

The Delta Lake Series Fundamentals and Performance

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



What's inside?

The Delta Lake Series of eBooks is published by Databricks to help leaders and practitioners understand the full capabilities of Delta Lake as well as the landscape it resides in. This eBook, **The Delta Lake Series – Fundamentals and Performance**, focuses on the fundamentals of Delta Lake as well as its performance.

What's next?

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



Here's what you'll find inside

Introduction

What is Delta Lake?

Chapter

01

The Fundamentals of Delta Lake: Why Reliability and Performance Matter

Chapter

02

Unpacking the Transaction Log

Chapter

03

How to Use Schema Enforcement and Evolution

Chapter

04

Delta Lake DML Internals

Chapter

05

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



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 life-cycle management to data lakes. Our customers have found that Delta Lake solves for challenges around malformed data ingestion, difficulties with deleting data for compliance or issues with 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.

A person wearing a red and blue plaid shirt is gesturing with their hands, palms facing each other, in a classroom or meeting setting. In the background, there are blurred figures of other people and a laptop on a desk. The overall lighting is dim and blue-toned.

**The Fundamentals of Delta Lake:
Why Reliability and Performance Matter**

CHAPTER 01



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 explicitly 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. 📦

A hand is pointing at a computer screen that displays a transaction log. The screen shows a grid of data with various colored cells (blue, yellow, red). The background is dark and blurry, suggesting a server room or data center environment.

Unpacking the Transaction Log

CHAPTER 02

02 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

Changes to the table are stored as *ordered, atomic* units called commits



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 abil-

ity 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 >](#)

Audit Delta Lake Table History

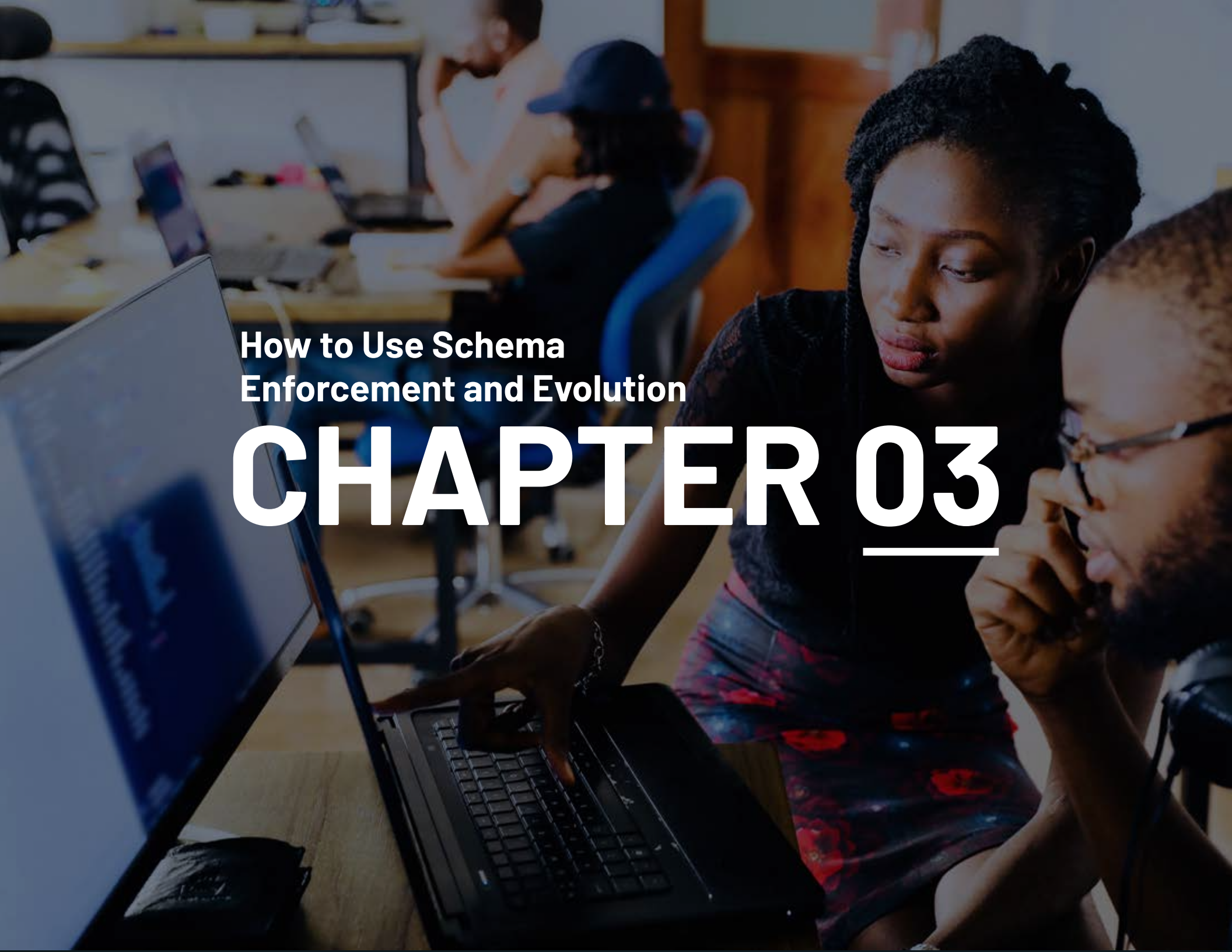
All changes to the Delta Lake table are recorded as commits in the table's transaction log. As you write into a Delta Lake table or directory, every operation is automatically versioned. You can use the DESCRIBE HISTORY command to view the table's history. For more information, check out the [docs](#).

```
%sql DESCRIBE HISTORY loans_delta
```

(1) Spark Jobs

	version	timestamp	userid	userName	operation	operationParameters
1	44	2020-12-28T23:44:35.000+0000	101001	Brenner.Heintz@databricks.com	STREAMING UPDATE	<pre>{ "outputMode": "Append", "queryId": "1ff09fc2-c348-476a-a5ec-f3d7b5b4b696", "epochId": "22" }</pre>
2	43	2020-12-28T23:44:34.000+0000	101001	Brenner.Heintz@databricks.com	STREAMING UPDATE	<pre>{ "outputMode": "Append", "queryId": "1ff09fc2-c348-476a-a5ec-f3d7b5b4b696", "epochId": "22" }</pre>
3	42	2020-12-28T23:44:32.000+0000	101001	Brenner.Heintz@databricks.com	STREAMING UPDATE	<pre>{ "outputMode": "Append", "queryId": "1ff09fc2-c348-476a-a5ec-f3d7b5b4b696", "epochId": "22" }</pre>
4	41	2020-12-28T23:44:30.000+0000	101001	Brenner.Heintz@databricks.com	STREAMING UPDATE	<pre>{ "outputMode": "Append", "queryId": "90165d7c-683b-49c4-b1e1-8d7c7c7c7c7c", "epochId": "22" }</pre>
5	40	2020-12-28T23:44:29.000+0000	101001	Brenner.Heintz@databricks.com	STREAMING UPDATE	<pre>{ "outputMode": "Append", "queryId": "1ff09fc2-c348-476a-a5ec-f3d7b5b4b696", "epochId": "22" }</pre>

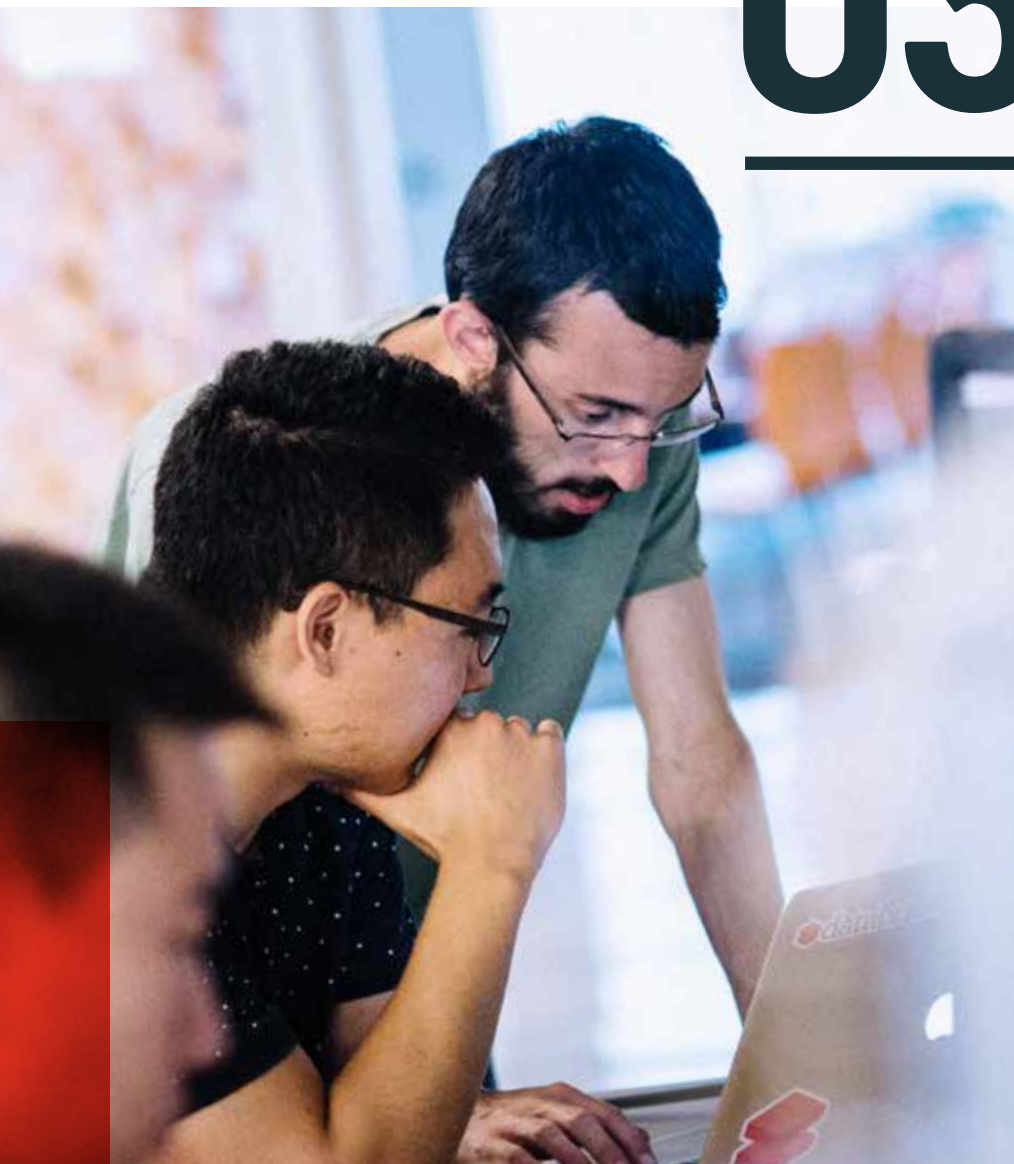
Showing all 49 rows.

A woman with dark hair in braids, wearing a black lace top and a colorful patterned skirt, is pointing at a laptop screen. A man with glasses and a beard is looking at the screen with a thoughtful expression. In the background, another person is working at a desk in a modern office environment.

**How to Use Schema
Enforcement and Evolution**

CHAPTER 03

03 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 example below.

```
#Add the mergeSchema option
loans.write.format("delta") \
.option("mergeSchema", "true") \
.mode("append") \
.save(DELTA_LAKE_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 type (in place)**
- **Renaming column names that differ only 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

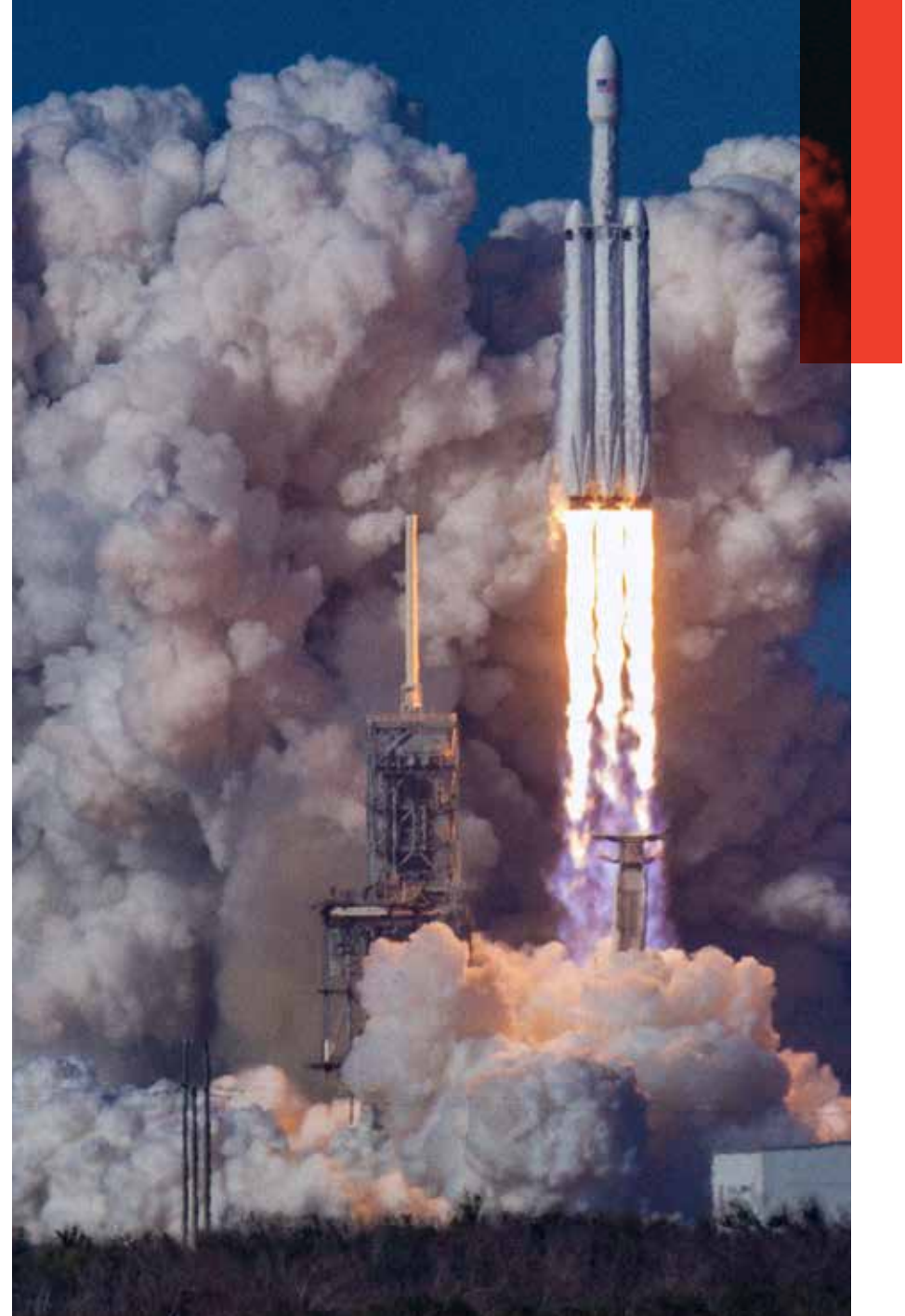
Schema enforcement rejects any new columns or other schema changes that aren't compatible with your table. By setting and upholding these high standards, analysts and engineers can trust that their data has the highest levels of integrity and can reason about it with clarity, allowing them to make better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



A control room with multiple monitors and desks. The room is dimly lit, with the primary light source being the screens. The monitors display various data, including a color calibration chart and a digital clock showing 23:59:59:24. The desks are equipped with laptops, keyboards, and mice. The overall atmosphere is professional and technical.

Delta Lake DML Internals

CHAPTER 04



04

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.

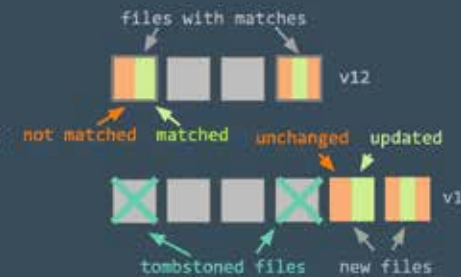
Delta Lake DML: UPDATE

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

1. Find and select the files containing **data that match the predicate**
2. Read each matching file into memory, update the relevant rows, and write out the result into a new data file.



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

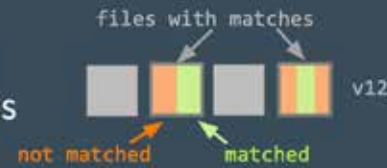
Delta Lake does not automatically VACUUM old files – you must run the command yourself, as shown below. If you want to specify a retention period that is different from the default of seven days, you can provide it as a parameter.

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

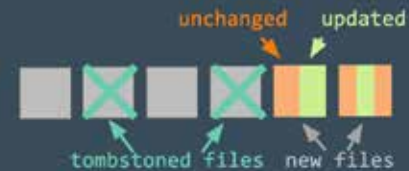


Merge – Under the hood

Scan 1: Inner join between target and source to select files that have matches



Scan 2: Outer join between the selected files in target and source and write the update/deleted/inserted data



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

CHAPTER 05

05

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

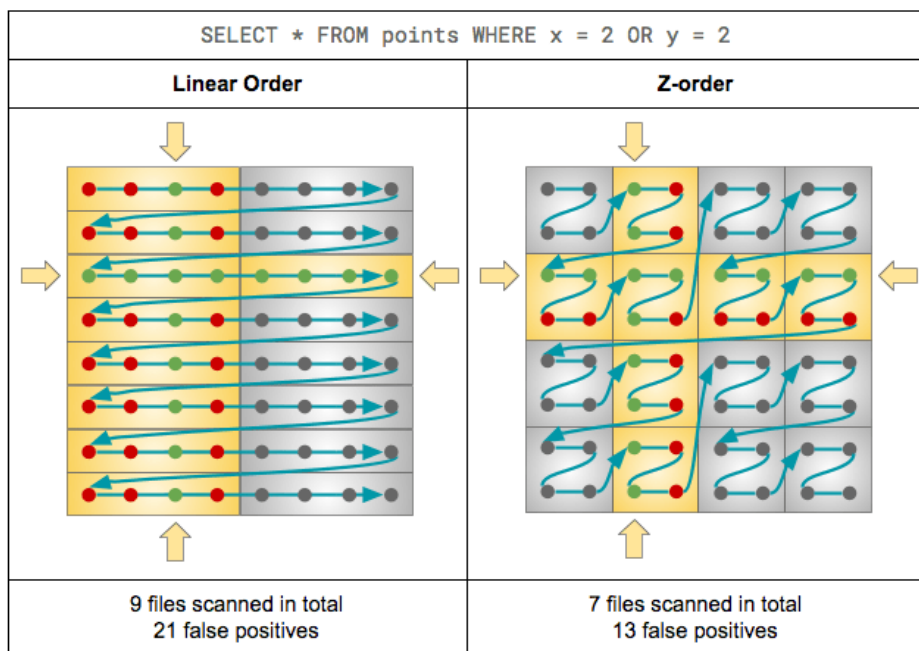
Delta Lake is capable of sifting through petabytes of data within seconds. Much of this speed is owed to two features: (1) data skipping and (2) Z-Ordering.

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 >](#)

What's next?

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

Explore subsequent eBooks in the collection >

- [The Delta Lake Series – Features](#)
- [The Delta Lake Series – Lakehouse](#)
- [The Delta Lake Series – Streaming](#)
- [The Delta Lake Series – Customer Use Cases](#)

Do a deep dive into Delta Lake >

[Visit the site for additional resources](#)

[Try Databricks for free >](#)

[Learn more >](#)