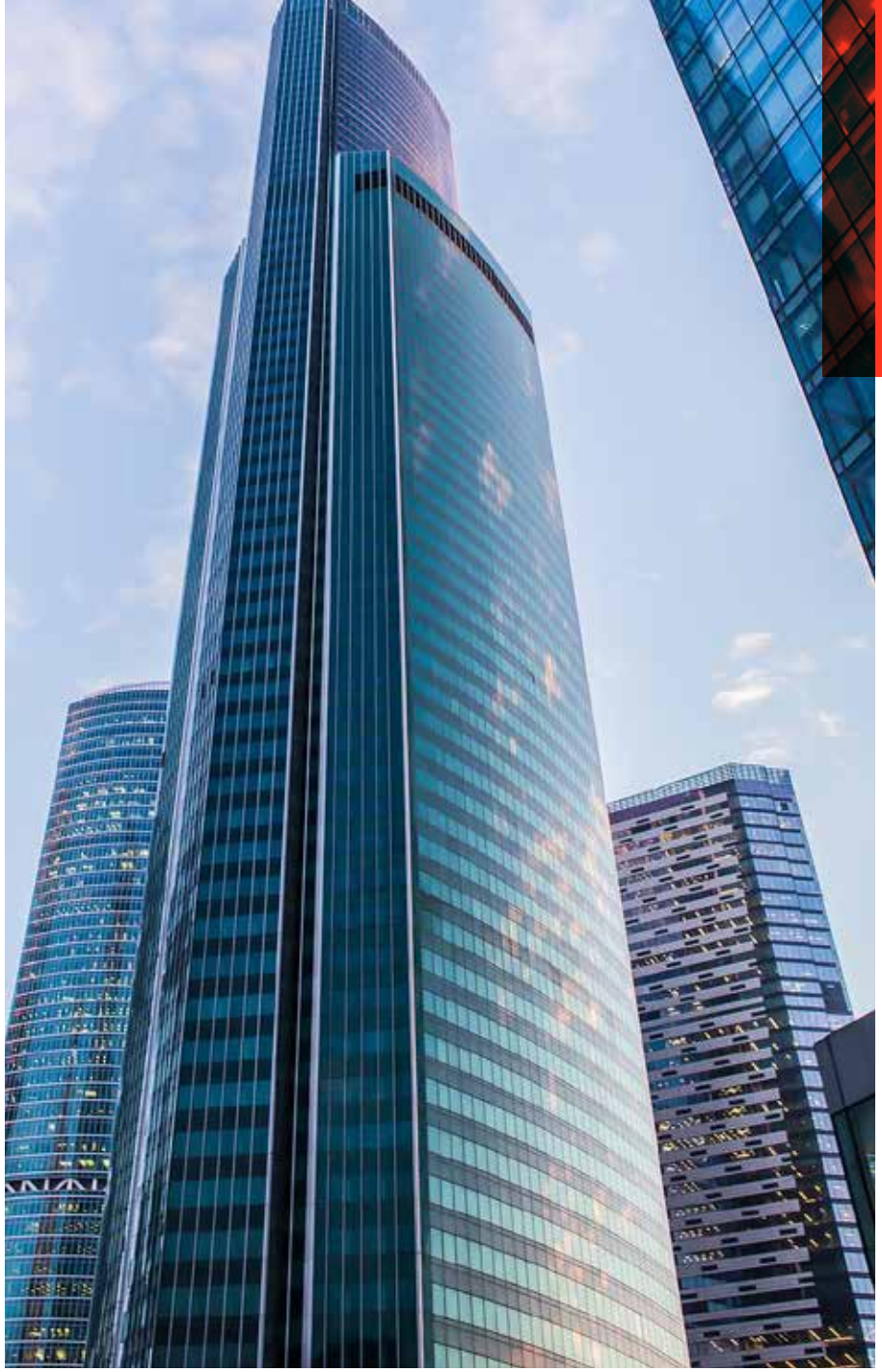
- Constrained to SQL queries (i.e., no machine learning or advanced analytics).
- Accessing streaming data and stored data together is very difficult, if at all possible.
- Data warehouses do not scale very well.
- Tying compute and storage together makes using a warehouse very expensive.

How Delta Lake on Databricks solves these issues

Delta Lake is a unified data management system that brings data reliability and performance optimizations to cloud data lakes. More succinctly, Delta Lake combines the advantages of data lakes and data warehouses with Apache Spark™ to allow you to do incredible things.

- Delta Lake, along with Structured Streaming, makes it possible to analyze streaming and historical data together at high speeds.
- When Delta Lake tables are used as sources and destinations of streaming big data, it is easy to consolidate disparate data sources.
- Upserts are supported on Delta Lake tables.
- Delta Lake is ACID compliant, making it easy to create a compliant data solution.
- Easily include machine learning scoring and advanced analytics into ETL and queries.
- Decouples compute and storage for a completely scalable solution.

In the following use cases, we'll share what this looks like in practice.



Simplifying Streaming Stock Data Analysis Using Delta Lake

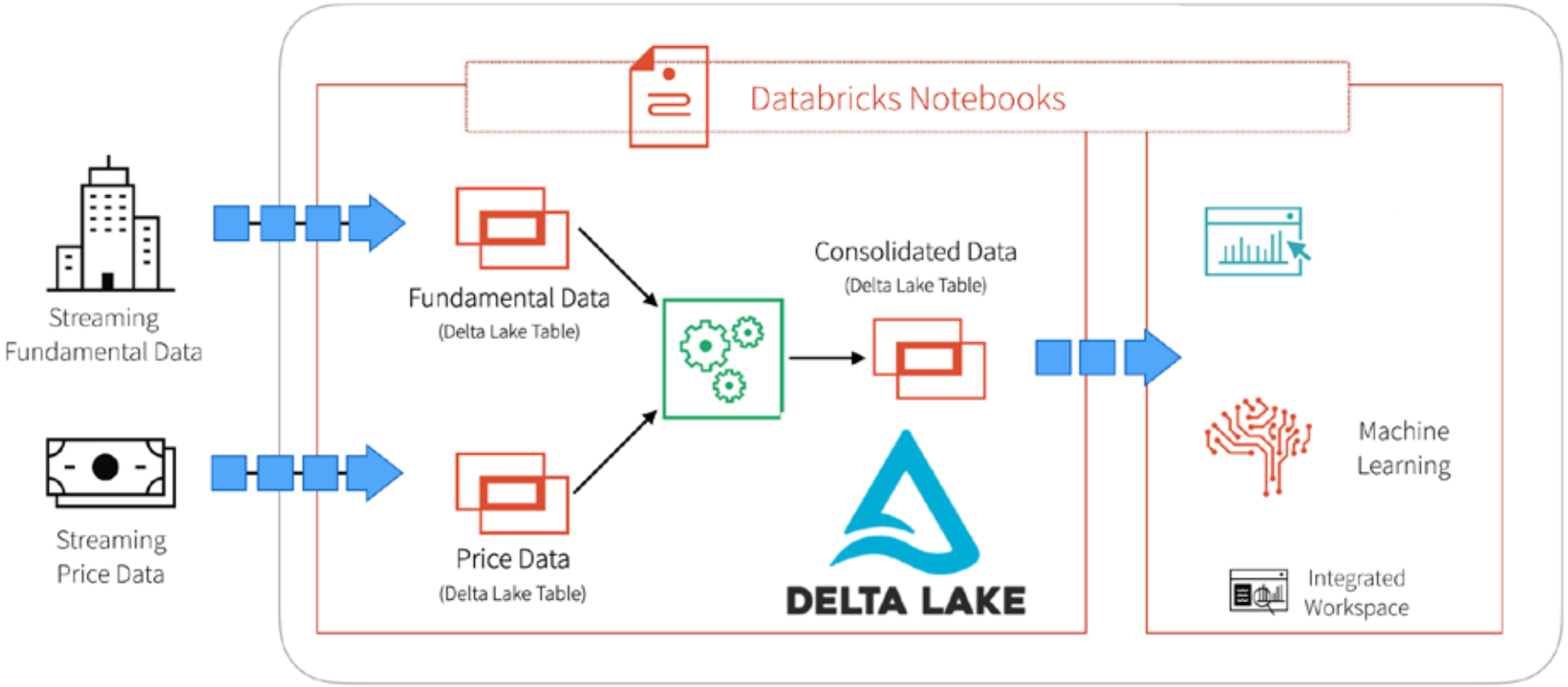
Real-time analysis of stock data is a complicated endeavor. After all, there are many challenges in maintaining a streaming system and ensuring transactional consistency of legacy and streaming data concurrently.

Thankfully, Delta Lake helps solve many of the pain points of building a streaming system to analyze stock data in real time. In this section, we'll share how to simplify the streaming of stock data analysis using Delta Lake.

In the following diagram, you can see a high-level architecture that simplifies this problem. We start by ingesting two different sets of data into two Delta Lake tables. The two data sets are stock prices and fundamentals.

After ingesting the data into their respective tables, we then join the data in an ETL process and write the data out into a third Delta Lake table for downstream analysis.

Delta Lake helps solve these problems by combining the scalability, streaming and access to the advanced analytics of Apache Spark with the performance and ACID compliance of a data warehouse.



Implement your streaming stock analysis solution with Delta Lake

Delta Lake and Apache Spark do most of the work for our solution; you can try out the full notebook and follow along with the code samples below.

As noted in the preceding diagram, we have two data sets to process — one for fundamentals and one for price data. To create our two Delta Lake tables, we specify the `.format('delta')` against our Databricks File System (DBFS) locations.

```
# Create Fundamental Data (Databricks Delta table)
dfBaseFund = spark \\
.read \\
.format('delta') \\
.load('/delta/stocksFundamentals')

# Create Price Data (Databricks Delta table)
dfBasePrice = spark \\
.read \\
.format('delta') \\
.load('/delta/stocksDailyPrices')
```

While we're updating the `stockFundamentals` and `stocksDailyPrices`, we will consolidate this data through a series of ETL jobs into a consolidated view (`stocksDailyPricesWFund`).

With the following code snippet, we can determine the start and end date of available data and then combine the price and fundamentals data for that date range into DBFS.

```
# Determine start and end date of available data
row = dfBasePrice.agg(
   func.max(dfBasePrice.price_date).alias("maxDate"),
   func.min(dfBasePrice.price_date).alias("minDate")
).collect()[0]
startDate = row["minDate"]
endDate = row["maxDate"]

# Define our date range function
def daterange(start_date, end_date):
    for n in range(int ((end_date - start_date).days)):
        yield start_date + datetime.timedelta(n)

# Define combinePriceAndFund information by date and
def combinePriceAndFund(theDate):
  dfFund = dfBaseFund.where(dfBaseFund.price_date == theDate)
  dfPrice = dfBasePrice.where(
dfBasePrice.price_date == theDate
).drop('price_date')
  # Drop the updated column
  dfPriceWFund = dfPrice.join(dfFund, ['ticker']).drop('updated')
  # Save data to DBFS
   dfPriceWFund
.write
.format('delta')
.mode('append')
.save('/delta/stocksDailyPricesWFund')

# Loop through dates to complete fundamentals + price ETL process
for single_date in daterange(
startDate, (endDate + datetime.timedelta(days=1))
):
  print 'Starting ' + single_date.strftime('%Y-%m-%d')
  start = datetime.datetime.now()
  combinePriceAndFund(single_date)
  end = datetime.datetime.now()
  print (end - start)
```

Now we have a stream of consolidated fundamentals and price data that is being pushed into DBFS in the /delta/stocksDailyPricesWFund location. We can build a Delta Lake table by specifying `.format("delta")` against that DBFS location.

```
dfPriceWithFundamentals = spark
.readStream
.format("delta")
.load("/delta/stocksDailyPricesWFund")

// Create temporary view of the data
dfPriceWithFundamentals.createOrReplaceTempView("priceWithFundamentals")
```

Now that we have created our initial Delta Lake table, let's create a view that will allow us to calculate the price/earnings ratio in real time (because of the underlying streaming data updating our Delta Lake table).

```
%sql
CREATE OR REPLACE TEMPORARY VIEW viewPE AS
select ticker,
       price_date,
       first(close) as price,
       (close/eps_basic_net) as pe
  from priceWithFundamentals
 where eps_basic_net > 0
 group by ticker, price_date, pe
```

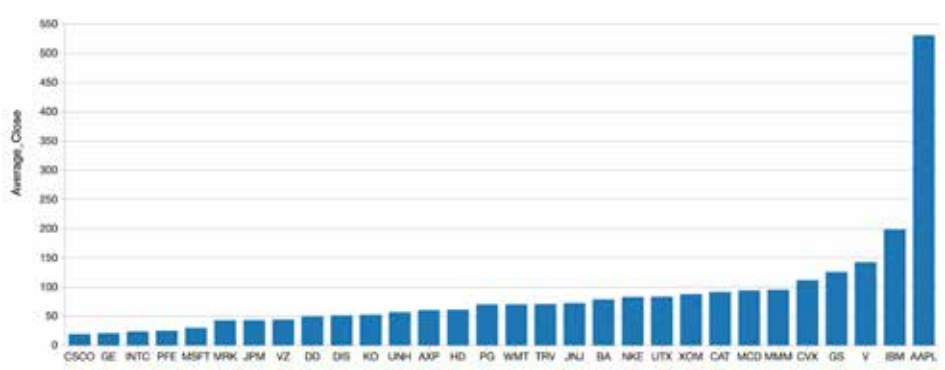
Analyze streaming stock data in real time

With our view in place, we can quickly analyze our data using Spark SQL.

```
%sql
select *
from viewPE
where ticker == "AAPL"
order by price_date
```

As the underlying source of this consolidated data set is a Delta Lake table, this view isn't just showing the batch data but also any new streams of data that are coming in as per the following streaming dashboard.

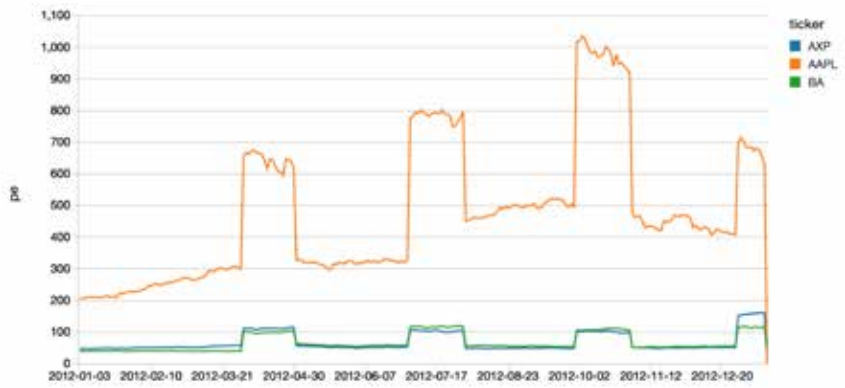
Underneath the covers, Structured Streaming isn't just writing the data to Delta Lake tables but also keeping the state of the distinct number of keys (in this case ticker symbols) that need to be tracked.



Because you are using Spark SQL, you can execute aggregate queries at scale and in real time.

```
%sql
SELECT ticker, AVG(close) as Average_Close
FROM priceWithFundamentals
GROUP BY ticker
ORDER BY Average_Close
```

In closing, we demonstrated how to simplify streaming stock data analysis using Delta Lake. By combining Spark Structured Streaming and Delta Lake, we can use the Databricks integrated workspace to create a performant, scalable solution that has the advantages of both data lakes and data warehouses.



The Databricks Unified Data Platform removes the data engineering complexities commonly associated with streaming and transactional consistency, enabling data engineering and data science teams to focus on understanding the trends in their stock data.

How Tilting Point Does Streaming Ingestion Into Delta Lake

Tilting Point is a new-generation games partner that provides top development studios with expert resources, services and operational support to optimize high-quality live games for success. Through its user acquisition fund and its world-class technology platform, Tilting Point funds and runs performance marketing management and live games operations to help developers achieve profitable scale.

By leveraging Delta Lake, Tilting Point is able to leverage quality data and make it readily available for analytics to improve the business. Diego Link, VP of Engineering at Tilting Point, provided insights for this use case.

The team at Tilting Point was running daily and hourly batch jobs for reporting on game analytics. They wanted to make their reporting near real-time, getting insights within 5–10 minutes.

They also wanted to make their in-game LiveOps decisions based on real-time player behavior for giving real-time data to a bundles-and-offer system, provide up-to-the-minute alerting on LiveOPs changes that actually might have unforeseen detrimental effects and even alert on service interruptions in game operations. The goal was to ensure that the game experience was as robust as possible for their players.

Additionally, they had to store encrypted Personally Identifiable Information (PII) data separately in order to maintain GDPR compliance.

How data flows and associated challenges

Tilting Point has a proprietary software development kit that developers integrate with to send data from game servers to an ingest server hosted in AWS. This service removes all PII data and then sends the raw data to an Amazon Firehose endpoint. Firehose then dumps the data in JSON format continuously to S3.

To clean up the raw data and make it available quickly for analytics, the team considered pushing the continuous data from Firehose to a message bus (e.g., Kafka, Kinesis) and then using Apache Spark's Structured Streaming to continuously process data and write to Delta Lake tables.

While that architecture sounds ideal for low latency requirements of processing data in seconds, Tilting Point didn't have such low latency needs for their ingestion pipeline. They wanted to make the data available for analytics in a few minutes, not seconds. Hence they decided to simplify our architecture by eliminating a message bus and instead use S3 as a continuous source for their structured streaming job.

But the key challenge in using S3 as a continuous source is identifying files that changed recently.

Listing all files every few minutes has two major issues:

- **Higher latency:** Listing all files in a directory with a large number of files has high overhead and increases processing time.
- **Higher cost:** Listing lots of files every few minutes can quickly add to the S3 cost.

Leveraging Structured Streaming with blob store as source and Delta Lake tables as sink

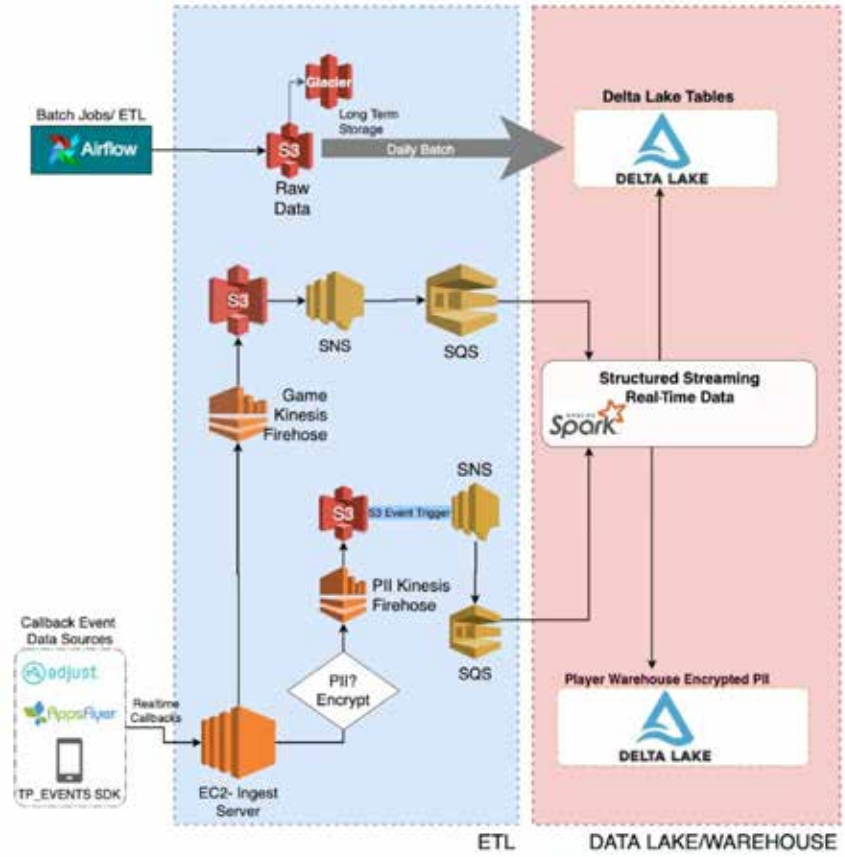
To continuously stream data from cloud blob storage like S3, Tilting Point uses Databricks' S3-SQS source. The S3-SQS source provides an easy way to incrementally stream data from S3 without the need to write any state management code on what files were recently processed.

This is how Tilting Point's ingestion pipeline looks:

- Configure Amazon S3 event notifications to send new file arrival information to SQS via SNS.
- Tilting Point uses the S3-SQS source to read the new data arriving in S3. The S3-SQS source reads the new file names that arrived in S3 from SQS and uses that information to read the actual file contents in S3. An example code below:

```
spark.readStream \
  .format("s3-sqs") \
  .option("fileFormat", "json") \
  .option("queueUrl", ...) \
  .schema(...) \
  .load()
```

- Tilting Point's structured streaming job then cleans up and transforms the data. Based on the game data, the streaming job uses the foreachBatch API of Spark streaming and writes to 30 different Delta Lake tables.
- The streaming job produces lots of small files. This affects performance of downstream consumers. So, an optimize job runs daily to compact small files in the table and store them as right file sizes so that consumers of the data have good performance while reading the data from Delta Lake tables. Tilting Point also runs a weekly optimize job for a second round of compaction.



Architecture showing continuous data ingest into Delta Lake tables

The above Delta Lake ingestion architecture helps in the following ways:

- **Incremental loading:** The S3-SQS source incrementally loads the new files in S3. This helps quickly process the new files without too much overhead in listing files.
- **No explicit file state management:** There is no explicit file state management needed to look for recent files.
- **Lower operational burden:** Since we use S3 as a checkpoint between Firehose and Structured Streaming jobs, the operational burden to stop streams and re-process data is relatively low.
- **Reliable ingestion:** Delta Lake uses optimistic concurrency control to offer ACID transactional guarantees. This helps with reliable data ingestion.
- **File compaction:** One of the major problems with streaming ingestion is tables ending up with a large number of small files that can affect read performance. Before Delta Lake, we had to set up a different table to write the compacted data. With Delta Lake, thanks to ACID transactions, we can compact the files and rewrite the data back to the same table safely.
- **Snapshot isolation:** Delta Lake's snapshot isolation allows us to expose the ingestion tables to downstream consumers while data is being appended by a streaming job and modified during compaction.
- **Rollbacks:** In case of bad writes, Delta Lake's Time Travel helps us roll back to a previous version of the table.

In this section, we walked through Tilting Point's use cases and how they do streaming ingestion using Databricks' S3-SQS source into Delta Lake tables efficiently without too much operational overhead to make good quality data readily available for analytics.

USE CASE #3

Building a Quality of Service Analytics Solution for Streaming Video Services

As traditional pay TV continues to stagnate, content owners have embraced direct-to-consumer (D2C) subscription and ad-supported streaming for monetizing their libraries of content. For companies whose entire business model revolved around producing great content, which they then licensed to distributors, the shift to now owning the entire glass-to-glass experience has required new capabilities, such as building media supply chains for content delivery to consumers, supporting apps for a myriad of devices and operating systems, and performing customer relationship functions like billing and customer service.

With most services renewing on a monthly basis, subscription service operators need to prove value to their subscribers at all times. General quality of streaming video issues (encompassing buffering, latency, pixelation, jitter, packet loss and the blank screen) have significant business impacts, whether it's increased subscriber churn or decreased video engagement.

When you start streaming, you realize there are so many places where breaks can happen and the viewer experience can suffer. There may be an issue at the source in the servers on-premises or in the cloud; in transit at either the CDN level or ISP level or the viewer's home network; or at the playout level with player/client issues. What breaks at n x 104 concurrent streamers is different from what breaks at n x 105 or n x 106. There is no pre-release testing that can quite replicate real-world users and their ability to push even the most redundant systems to their breaking point as they

channel surf, click in and out of the app, sign on from different devices simultaneously and so on. And because of the nature of TV, things will go wrong during the most important, high-profile events drawing the largest audiences. If you start receiving complaints on social media, how can you tell if they are unique to that one user or rather regional or a national issue? If national, is it across all devices or only certain types (e.g., possibly the OEM updated the OS on an older device type, which ended up causing compatibility issues with the client)?

Identifying, remediating and preventing viewer quality of experience issues becomes a big data problem when you consider the number of users, the number of actions they are taking and the number of handoffs in the experience (servers to CDN to ISP to home network to client). Quality of Service (QoS) helps make sense of these streams of data so you can understand what is going wrong, where and why. Eventually you can get into predictive analytics around what could go wrong and how to remediate it before anything breaks.

Databricks Quality of Service solution overview

The aim of this solution is to provide the core for any streaming video platform that wants to improve their QoS system. It is based on the AWS Streaming Media Analytics Solution provided by AWS Labs, which we then built on top of to add Databricks as a Unified Data Analytics Platform for both the real-time insights and the advanced analytics capabilities.

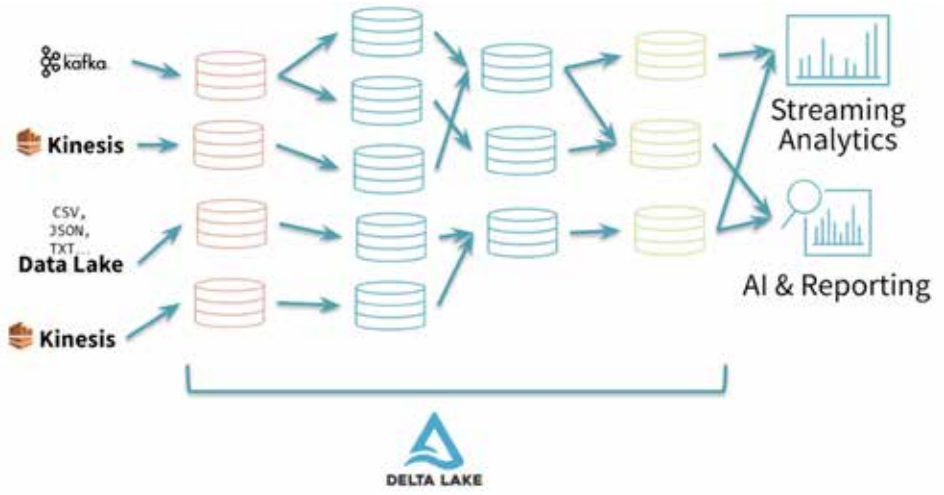
By using Databricks, streaming platforms can get faster insights by always leveraging the most complete and recent data sets powered by robust and reliable data pipelines. This decreases time to market for new features by accelerating data science using a collaborative environment. It provides support for managing the end-to-end machine learning lifecycle and reduces operational costs across all cycles of software development by having a unified platform for both data engineering and data science.

Video QoS solution architecture

With complexities like low-latency monitoring alerts and highly scalable infrastructure required for peak video traffic hours, the straightforward architectural choice was the Delta Architecture — both standard big data architectures like Lambda and Kappa Architectures have disadvantages around the operational effort required to maintain multiple types of pipelines (streaming and batch) and lack support for a unified data engineering and data science approach.

The Delta Architecture is the next-generation paradigm that enables all the data personas in your organization to be more productive:

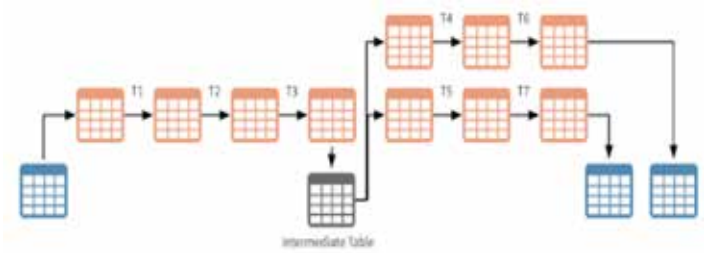
- Data engineers can develop data pipelines in a cost-efficient manner continuously without having to choose between batch and streaming
- Data analysts can get near real-time insights and faster answers to their BI queries
- Data scientists can develop better machine learning models using more reliable data sets with support for time travel that facilitates reproducible experiments and reports



Delta Architecture using the "multi-hop" approach for data pipelines

Writing data pipelines using the Delta Architecture follows the best practices of having a multi-layer "multi-hop" approach where we progressively add structure to data: "Bronze" tables or Ingestion tables are usually raw data sets in the native format (JSON, CSV or txt), "Silver" tables represent cleaned/transformed data sets ready for reporting or data science, and "Gold" tables are the final presentation layer.

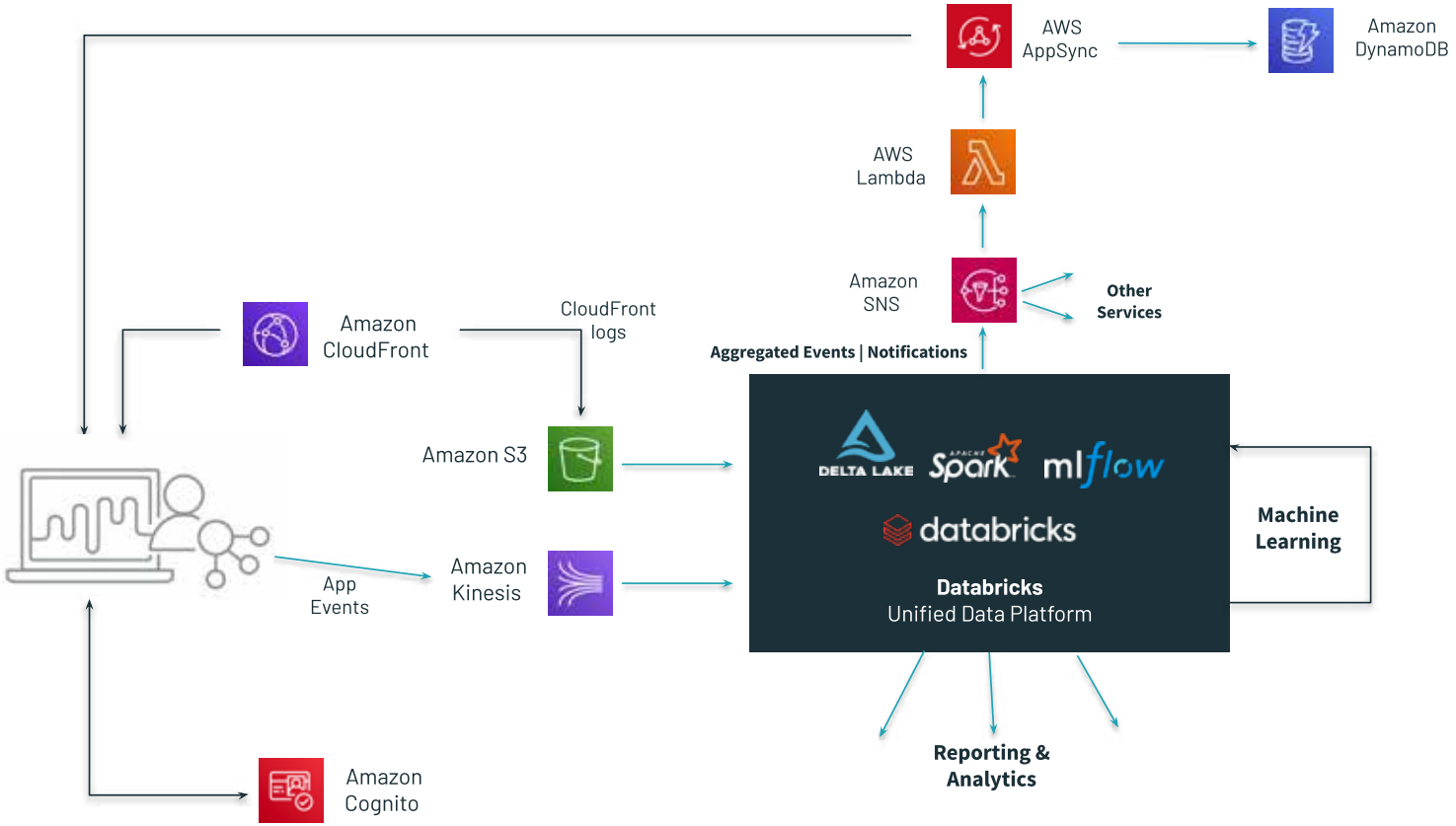
For the pure streaming use cases, the option of materializing the DataFrames in intermediate Delta Lake tables is basically just a trade-off between latency/SLAs and cost (an example being real-time monitoring alerts vs. updates of the recommender system based on new content).



A streaming architecture can still be achieved while materializing DataFrames in Delta Lake tables

The number of "hops" in this approach is directly impacted by the number of consumers downstream, complexity of the aggregations (e.g., Structured Streaming enforces certain limitations around chaining multiple aggregations) and the maximization of operational efficiency.

The QoS solution architecture is focused around best practices for data processing and is not a full video-on-demand (VoD) solution — with some standard components like the "front door" service Amazon API Gateway being avoided from the high-level architecture in order to keep the focus on data and analytics.



High-level architecture for the QoS platform

Making your data ready for analytics

Both sources of data included in the QoS solution (application events and CDN logs) are using the JSON format, great for data exchange — allowing you to represent complex nested structures, but not scalable and difficult to maintain as a storage format for your data lake / analytics system.

In order to make the data directly queryable across the entire organization, the Bronze to Silver pipeline (the "make your data available to everyone" pipeline) should transform any raw formats into Delta Lake and include all the quality checks or data masking required by any regulatory agencies.

type	ts	jsonData
stream	2020-04-08T09:53:58.864+0000	{"metrictype":"stream","at":105.077308,"rtt":350,"connection_type":"4g","package":"hls","resolution":"640x360","fps":"29.970","avg_bitrate":1330203,"duration":597,"cdn_tracking_id":"pxiujwunmzfiqxa8soettjrjvznfwsovtnh8epu0-6azzy88avi1vg==","user_id":"us-west-2:610f9e88-55e8-4a80-9b83-936446e6df2d","video_id":"bigbuckbunny","playlist_type":"live","timestamp":1586339838864}
stream	2020-04-08T09:54:00.396+0000	{"metrictype":"stream","at":415.111827,"rtt":526,"connection_type":"3g","package":"hls","resolution":"640x360","fps":"29.970","avg_bitrate":1350743,"duration":735,"cdn_tracking_id":"jahzaxapw4y-xhrjoescatcytahbjvduwfgcvawqyihu4pagnrsya==","user_id":"us-west-2:cc7af3ef-6cf9-4da1-8274-3212db46be48","video_id":"tearsofsteel","playlist_type":"live","timestamp":1586339640396}
stream	2020-04-08T09:54:01.332+0000	{"metrictype":"stream","at":410.095889,"rtt":591,"connection_type":"4g","package":"hls","resolution":"640x360","fps":"29.970","avg_bitrate":1330203,"duration":597,"cdn_tracking_id":"svymfjqsrpmuy-zewwi4u3pbdn2sfgcsodm-2cgdpfihhy4zhka==","user_id":"us-west-2:83cb8ebe-4f29-4622-80a2-feb10095a9a2","video_id":"bigbuckbunny","playlist_type":"live","timestamp":1586339641332}
stream	2020-04-08T09:53:53.498+0000	{"metrictype":"stream","at":385.057842741,"rtt":523,"connection_type":"not available","package":"hls","resolution":"640x360","fps":"29.970","avg_bitrate":1350743,"duration":735,"cdn_tracking_id":"h3dpitndfpfy9acagudbie3kwck8gkrih98fjdu7uqhikigwagc8ga==","user_id":"us-west-2:2f7000b3-1e01-4cba-a0ef-d49b8952833c","video_id":"tearsofsteel","playlist_type":"live","timestamp":1586339833498}
buffer	2020-04-	{"metrictype":"buffer","buffer_type":"firstbuffer","at":385.057842741,"rtt":523,"connection_type":"not available","time_millisecond":386983,"cdn_tracking_id":"h3dpitndfpfy9acagudbie3kwck8gkrih98fjdu7uqhikigwagc8ga==","user_id":"us-west-2:2f7000b3-1e01-4cba-

Showing the first 1000 rows.

Raw format of the app events

Video applications events

Based on the architecture, the video application events are pushed directly to Kinesis Streams and then just ingested to a Delta Lake append-only table without any changes to the schema.

Using this pattern allows a high number of consumers downstream to process the data in a streaming paradigm without having to scale the throughput of the Kinesis stream. As a side effect of using a Delta Lake table as a sink (which supports optimize!), we don't have to worry about the way the size of the processing window will impact the number of files in your target table — known as the "small files" issue in the big data world.

Both the timestamp and the type of message are being extracted from the JSON event in order to be able to partition the data and allow consumers to choose the type of events they want to process. Again combining a single Kinesis stream for the events with a Delta Lake "Events" table reduces the operational complexity while making things easier for scaling during peak hours.

Schema:

col_name	data_type
browserfamily	string
bytes	string
cdn_source	string
isbot	boolean
origin	string
location	string
logdate	date
logtime	string
osfamily	string
requestid	string
ip	string
resulttype	string
year	int
month	int
day	int
hour	int

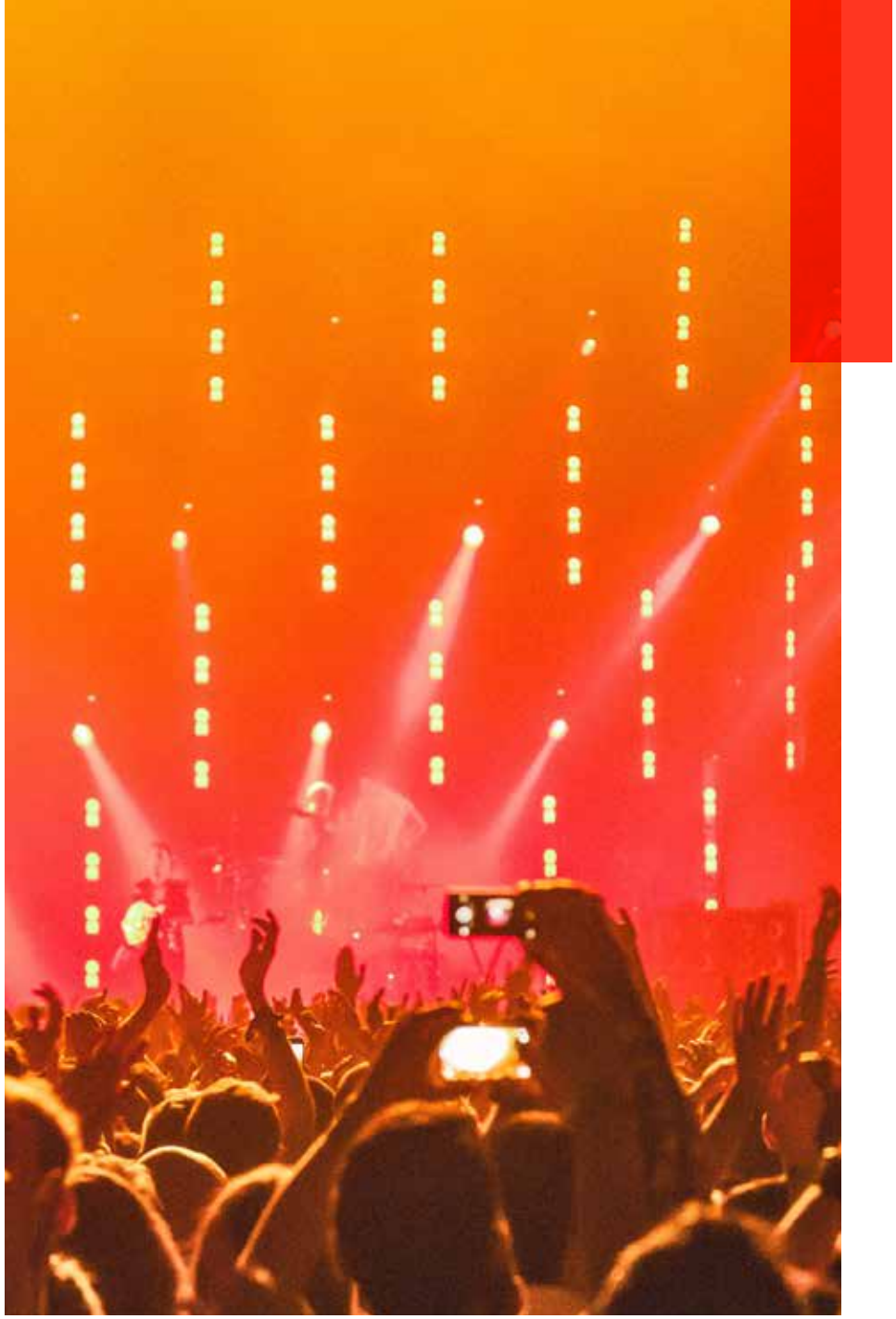
All the details are extracted from JSON for the Silver table

CDN logs

The CDN logs are delivered to S3, so the easiest way to process them is the Databricks Auto Loader, which incrementally and efficiently processes new data files as they arrive in S3 without any additional setup.

```
    auto_loader_df = spark.readStream.format("cloudFiles") \
        .option("cloudFiles.format", "json") \
        .option("cloudFiles.region", region) \
        .load(input_location)

     anonymized_df = auto_loader_df.select('*', ip_
anonymizer('requestip').alias('ip'))\
        .drop('requestip')\
        .withColumn("origin", map_ip_to_location(col('ip')))


    anonymized_df.writeStream \
        .option('checkpointLocation', checkpoint_location)\
        .format('delta') \
        .table(silver_database + '.cdn_logs')
```

As the logs contain IPs — considered personal data under the GDPR regulations — the "make your data available to everyone" pipeline has to include an anonymization step. Different techniques can be used, but we decided to just strip the last octet from IPv4 and the last 80 bits from IPv6. On top, the data set is also enriched with information around the origin country and the ISP provider, which will be used later in the Network Operation Centers for localization.



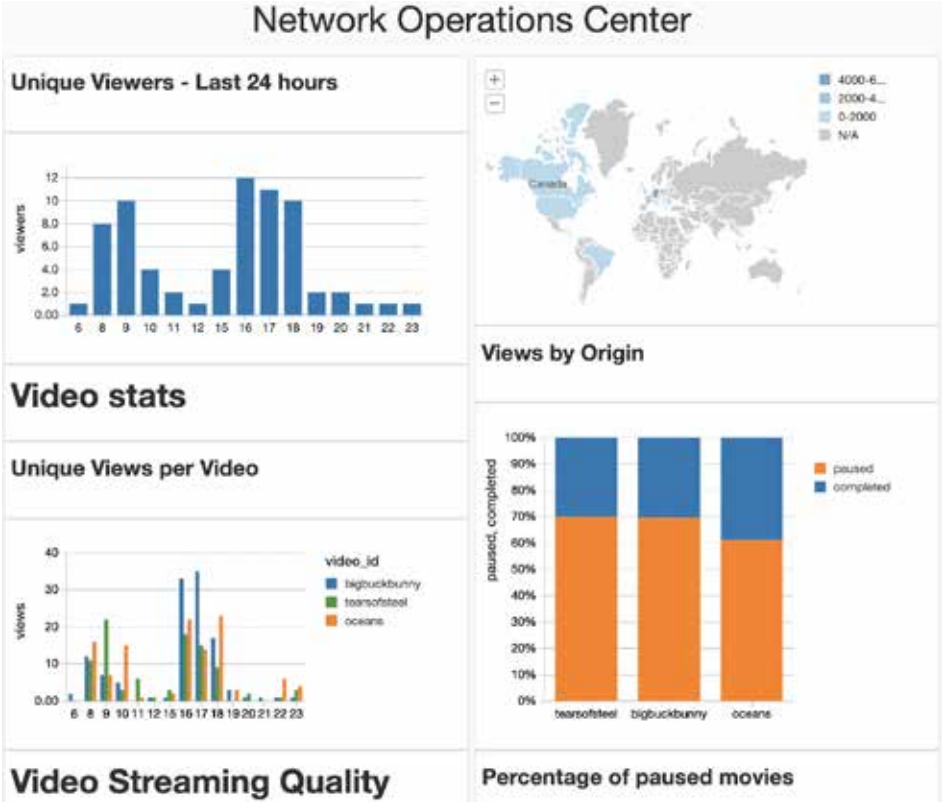
Creating the Dashboard / Virtual Network Operation Centers

Streaming companies need to monitor network performance and the user experience as near real-time as possible, tracking down to the individual level with the ability to abstract at the segment level, easily defining new segments such as those defined by geos, devices, networks and/or current and historical viewing behavior.

For streaming companies that has meant adopting the concept of Network Operation Centers (NOC) from telco networks for monitoring the health of the streaming experience for their users at a macro level, flagging and responding to any issues early on. At their most basic, NOCs should have dashboards that compare the current experience for users against a performance baseline so that the product teams can quickly and easily identify and attend to any service anomalies.

In the QoS solution we have incorporated a Databricks dashboard. BI tools can also be effortlessly connected in order to build more complex visualizations, but based on customer feedback, built-in dashboards are, most of the time, the fastest way to present the insights to business users.

The aggregated tables for the NOC will basically be the Gold layer of our Delta Architecture — a combination of CDN logs and the application events.

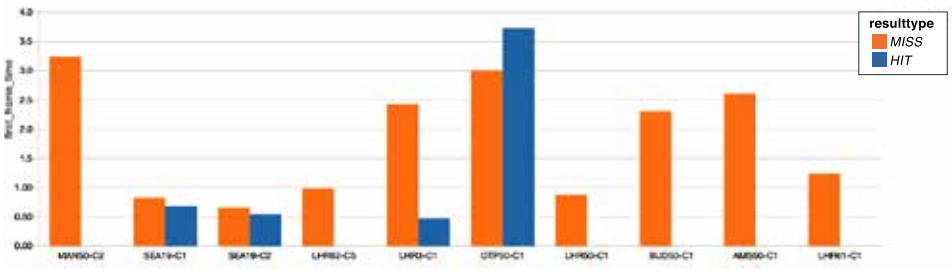


Example of Network Operations Center dashboard

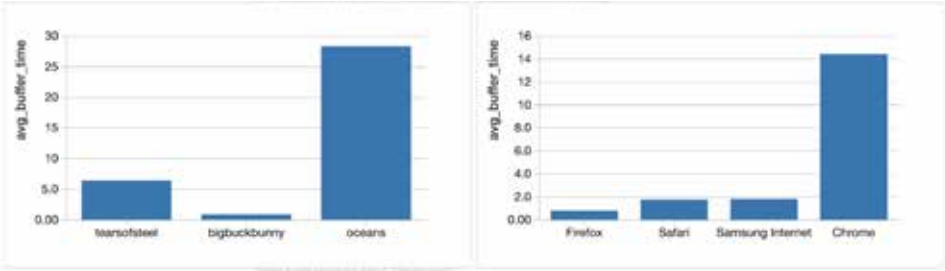
The dashboard is just a way to visually package the results of SQL queries or Python / R transformation — each notebook supports multiple dashboards so in case of multiple end users with different requirements we don't have to duplicate the code — as a bonus the refresh can also be scheduled as a Databricks job.

Visualization of the results of a SQL query

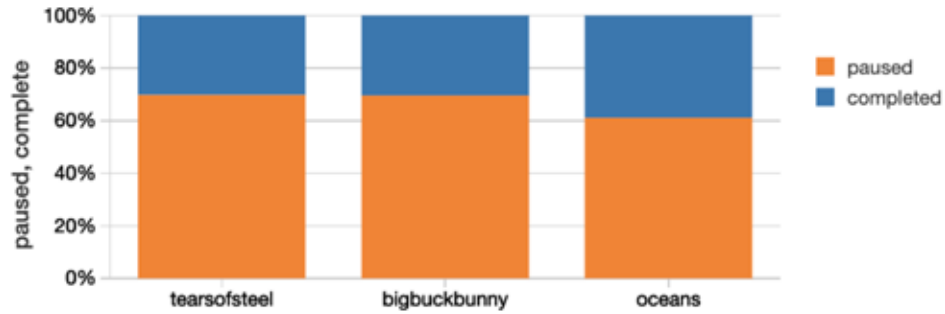
Loading time for videos (time to first frame) allows better understanding of the performance for individual locations of your CDN — in this case the AWS CloudFront Edge nodes — which has a direct impact in your strategy for improving this KPI — either by spreading the user traffic over multi-CDNs or maybe just implementing a dynamic origin selection in case of AWS CloudFront using Lambda@Edge.

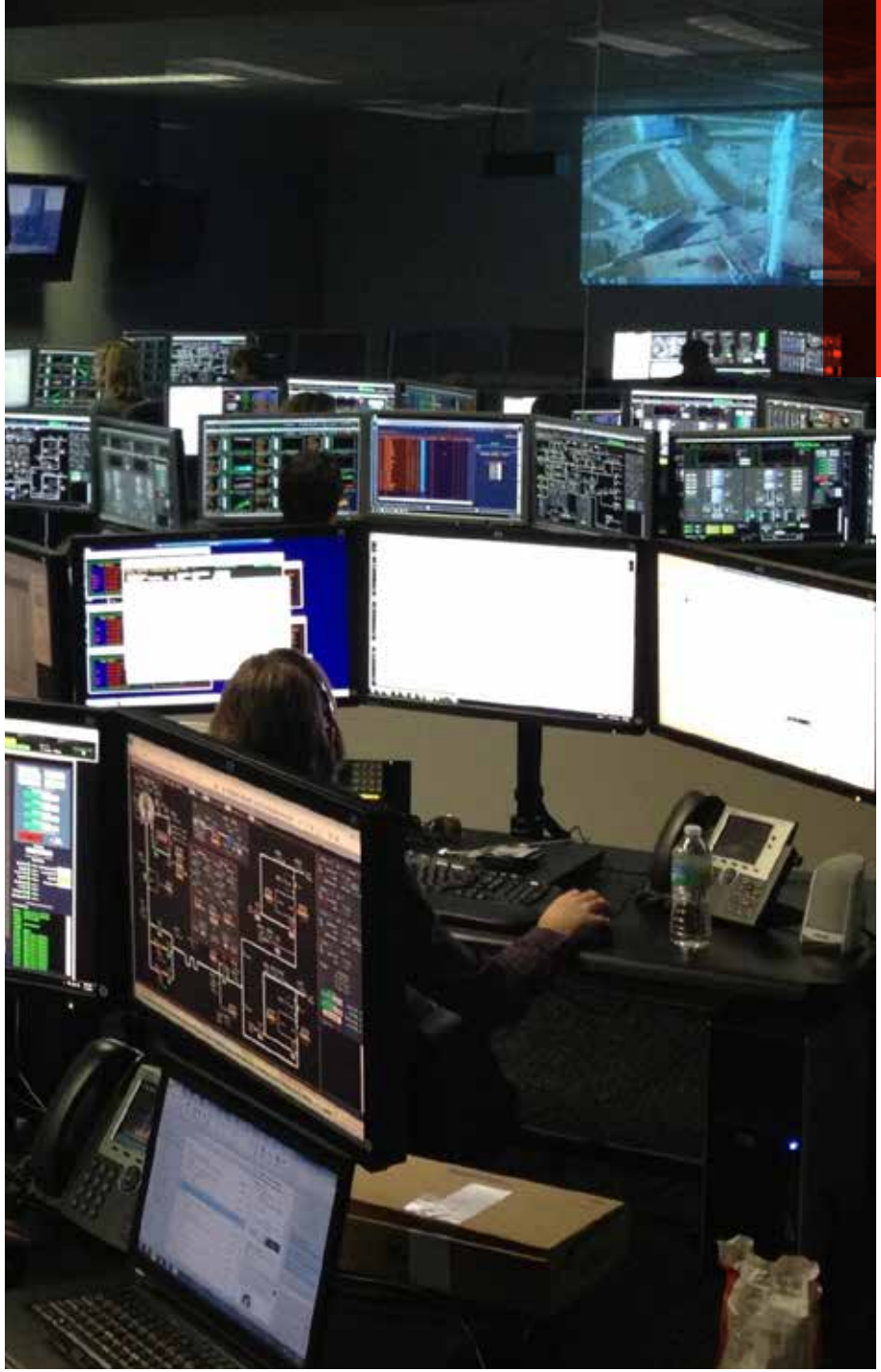


Failure to understand the reasons for high levels of buffering — and the poor video quality experience that it brings — has a significant impact on subscriber churn rate. On top of that, advertisers are not willing to spend money on ads responsible for reducing the viewer engagement — as they add extra buffering on top, so the profits on the advertising business usually are impacted too. In this context, collecting as much information as possible from the application side is crucial to allow the analysis to be done not only at video level but also browser or even type / version of application.



On the content side, events for the application can provide useful information about user behavior and overall quality of experience. How many people that paused a video have actually finished watching that episode / video? What caused the stoppage: The quality of the content or delivery issues? Of course, further analyses can be done by linking all the sources together (user behavior, performance of CDNs /ISPs) to not only create a user profile but also to forecast churn.



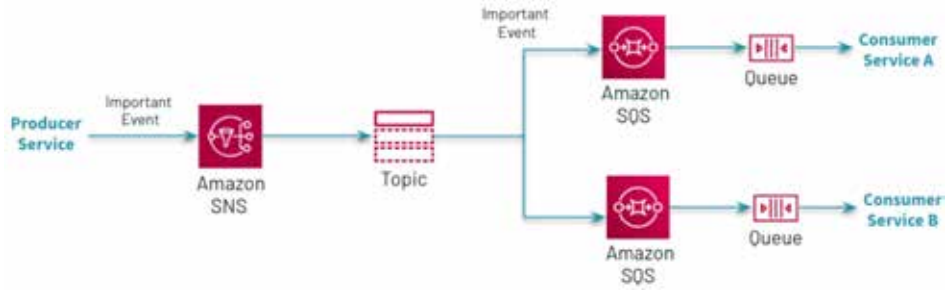


Creating (near) real-time alerts

When dealing with the velocity, volume and variety of data generated in video streaming from millions of concurrent users, dashboard complexity can make it harder for human operators in the NOC to focus on the most important data at the moment and zero-in on root cause issues. With this solution, you can easily set up automated alerts when performance crosses certain thresholds that can help the human operators of the network as well as set off automatic remediation protocols via a Lambda function. For example:

- If a CDN is having latency much higher than baseline (e.g., if it's more than 10% latency vs. baseline average), initiate automatic CDN traffic shifts.
- If more than [some threshold, e.g., 5%] of clients report playback errors, alert the product team that there is likely a client issue for a specific device.
- If viewers on a certain ISP are having higher-than-average buffering and pixelation issues, alert frontline customer representatives on responses and ways to decrease issues (e.g., set stream quality lower).

From a technical perspective, generating real-time alerts requires a streaming engine capable of processing data real time and publish-subscribe service to push notifications.



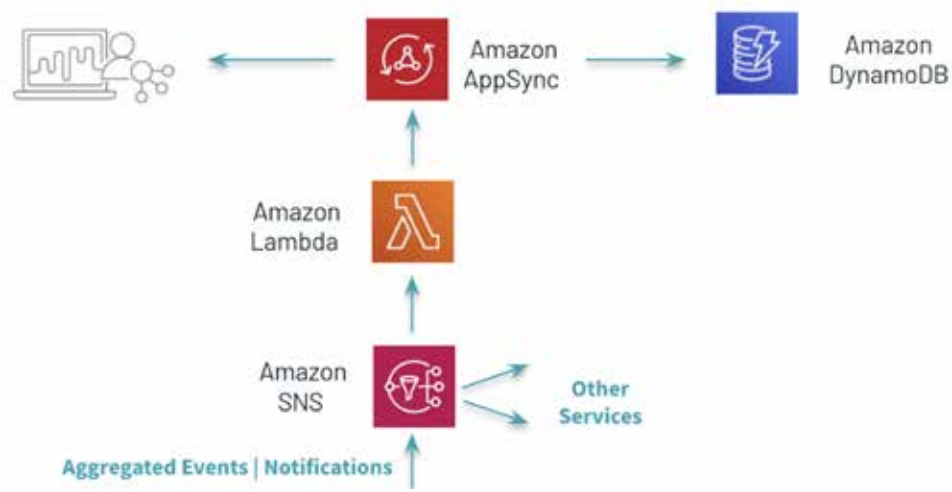
Integrating microservices using Amazon SNS and Amazon SQS

The QoS solution implements the AWS best practices for integrating microservices by using Amazon SNS and its integrations with Amazon Lambda (see below for the updates of web applications) or Amazon SQS for other consumers. The custom for each writer option makes the writing of a pipeline to send email notifications based on a rule-based engine (e.g., validating the percentage of errors for each individual type of app over a period of time) really straightforward.

```
   def send_error_notification(row):

        sns_client = boto3.client('sns', region)

        error_message = 'Number of errors for the App has exceeded the
threshold {}'.format(row['percentage'])

        response = sns_client.publish(
            TopicArn=,
            Message= error_message,
            Subject=,
            MessageStructure='string')

        # Structured Streaming Job

        getKinesisStream("player_events")\
            .selectExpr("type", "app_type")\
            .groupBy("app_type")\
            .apply(calculate_error_percentage)\
            .where("percentage > {}".format(threshold)) \
            .writeStream\
            .foreach(send_error_notification)\
            .start()
```

Sending email notifications using AWS SNS

On top of the basic email use case, the Demo Player includes three widgets updated in real time using AWS AppSync: the number of active users, the most popular videos and the number of users concurrently watching a video.



Updating the application with the results of real-time aggregations

The QoS solution is applying a similar approach — Structured Streaming and Amazon SNS — to update all the values allowing for extra consumers to be plugged in using AWS SQS. This is a common pattern when huge volumes of events have to be enhanced and analyzed; pre-aggregate data once and allow each service (consumer) to make their own decision downstream.

Next steps: machine learning

Manually making sense of the historical data is important but is also very slow. If we want to be able to make automated decisions in the future, we have to integrate machine learning algorithms.

As a Unified Data Platform, Databricks empowers data scientists to build better data science products using features like Runtime for Machine Learning with built-in support for Hyperopt / Horvod / AutoML or the integration with MLflow, the end-to-end machine learning lifecycle management tool.

We have already explored a few important use cases across our customer base while focusing on the possible extensions to the QoS solution.

Point-of-failure prediction and remediation

As D2C streamers reach more users, the costs of even momentary loss of service increases. ML can help operators move from reporting to prevention by forecasting where issues could come up and remediating before anything goes wrong (e.g., a spike in concurrent viewers leads to switching CDNs to one with more capacity automatically).

Customer churn

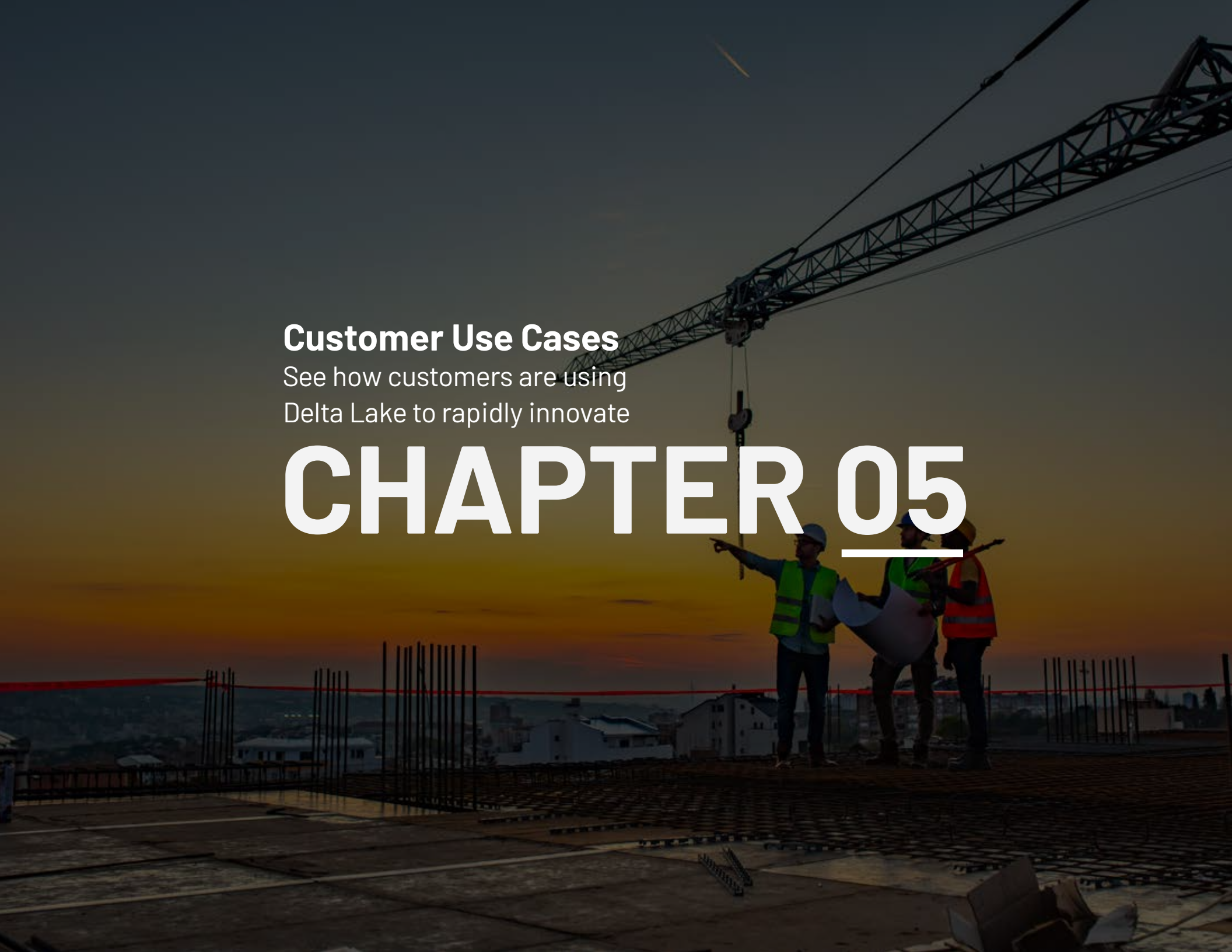
Critical to growing subscription services is keeping the subscribers you have. By understanding the quality of service at the individual level, you can add QoS as a variable in churn and customer lifetime value models. Additionally, you can create customer cohorts for those who have had video quality issues in order to test proactive messaging and save offers.

Getting started with the Databricks streaming video QoS solution

Providing consistent quality in the streaming video experience is table stakes at this point to keep fickle audiences with ample entertainment options on your platform. With this solution we have sought to create a quick start for most streaming video platform environments to embed this QoS real-time streaming analytics solution in a way that:

1. Scales to any audience size
2. Quickly flags quality performance issues at key parts of the distribution workflow
3. Is flexible and modular enough to easily customize for your audience and your needs, such as creating new automated alerts or enabling data scientists to test and roll out predictive analytics and machine learning

To get started, download the notebooks for the Databricks streaming video QoS solution. For more guidance on how to unify batch and streaming data into a single system, view the Delta Architecture webinar.



Customer Use Cases

See how customers are using
Delta Lake to rapidly innovate

CHAPTER 05

USE CASE #1

Healthdirect Australia

Provides Personalized and Secure Online Patient Care With Databricks

As the shepherds of the National Health Services Directory (NHSD), Healthdirect is focused on leveraging terabytes of data covering time-driven, activity-based healthcare transactions to improve health care services and support. With governance requirements, siloed teams and a legacy system that was difficult to scale, they moved to Databricks. This boosted data processing for downstream machine learning while improving data security to meet HIPAA requirements.

Spotlight on Healthdirect

Industry: Healthcare and life sciences

6x

Improvement in data processing

20M

Records ingested in minutes

Data quality and governance issues, silos, and the inability to scale

Due to regulatory pressures, Healthdirect Australia set forth to improve overall data quality and ensure a level of governance on top of that, but they ran into challenges when it came to data storage and access. On top of that, data silos were blocking the team from efficiently preparing data for downstream analytics. These disjointed data

sources impacted the consistency of data reads, as data was oftentimes out-of-sync between the various systems in their stack. The low-quality data also led to higher error rates and processing inefficiencies. This fragmented architecture created significant operational overhead and limited their ability to have a comprehensive view of the patient.

Further, they needed to ingest over 1 billion data points due to a changing landscape of customer demand such as bookings, appointments, pricing, eHealth transaction activity, etc. — estimated at over 1TB of data.

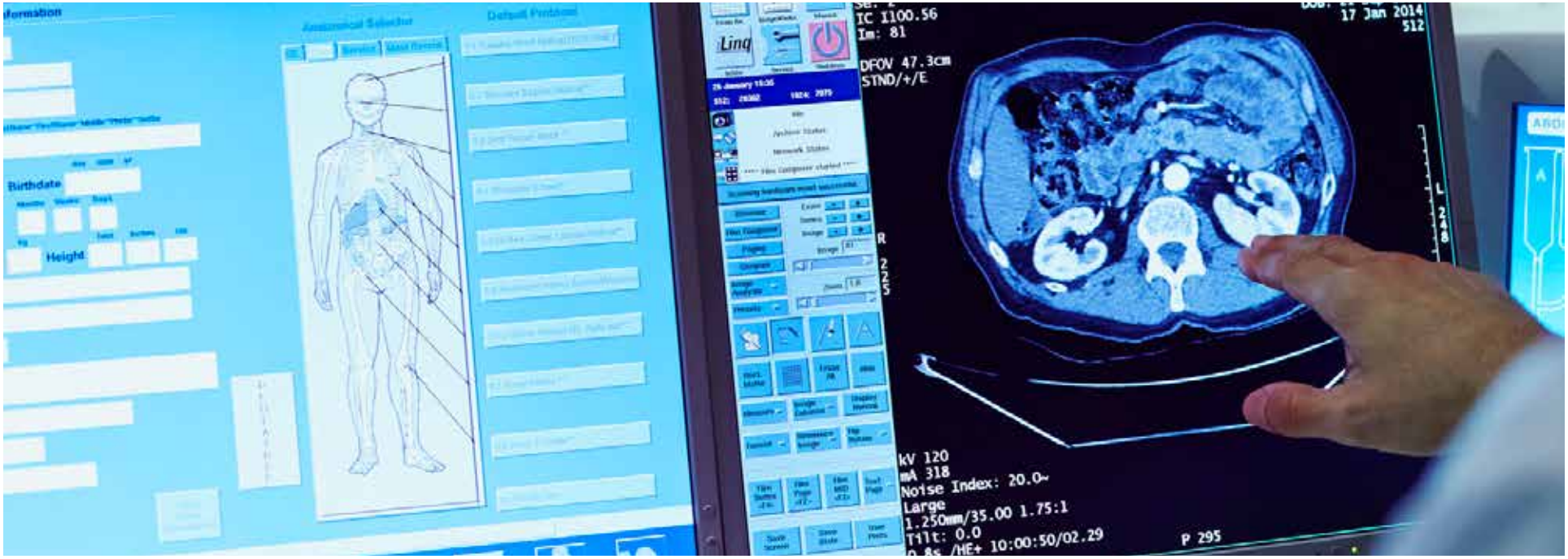
"We had a lot of data challenges. We just couldn't process efficiently enough. We were starting to get batch overruns. We were starting to see that a 24-hour window isn't the most optimum time in which we want to be able to deliver healthcare data and services," explained Peter James, Chief Architect at Healthdirect Australia.

Ultimately, Healthdirect realized they needed to modernize their end-to-end process and tech stack to properly support the business.

Modernizing analytics with Databricks and Delta Lake

Databricks provides Healthdirect Australia with a Unified Data Platform that simplifies data engineering and accelerates data science innovation. The notebook environment enables them to make content changes in a controlled fashion rather than having to run bespoke jobs each time.

"Databricks has provided a big uplift for our teams and our data operations," said James. "The analysts were working directly with the data operations teams. They are able to achieve the same pieces of work together within the same time frames that used to take twice as long. They're working together, and we're seeing just a massive acceleration in the speed at which we can deliver service."



With Delta Lake, they've created logical data zones: Landing, Raw, Staging and Gold. Within these zones, they store their data "as is," in their structured or unstructured state, in Delta Lake tables. From there, they use a metadata-driven schema and hold the data within a nested structure within that table. What this allows them to do is handle data consistently from every source and simplifies the mapping of data to the various applications pulling the data.

Meanwhile, through Structured Streaming, they were able to convert all of their ETL batch jobs into streaming ETL jobs that could serve multiple applications consistently. Overall, the advent of Spark Structured Streaming, Delta Lake and the Databricks Unified Data Platform provides significant architectural improvements that have boosted performance, reduced operational overheads and increased process efficiencies.

Faster data pipelines result in better patient-driven healthcare

As a result of the performance gains delivered by Databricks and the improved data reliability through Delta Lake, Healthdirect Australia realized improved accuracy of their fuzzy name match algorithm from less than 80% with manual verification to 95% and no manual intervention.

The processing improvements with Delta Lake and Structured Streaming allowed them to process more than 30,000 automated updates per month. Prior to Databricks, they had to use unreliable batch jobs that were highly manual to process the same number of updates over a span of 6 months — a 6x improvement in data processing.

> "Databricks delivered the time to market as well as the analytics and operational uplift that we needed in order to be able to meet the new demands of the healthcare sector."
>
> – Peter James, Chief Architect, Healthdirect Australia

They were also able to increase their data load rate to 1 million records per minute, loading their entire 20 million record data set in 20 minutes. Before the adoption of Databricks, this used to take more than 24 hours to process the same 1 million transactions, blocking analysts from making swift decisions to drive results.

Last, data security, which was critical to meet compliance requirements, was greatly improved. Databricks provides standard security accreditations like HIPAA, and Healthdirect was able to use Databricks to meet Australia's security requirements. This yielded significant cost reductions and gave them continuous data assurance by monitoring changes to access privileges like changes in roles, metadata-level security changes, data leakage, etc.

"Databricks delivered the time to market as well as the analytics and operational uplift that we needed in order to be able to meet the new demands of the healthcare sector," said James.

With the help of Databricks, they have proven the value of data and analytics and how it can impact their business vision. With transparent access to data that boasts well-documented lineage and quality, participation across various business and analyst groups has increased — empowering teams to collaborate and more easily and quickly extract value from their data with the goal of improving healthcare for everyone.

USE CASE #2

Comcast

Uses Delta Lake and MLflow to Transform the Viewer Experience

Spotlight on Comcast

Industry: Media and entertainment

10x

Reduction in overall compute costs to process data

90%

Reduction in required DevOps resources to manage infrastructure

Reduced

Deployment times from weeks to minutes

As a global technology and media company connecting millions of customers to personalized experiences, Comcast struggled with massive data, fragile data pipelines and poor data science collaboration. With Databricks — leveraging Delta Lake and MLflow — they can build performant data pipelines for petabytes of data and easily manage the lifecycle of hundreds of models to create a highly innovative, unique and award-winning viewer experience using voice recognition and machine learning.

Infrastructure unable to support data and ML needs

Instantly answering a customer's voice request for a particular program while turning billions of individual interactions into actionable insights, strained Comcast's IT infrastructure and data analytics and data science teams. To make matters more complicated, Comcast needed to deploy models to a disjointed and disparate range of environments: cloud, on-premises and even directly to devices in some instances.

- **Massive data:** Billions of events generated by the entertainment system and 20+ million voice remotes, resulting in petabytes of data that need to be sessionized for analysis.
- **Fragile pipelines:** Complicated data pipelines that frequently failed and were hard to recover. Small files were difficult to manage, slowing data ingestion for downstream machine learning.
- **Poor collaboration:** Globally dispersed data scientists working in different scripting languages struggled to share and reuse code.
- **Manage management of ML models:** Developing, training and deploying hundreds of models was highly manual, slow and hard to replicate, making it difficult to scale.
- **Friction between dev and deployment:** Dev teams wanted to use the latest tools and models while ops wanted to deploy on proven infrastructure.

Automated infrastructure, faster data pipelines with Delta Lake

Comcast realized they needed to modernize their entire approach to analytics from data ingest to the deployment of machine learning models to delivering new features that delight their customers. Today, the Databricks Unified Data Platform enables Comcast to build rich data sets and optimize machine learning at scale, streamline workflows across teams, foster collaboration, reduce infrastructure complexity, and deliver superior customer experiences.

- **Simplified infrastructure management:** Reduced operational costs through automated cluster management and cost management features such as autoscaling and spot instances.
- **Performant data pipelines:** Delta Lake is used for the ingest, data enrichment and initial processing of the raw telemetry from video and voice applications and devices.
- **Reliably manage small files:** Delta Lake enabled them to optimize files for rapid and reliable ingestion at scale.
- **Collaborative workspaces:** Interactive notebooks improve cross-team collaboration and data science creativity, allowing Comcast to greatly accelerate model prototyping for faster iteration.
- **Simplified ML lifecycle:** Managed MLflow simplifies the machine learning lifecycle and model serving via the Kubeflow environment, allowing them to track and manage hundreds of models with ease.
- **Reliable ETL at scale:** Delta Lake provides efficient analytics pipelines at scale that can reliably join historic and streaming data for richer insights.

Delivering personalized experiences with ML

In the intensely competitive entertainment industry, there is no time to press the Pause button. Armed with a unified approach to analytics, Comcast can now fast-forward into the future of AI-powered entertainment — keeping viewers engaged and delighted with competition-beating customer experiences.

- **Emmy-winning viewer experience:** Databricks helps enable Comcast to create a highly innovative and award-winning viewer experience with intelligent voice commands that boosts engagement.
- **Reduced compute costs by 10x:** Delta Lake has enabled Comcast to optimize data ingestion, replacing 640 machines with 64 while improving performance. Teams can spend more time on analytics and less time on infrastructure management.
- **Less DevOps:** Reduced the number of DevOps full-time employees required for onboarding 200 users from 5 to 0.5.
- **Higher data science productivity:** Fostered collaboration between global data scientists by enabling different programming languages through a single interactive workspace. Also, Delta Lake has enabled the data team to use data at any point within the data pipeline, allowing them to act more quickly in building and training new models.
- **Faster model deployment:** Reduced deployment times from weeks to minutes as operations teams deployed models on disparate platforms.

USE CASE #3

Banco Hipotecario

Personalizes the Banking Experience With Data and ML

Banco Hipotecario — a leading Argentinian commercial bank — is on a mission to leverage machine learning to deliver new insights and services that will delight customers and create upsell opportunities. With a legacy analytics and data warehousing system that was rigid and complex to scale, they turned to Databricks to unify data science, engineering and analytics.

As a result of this partnership, they were able to significantly increase customer acquisition and cross-sells while lowering the cost for acquisition, greatly impacting overall customer retention and profitability.

Spotlight on Banco Hipotecario

Industry: Financial services

35%

Reduction in cost of acquisition

Technical use cases: Ingest and ETL, machine learning and SQL Analytics

Legacy analytics tools are slow, rigid and impossible to scale

Banco Hipotecario set forth to increase customer acquisition by reducing risk and improving the customer experience. With data analytics and machine learning anchoring their strategy, they hoped to influence a range of use cases from fraud detection and risk analysis to serving product recommendations to drive upsell and cross-sell opportunities and forecast sales.

Banco Hipotecario faced a number of the challenges that often come along with outdated technology and processes: disorganized or inaccurate data; poor cross-team collaboration; the inability to innovate and scale; resource-intensive workflows, — the list goes on.

"In order to execute on our data analytics strategy, new technologies were needed in order to improve data engineering and boost data science productivity," said Daniel Sanchez, Enterprise Data Architect at Banco Hipotecario. "The first steps we took were to move to a cloud-based data lake, which led us to Azure Databricks and Delta Lake."

A unified platform powers the data lake and easy collaboration

Banco Hipotecario turned to Databricks to modernize their data warehouse environment, improve cross-team collaboration, and drive data science innovation. Fully managed in Microsoft Azure, they were able to easily and reliably ingest massive volumes of data, spinning up their whole infrastructure in 90 days. With Databricks' automated cluster management capabilities, they are able to scale clusters on-demand to support large workloads.

Delta Lake has been especially useful in bringing reliability and performance to Banco Hipotecario's data lake environment. With Delta Lake, they are now able to build reliable and performant ETL pipelines like never before.

Meanwhile, performing SQL Analytics on Databricks has helped them do data exploration, cleansing and generate data sets in order to create models, enabling the team to deploy their first model within the first three months, and the second model generated was rolled out in just two weeks.

At the same time, data scientists were finally able to collaborate, thanks to interactive notebooks; this meant faster builds, training and deployment. And MLflow streamlined the ML lifecycle and removed the overreliance on data engineering.

"Databricks gives our data scientists the means to easily create our own experiments and deploy them to production in weeks, rather than months," said Miguel Villalba, Head of Data Engineering and Data Science.

An efficient team maximizes customer acquisition and retention

Since moving to Databricks, the data team at Banco Hipotecario could not be happier, as Databricks has unified them across functions in an integrated fashion.

The results of data unification and markedly improved collaboration and autonomy cannot be overstated. Since deploying Databricks, Banco Hipotecario has increased their cross-sell into new products by a whopping 90%, while machine learning has reduced the cost of customer acquisition by 35%.

USE CASE #4

Viacom18

Migrates From Hadoop to Databricks to Deliver More Engaging Experiences

Viacom18 Media Pvt. Ltd. is one of India's fastest-growing entertainment networks with 40x growth over the past decade. They offer multi-platform, multigenerational and multicultural brand experiences to 600+ million monthly viewers.

In order to deliver more engaging experiences for their millions of viewers, Viacom18 migrated from their Hadoop environment due to its inability to process data at scale efficiently. With Databricks, they have streamlined their infrastructure management, increased data pipeline speeds and increased productivity among their data teams.

Today, Viacom18 is able to deliver more relevant viewing experiences to their subscribers, while identifying opportunities to optimize the business and drive greater ROI.

Spotlight on Viacom18

Industry: Media and entertainment

26%

Increase in operational efficiency lowers overall costs

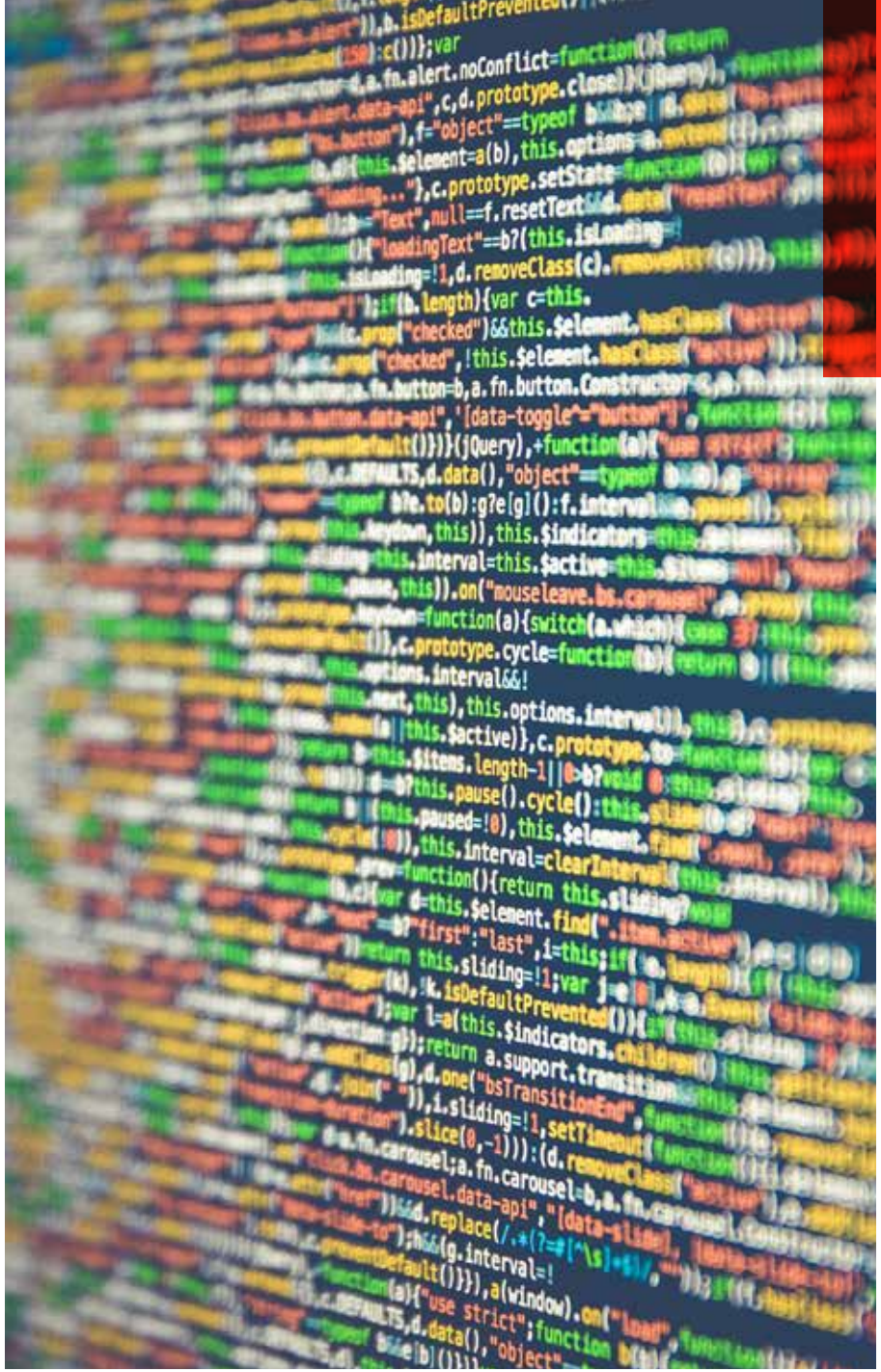
Growth in subscribers and terabytes of viewing data push Hadoop to its limits

Viacom18, a joint venture between Network18 and ViacomCBS, is focused on providing its audiences with highly personalized viewing experiences. The core of this strategy requires implementing an enterprise data architecture that enables the building of powerful customer analytics on daily viewer data. But with millions of consumers across India, the sheer amount of data was tough to wrangle: They were tasked with ingesting and processing over 45,000 hours of daily content on VOOT (Viacom18's on-demand video subscription platform), which easily generated 700GB to 1TB of data per day.

"Content is at the heart of what we do," explained Parijat Dey, Viacom18's Assistant Vice President of Digital Transformation and Technology. "We deliver personalized content recommendations across our audiences around the world based on individual viewing history and preferences in order to increase viewership and customer loyalty."

Viacom18's data lake, which was leveraging on-premises Hadoop for operations, wasn't able to optimally process 90 days of rolling data within their management's defined SLAs, limiting their ability to deliver on their analytics needs, which impacted not only the customer experience but also overall costs.

To meet this challenge head-on, Viacom18 needed a modern data warehouse with the ability to analyze data trends for a longer period of time instead of daily snapshots. They also needed a platform that simplified infrastructure by allowing their team to easily provision clusters with features like auto-scaling to help reduce compute costs.



Rapid data processing for analytics and ML with Databricks

To enable the processing power and data science capabilities they required, Viacom18 partnered with Celebal Technologies, a premier Salesforce, data analytics and big data consulting organization based in India. The team at Celebal leveraged Azure Databricks to provide Viacom18 with a unified data platform that modernizes its data warehousing capabilities and accelerates data processing at scale.

The ability to cache data within Delta Lake resulted in the much-needed acceleration of queries, while cluster management with auto-scaling and the decoupling of storage and compute simplified Viacom18's infrastructure management and optimized operational costs. "Delta Lake has created a streamlined approach to the management of data pipelines," explained Dey. "This has led to a decrease in operational costs while speeding up time-to-insight for downstream analytics and data science."

The notebooks feature was an unexpected bonus for Viacom18, as a common workspace gave data teams a way to collaborate and increase productivity on everything from model training to ad hoc analysis, dashboarding and reporting via PowerBI.

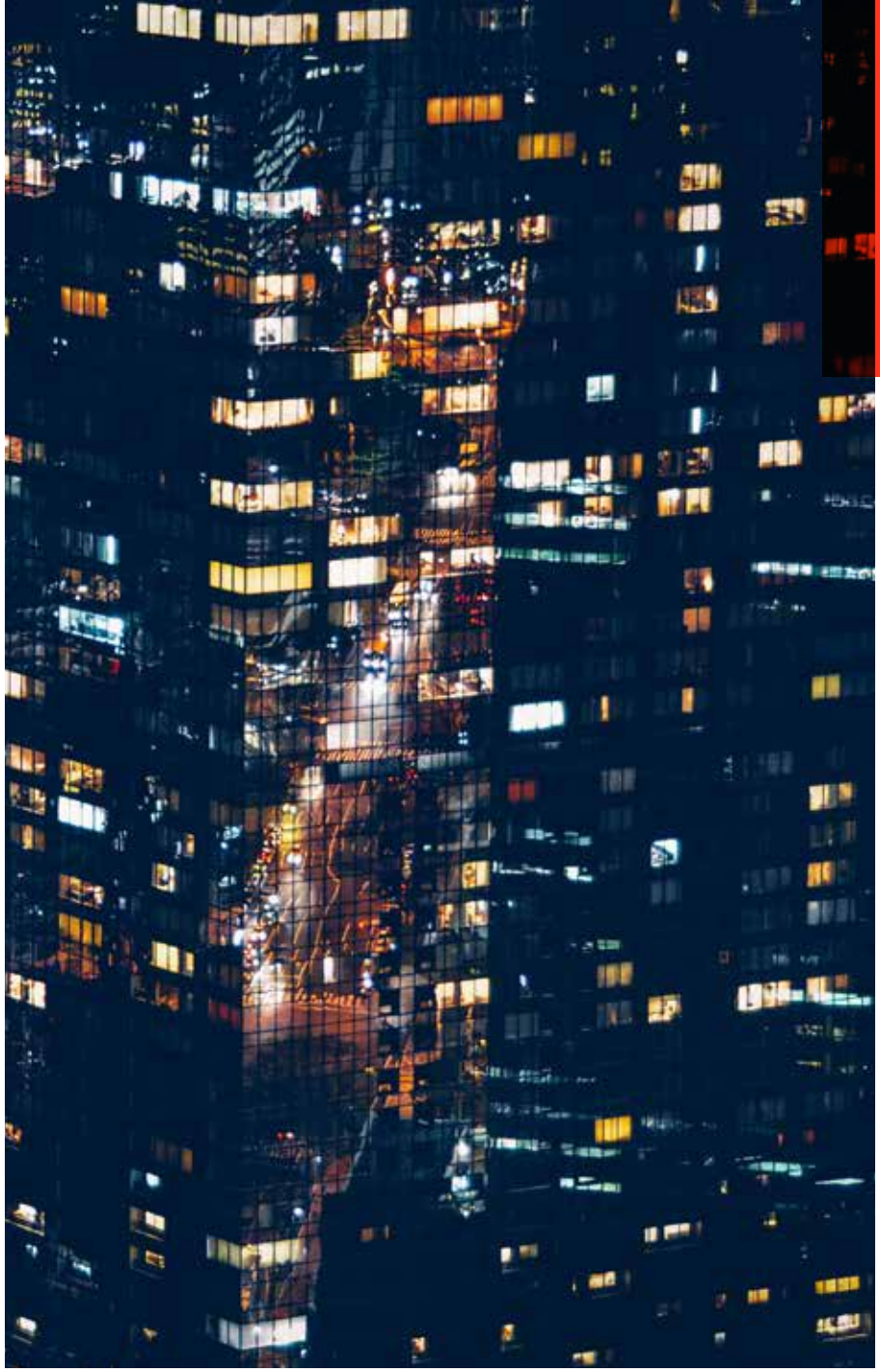
Leveraging viewer data to power personalized viewing experiences

Celebal Technologies and Databricks have enabled Viacom18 to deliver innovative customer solutions and insights with increased cross-team collaboration and productivity. With Databricks, Viacom18's data team is now able to seamlessly navigate their data while better serving their customers.

"With Databricks, Viacom18's engineers can now slice and dice large volumes of data and deliver customer behavioral and engagement insights to the analysts and data scientists," said Dey.

In addition to performance gains, the faster query times have also lowered the overall cost of ownership, even with daily increases in data volumes. "Azure Databricks has greatly streamlined processes and improved productivity by an estimated 26%," concluded Dey.

Overall, Dey cites the migration from Hadoop to Databricks has delivered significant business value — reducing the cost of failure, accelerating processing speeds at scale, and simplifying ad hoc analysis for easier data exploration and innovations that deliver highly engaging customer experiences.



What's next?

Now that you understand Delta Lake, it may be time to take a look at some additional resources.

Do a deep dive into Delta Lake >

- Getting Started With Delta Lake Tech Talk Series
- Diving Into Delta Lake Tech Talk Series
- Visit the site for additional resources

Try Databricks for free >

Learn more >

© Databricks 2021. All rights reserved. Apache, Apache Spark, Spark and the Spark logo are trademarks of the Apache Software Foundation.

databricks