



DELTA LAKE

Agenda

- Introduction
- Why Delta Lake
- Key Features
- How to use Delta Lake in Databricks
- Delta Lake Examples
- Delta Lake DML Operations
- Delta Lake Versioning
- Delta Lake SCD Type 2
- Delta Lake Partitioning
- Delta Lake Clone

Introduction

- Delta Lake is an open source storage layer that brings reliability to data lakes.
- Delta Lake provides ACID transactions, scalable metadata handling
- Delta Lake runs on top of your existing data lake and is fully compatible with Apache Spark APIs.
- Delta Lake supports Parquet format

Challenges faced by most of the data lakes

1. Collect Everything



Garbage In

2. Store it all in the Data Lake



Garbage Stored

3. Data Science & Machine Learning



- Recommendation Engines
- Risk, Fraud Detection
- IoT & Predictive Maintenance
- Genomics & DNA Sequencing

Garbage Out

Why Delta?

- Challenges in implementation of a data lake.
- Missing ACID properties.
- Lack of Schema enforcement.
- Lack of Consistency.
- Lack of Data Quality.
- Too many small files.
- Corrupted data due to Frequent job failures in prod



	Data Warehouse	Hadoop M/R	
Separate Compute & Storage	✗	✓	✓
More than SQL (i.e ML)	✗	✓	✓
Open Source at Scale	✗	✓	✓
SQL & Optimization	✓	✗	✓
Data Model & Catalog	✓	✗	✓
ACID Transactions	✓	✗	✓



Challenges with Data Lakes: Reliability



Failed production jobs leave data in corrupt state requiring tedious recovery



Lack of consistency makes it almost impossible to mix appends, deletes, upserts and get consistent reads

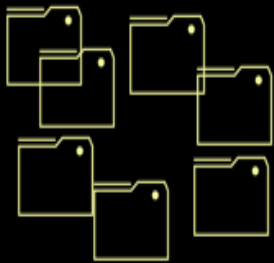


Lack of schema enforcement creates inconsistent and low quality data

Challenges with Data Lakes: Performance



Lack of consistency makes multi-processing impossible.



Too many small files - more time opening & closing files rather than reading contents (worse with streaming).



Partitioning aka “poor man’s indexing” - **breaks down** if you picked the wrong fields or when data has many dimensions, high cardinality columns.



No caching - cloud storage throughput is low



DELTA

The
SCALE
of data lake

The
**RELIABILITY &
PERFORMANCE**
of data warehouse

The
LOW-LATENCY
of streaming

Challenges solved: Reliability



Problem:

Failed production jobs leave data in corrupt state requiring tedious recovery



Solution:

Failed write jobs do not update the commit log, hence partial / corrupt files not visible to readers

Challenges solved: Reliability



Challenge :

Lack of consistency makes it almost impossible to mix appends, deletes, upserts and get consistent reads



Solution:

All reads have full snapshot consistency

All successful writes are consistent

In practice, most writes don't conflict

Tunable isolation levels

Challenges solved: Reliability



Challenge :

Lack of schema enforcement creates inconsistent and low quality data

Solution:

Schema recorded in the log

Fails attempts to commit data with incorrect schema

Allows explicit schema evolution

Allows invariant and constraint checks (high data quality)



Challenges solved: Performance



Challenge:

Too many small files increase resource usage significantly

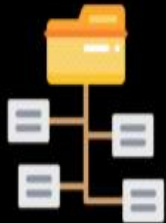


Solution:

Transactionally performed compaction using OPTIMIZE

```
OPTIMIZE table WHERE date = '2019-04-04'
```

Challenges solved: Performance



Challenge:

Partitioning breaks down with many dimensions and/or high cardinality columns

Solution:

Optimize using multi-dimensional clustering on multiple columns



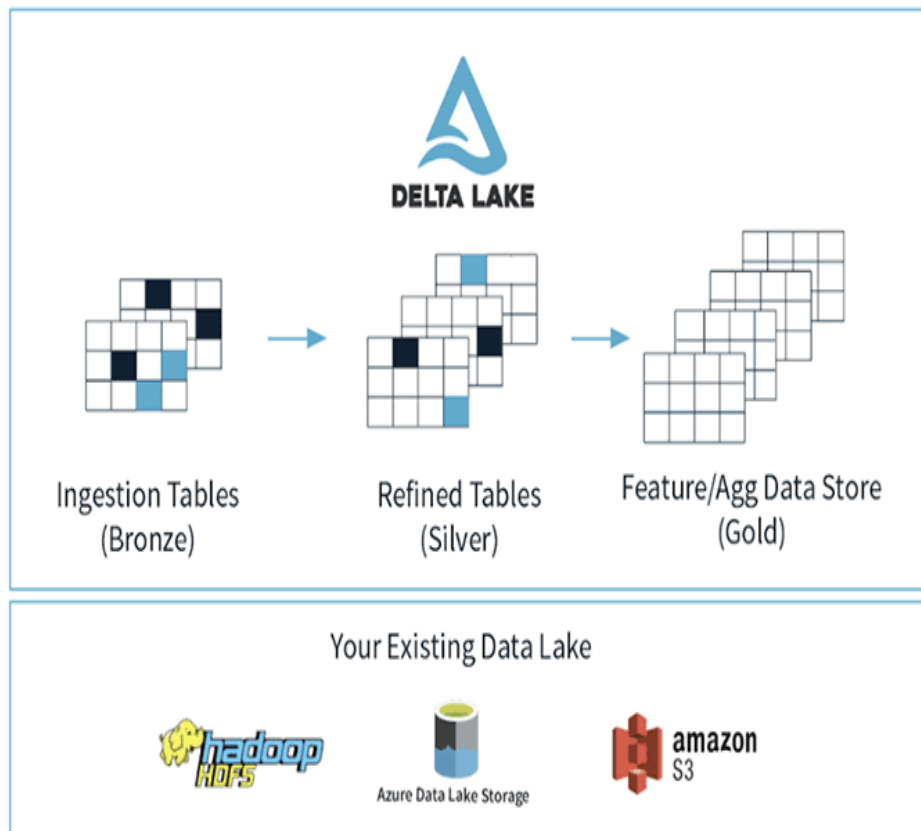
```
OPTIMIZE conns WHERE date = '2019-04-04'  
ZORDER BY (srcIP, destIP)
```


Delta Lake Features

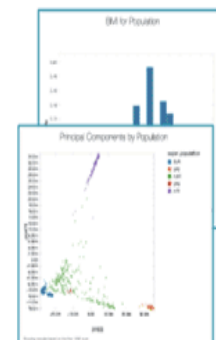
An open-source storage format that brings ACID transactions to Apache Spark™ and big data workloads.

- **Open format:** Stored as Parquet format in blob storage.
- **ACID Transactions:** Ensures data integrity and read consistency with complex, concurrent data pipelines.
- **Schema Enforcement and Evolution:** Ensures data cleanliness by blocking writes with unexpected.
- **Audit History:** History of all the operations that happened in the table.
- **Time Travel:** Query previous versions of the table by time or version number.
- **Deletes and upserts:** Supports deleting and upserting into tables with programmatic APIs.
- **Scalable Metadata management:** Able to handle millions of files are scaling the metadata operations with Spark.
- **Unified Batch and Streaming Source and Sink:** A table in Delta Lake is both a batch table, as well as a streaming source and sink. Streaming data ingest, batch historic backfill, and interactive queries all just work out of the box.

11,000 Patient
Records



Interactive Clinical
Dashboards



Bronze tables contain raw data ingested from various sources (JSON files, RDBMS data, IoT data, etc.).

Silver tables will provide a more refined view of our data. We can join fields from various bronze tables to enrich streaming records, or update account statuses based on recent activity.

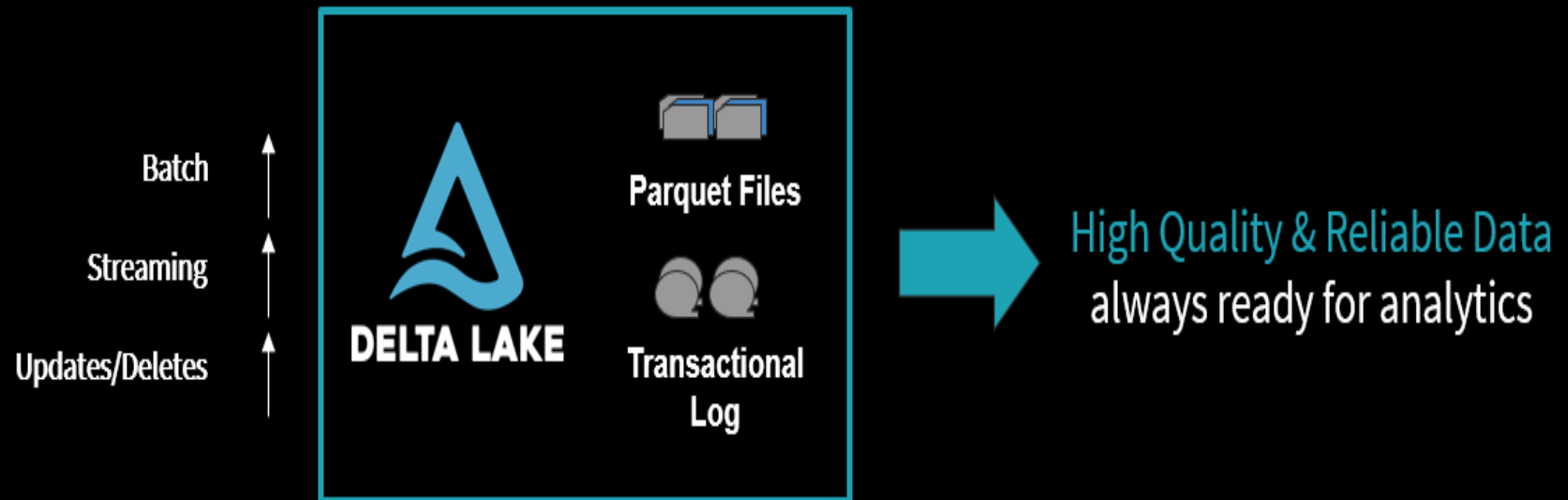
Gold tables provide business level aggregates often used for reporting and dashboarding. This would include aggregations such as daily active website users, weekly sales per store, or gross revenue per quarter by department.

The end outputs are actionable insights, dashboards, and reports of business metrics.

How Delta Lake Works

- Delta lake provides a storage layer on top of existing storage data lake. It acts as a middle layer between Spark runtime and storage
- Delta Lake will generate delta logs for each committed transactions
- Delta logs will have delta files stored as JSON which has information about the operations occurred
- Delta files are sequentially increasing named JSON files and together make up the log of all changes that have occurred to a table

Delta Lake ensures data *reliability*



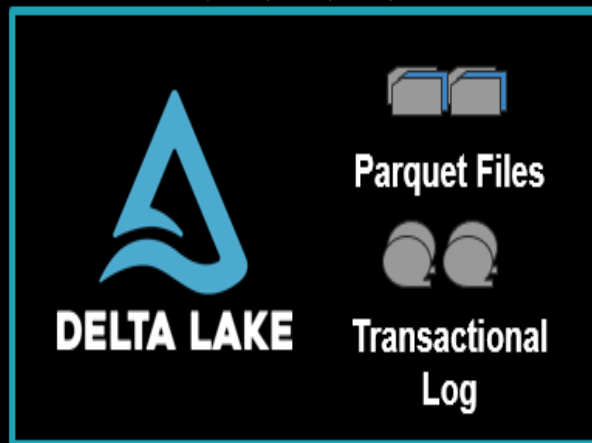
Key Features

- ACID Transactions
- Inserts/Updates/Deletes
- Schema Enforcement
- Unified Batch & Streaming
- Time Travel

Delta Lake optimizes *performance*



Highly Performant
queries at scale



Key Features

- Compaction
- Caching
- Data skipping
- Z-Ordering ²⁰

We have seen that Spark and Delta Lake make it easy for us to ingest data from disparate sources, and work with it as a relational database.

- **Cleansing** the data to remove corrupt or inaccurate information
- **Formatting** and pruning the data for downstream use
- **Enriching** the data by joining it with other data sources
- **Aggregating** the data to make it more convenient for use

Get Started with Delta using Spark APIs

Add Spark Package

```
pyspark --packages io.delta:delta-core_2.12:0.1.0  
bin/spark-shell --packages io.delta:delta-core_2.12:0.1.0
```

Maven

```
<dependency>  
  <groupId>io.delta</groupId>  
  <artifactId>delta-core_2.12</artifactId>  
  <version>0.1.0</version>  
</dependency>
```

Instead of **parquet**...

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

... simply say **delta**

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


Creating Delta Table in SQL & Python

```
df.write.format("delta").saveAsTable("events")      # create table in the metastore

df.write.format("delta").save("/delta/events")      # create table by path

df.write.format("delta").partitionBy("date").saveAsTable("events")      # create table in the metastore

df.write.format("delta").partitionBy("date").save("/delta/events")      # create table by path
```


```
-- Create table in the metastore
CREATE TABLE events (
  date DATE,
  eventId STRING,
  eventType STRING,
  data STRING)
USING DELTA
```

```
-- 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 your source files are in Parquet format, you can use the SQL Convert to Delta statement to convert files in place to create an unmanaged table:

```
CONVERT TO DELTA parquet.`/mnt/delta/events`
```


Delta Lake Storage & Types Of Files



databricks[®]
DELTA

$$= \text{Scalable storage} + \text{Transactional log}$$



Versioned
Parquet Files



Transactional
Delta Log



Indexes &
Stats

Tables created with a specified LOCATION are considered unmanaged by the metastore. Unlike a managed table, where no path is specified, an unmanaged table's files are not deleted when you DROP the table.

```
CREATE TABLE events
USING DELTA
LOCATION '/delta/events'
```

the table in the Hive metastore automatically inherits the schema, partitioning, and table properties of the existing data. This functionality can be used to “import” data into the metastore.

Reading Delta Table in SQL & Python

```
SELECT * FROM events    -- query table in the metastore
```

```
SELECT * FROM delta.`/delta/events`  -- query table by path
```

```
spark.table("events")    # query table in the metastore
```

```
spark.read.format("delta").load("/delta/events")  # query table by path
```

Batch upserts

To merge a set of updates and insertions into an existing table, you use the **MERGE INTO** statement. For example, the following statement takes a stream of updates and merges it into the events table. When there is already an event present with the same eventId, Delta Lake updates the data column using the given expression. When there is no matching event, Delta Lake adds a new row.

SQL

```
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)
```

You must specify a value for every column in your table when you perform an INSERT (for example, when there is no matching row in the existing dataset). However, you do not need to update all values.

Scalable storage

table data stored as Parquet files
on HDFS, AWS S3, Azure Blob Stores

pathToTable/

+---- 000.parquet

+---- 001.parquet

+---- 002.parquet

+ ...

|

+---- _delta_log/

+---- 000.json

+---- 001.json

...

Transactional log

sequence of metadata files to track
operations made on the table

stored in scalable storage along with table

Log Structured Storage

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

Each commit is a set of
actions file in directory
_delta_log

_delta_log/

000.json

001.json

INSERT actions

Add 001.parquet

Add 002.parquet

UPDATE actions

Remove 001.parquet

Remove 002.parquet

Add 003.parquet

Table = result of a set of actions

Change Metadata – name, schema, partitioning, etc

Add File – adds a file (with optional statistics)

Remove File – removes a file

Result: Current Metadata, List of Files, List of Txns, Version

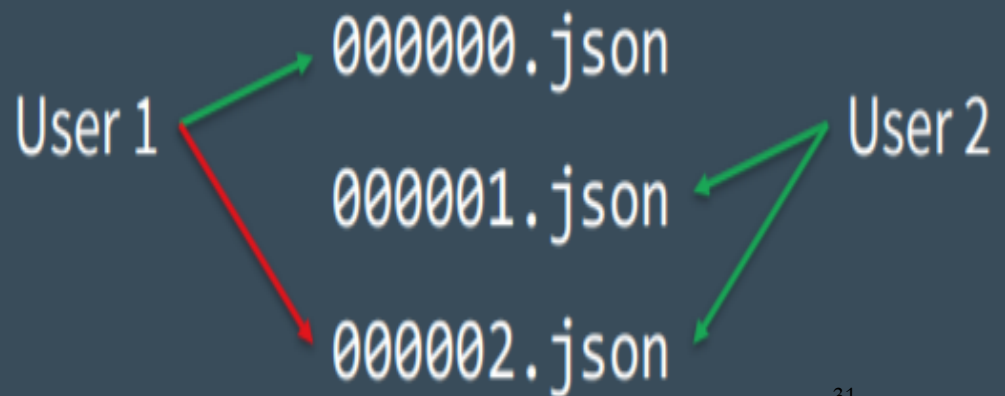
Implementing Atomicity

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



Ensuring Serializability

Need to agree on the order of changes, even when there are multiple writers.

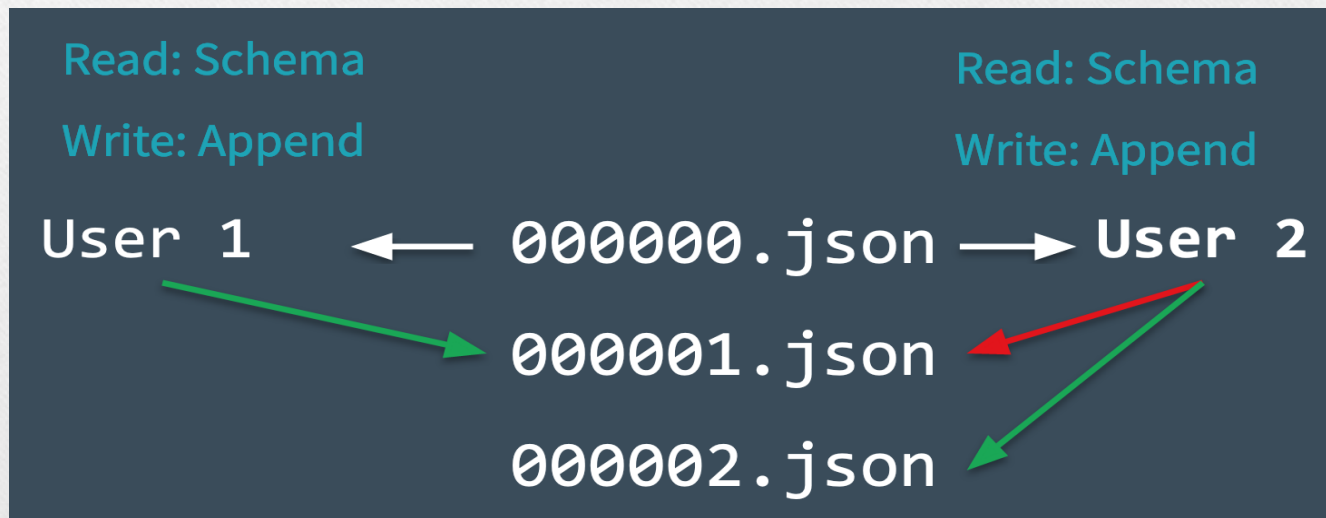


Solving Conflicts Optimistically

In order to offer ACID transactions, Delta Lake has a protocol for figuring out how commits should be ordered (known as the concept of **serializability** in databases). It should allow two or more commits are made at the same time. Delta Lake handles these cases by implementing a rule of mutual exclusion, then attempting to solve any conflict optimistically. This protocol allows Delta Lake to deliver on the ACID principle of isolation,

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

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



Breaking Down Transactions Into Atomic Commits

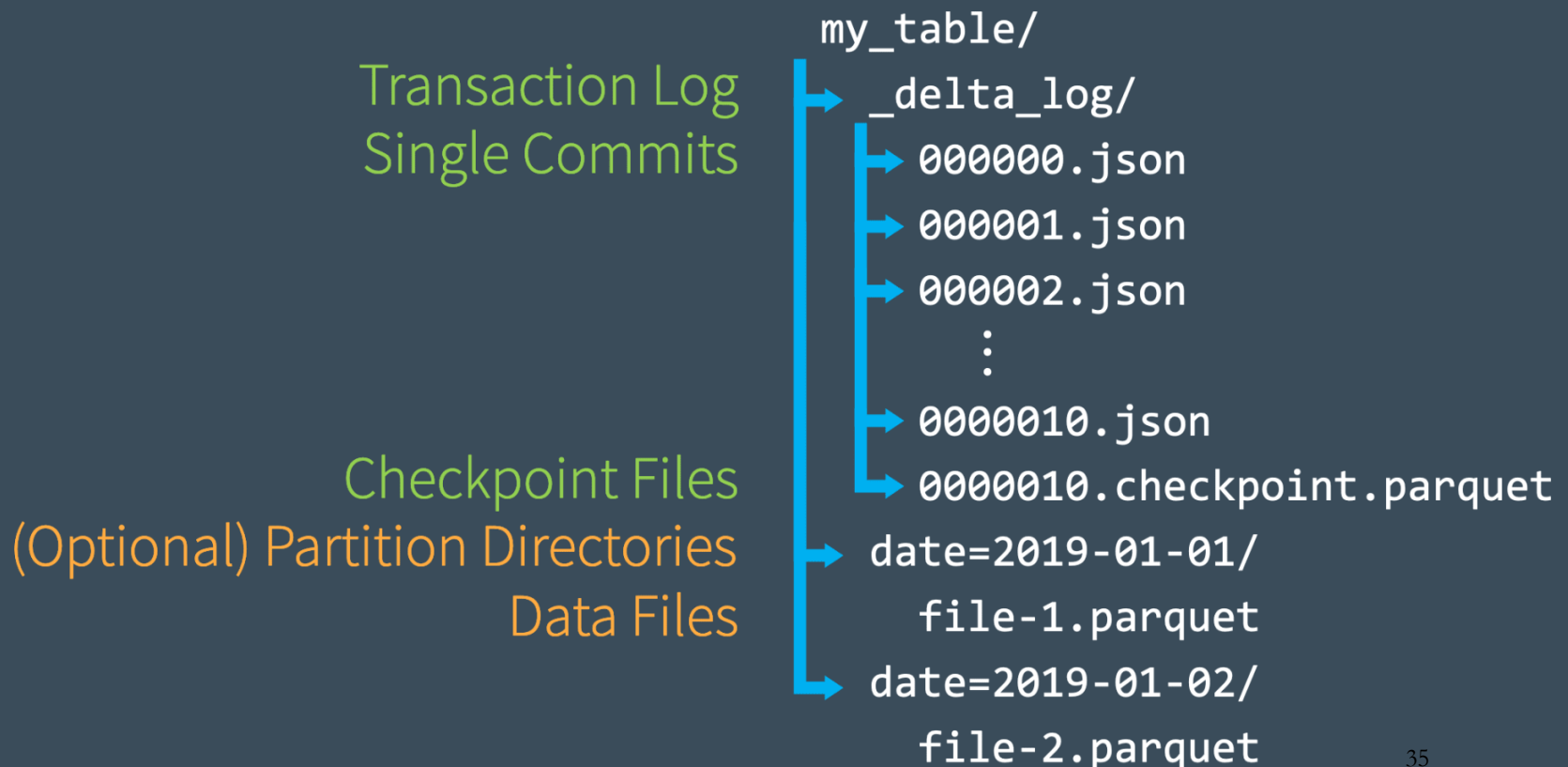
Whenever a user performs an operation to modify a table (such as an INSERT, UPDATE or DELETE), Delta Lake breaks that operation down into a series of discrete steps composed of one or more of the **actions** below.

- **Add file** – adds a data file.
- **Remove file** – removes a data file.
- **Update metadata** – Updates the table's metadata (e.g., changing the table's name, schema or partitioning).
- **Set transaction** – Records that a structured streaming job has committed a micro-batch with the given ID.
- **Change protocol** – enables new features by switching the Delta Lake transaction log to the newest software protocol.
- **Commit info** – Contains information around the commit, which operation was made, from where and at what time.

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

Quickly Recomputing State With Checkpoint Files

Once we've made a total of 10 commits to the transaction log, Delta Lake saves a checkpoint file in Parquet format in the same `_delta_log` subdirectory. Delta Lake automatically generates checkpoint files every 10 commits.



Major Differences between the CSV table and Delta Lake table

- Delta Lake adds a Transaction Log
- Delta Lake data is stored in Parquet format

These differences are the heart and soul of Delta Lake.

- The transaction log enables ACID compliance and many other important features. We'll be looking more deeply into these features throughout the rest of this workshop.

- Parquet is a popular data format for many data lakes. It stores data in columnar format, and generally provides faster performance. Let's see how Delta Lake's Data Skipping improves query performance over CSV.

Table Partitioning

All Big Data lakes divide logical tables into physical partitions. This keeps physical file sizes manageable, and can also be used to speed query processing.

Summarizing Topic 1: Partitioning

1. We have seen the benefits of Partitioning, a performance-enhancing feature common to all Big Data lakes. As we have seen:
2. Partitioning splits large files into smaller chunks
3. We can choose a semantic partition key that can make appropriate queries run much faster

However, partitioning has some limitations:

- We can't use partitioning to support a wide range of diverse queries. In order to benefit, queries must filter on the partition key
- We must choose a partition key of moderate cardinality

Optimize performance with file management

To improve query speed, Delta Lake on Databricks supports the ability to optimize the layout of data stored in cloud storage. Delta Lake on Databricks supports few algorithms: **bin-packing** , **Data-Skipping** And **Z-Ordering**.

Compaction (bin-packing)

Delta Lake on Databricks can improve the speed of read queries from a table by coalescing small files into larger ones. You trigger compaction by running the **OPTIMIZE** command

SQL

```
OPTIMIZE delta.`/data/events`
```

or

SQL

```
OPTIMIZE events
```

If you have a large amount of data and only want to optimize a subset of it, you can specify an optional partition predicate using **WHERE**

SQL

```
OPTIMIZE events WHERE date >= '2017-01-01'
```


Data Skipping

Databricks' Data Skipping feature takes advantage of the multi-file structure. As new data is inserted into a Databricks Delta table, file-level min/max statistics are collected for **all** columns. Then, when there's a lookup query against the table, Databricks Delta first consults these statistics in order to determine which files can safely be skipped. The picture below illustrates the process:

1. Keep track of simple statistics such as minimum and maximum values at a certain granularity that's correlated with I/O granularity.
2. Leverage those statistics at query planning time in order to avoid unnecessary I/O.

```
SELECT input_file_name() as "file_name",  
       min(col) AS "col_min",  
       max(col) AS "col_max"  
FROM table  
GROUP BY input_file_name()
```

file_name	col_min	col_max
data_file_1	6	8
data_file_2	3	10
data_file_3	1	4

Data Skipping Example

In this example, we skip file 1 because its minimum value is higher than our desired value.

```
SELECT * FROM table WHERE col <= 5
```



```
SELECT file_name FROM index  
WHERE col_min <= 5
```

file_name	col_min	col_max
data_file_1	6	8
data_file_2	3	10
data_file_3	1	4

Similarly, we skip file 3 based on its maximum value. File 2 is the only one we need to access.

```
SELECT * FROM table WHERE col == 5
```



```
SELECT file_name FROM index  
WHERE col_min <= 5 AND col_max >= 5
```

file_name	col_min	col_max
data_file_1	6	8
data_file_2	3	10
data_file_3	1	4

Z-Ordering (multi-dimensional clustering)

Z-Ordering is another Databricks enhancement to Delta Lake.

What is Z-Ordering?

Relational databases use *secondary indexes* to help speed queries. However, indexing becomes impractical with Big Data; the indexes themselves become very large, and they cannot be updated at write time in a performant manner. Z-ordering is a technique that replaces indexes by placing data columns that are "close" in value into the same physical files within a partition.

Z-Ordering is a technique to colocate related information in the same set of files. This co-locality is automatically used by Delta Lake on Databricks **data-skipping** algorithms to dramatically reduce the amount of data that needs to be read. To Z-Order data, you specify the columns to order on in the ZORDER BY clause:

The chess board is a 2-dimensional array. But when I persist the data, I need to express it in only one dimension; in other words, I need to *serialize* it. My desire is to have squares that are close to each other in the array remain close to each other in physical files in my Delta partition directories. For this illustration, I've decided that each physical file should hold 4 squares.

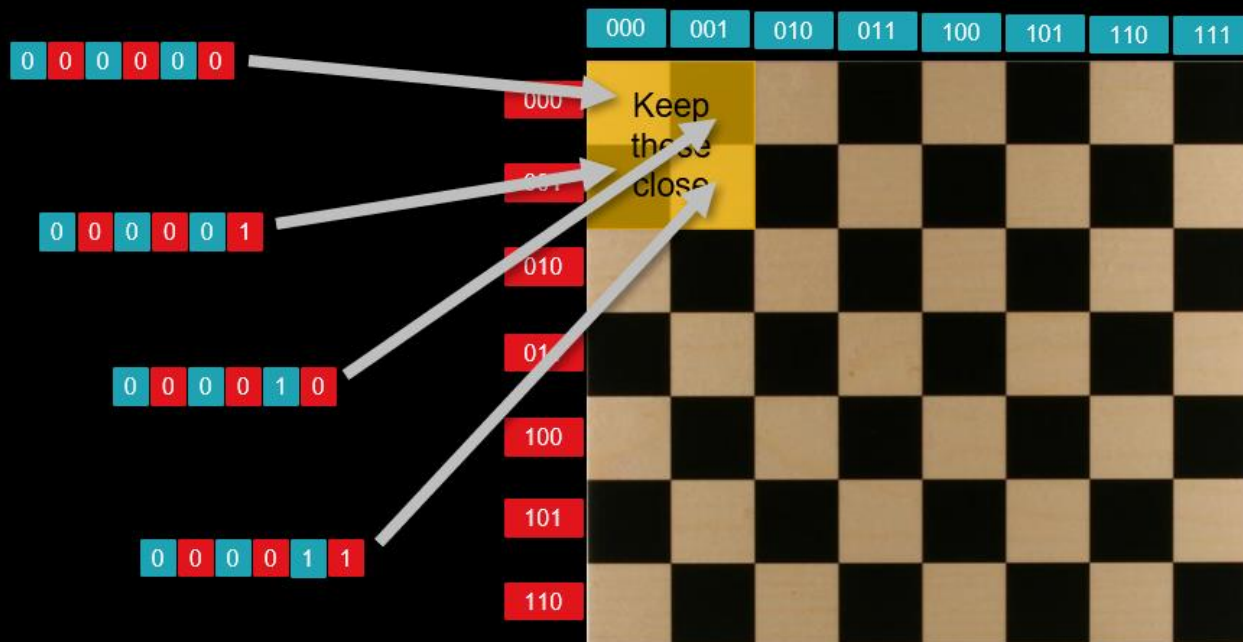
How can we store multi-dimensional data efficiently in a one-dimensional file?

	000	001	010	011	100	101	110	111
000	Keep these close							
001								
010								
011								
100								
101								
110								

If I serialize row-by-row, I'll distance myself from 2 of my closest neighbors.
The same thing happens if I serialize column-by-column.

Z-ordering is a solution to this problem.

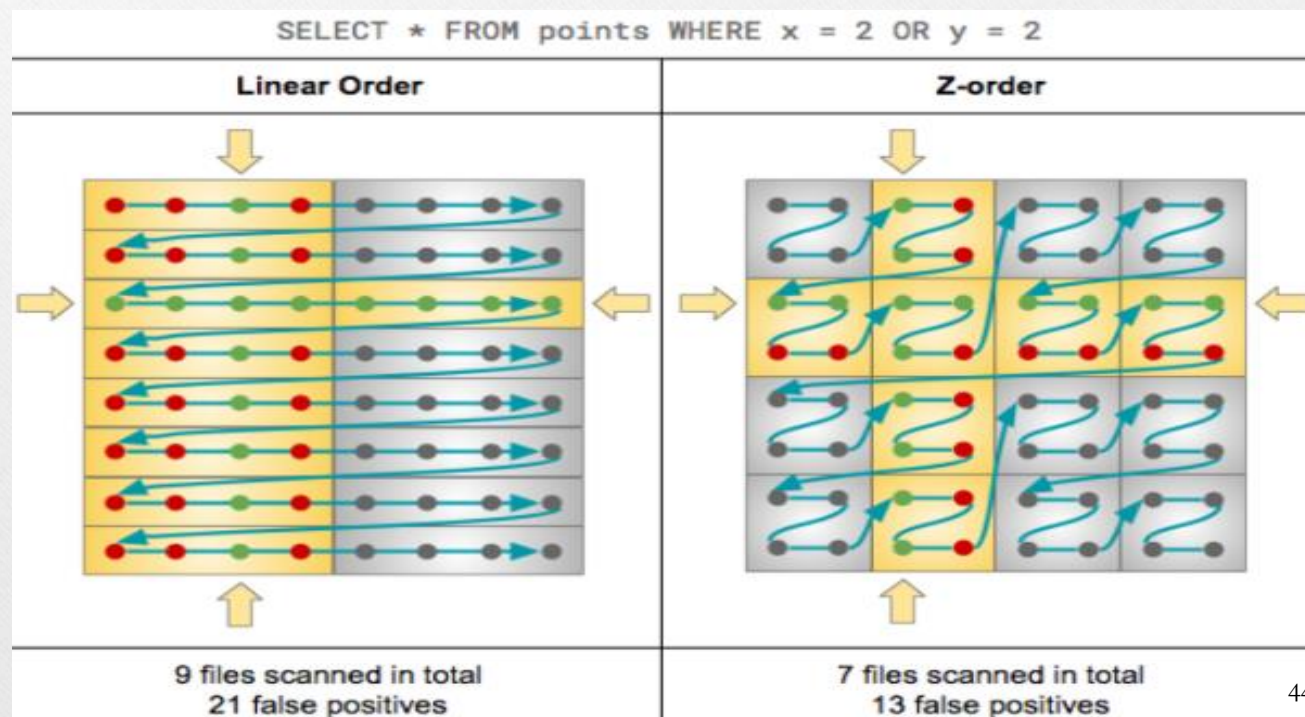
Solution: Interleave the bits!



If we interleave the bits of each dimension's values, and sort by the result, we can write files where the 2-dimensional neighbors are also close in 1 dimension.

Even better, Z-ordering works even if the values are of different lengths. It also works across more than 2 dimensions, although it breaks down quickly after 4-5.

Z-ordering lets us skip more files, and get fewer false positives in the files we do read.



ZORDER - Examples..

```
%sql
```

```
-- NOTE: this cell may take several minutes to run  
-- If time is tight, you may want to just read through the rest of this section
```

```
DROP TABLE IF EXISTS flights_delta_z;
```

```
-- create a Delta table
```

```
CREATE TABLE IF NOT EXISTS flights_delta_z  
  USING DELTA  
  PARTITIONED BY (Origin)  
  AS  
    SELECT *  
    FROM flights_temp;
```

```
-- Now add Z-Ordering
```

```
-- In this example, we'll just Z-Order by one column, TailNum,  
-- to see how it improves queries using TailNum with the partition key, Origin
```

```
OPTIMIZE flights_delta_z ZORDER BY (TailNum)
```

```
SQL
```

```
OPTIMIZE events
```

```
WHERE date >= current_timestamp() - INTERVAL 1 day  
ZORDER BY (eventType)
```

Star Schemas and Dynamic Partition Pruning

Notice that the Dimension table introduces a level of indirection between the query and the partitions we would like to prune (which are on the Fact table). In most data lakes, this means that we will have to scan the entire Fact table (which of course, is the biggest table in a star schema). Databricks' query optimizer, however, will understand this query, and skip any partitions on the fact table that do not match the filter.

The problem of slowly-changing dimensions

Example of a supplier table:

Supplier_Key	Supplier_Code	Supplier_Name	Supplier_State
123	ABC	Acme Supply Co	CA

What happens when the Supplier moves to a new state?

Star Schemas and Dynamic Partition Pruning

The **star schema** is very effective for Analytics queries, as long as all the dimensions stay the same. But what happens when something changes in a dimension table? For example, here is a dimension table that represents our company's Suppliers. Suppose we are keeping 5 years of history in our warehouse, and at some point, say year 3, this Supplier moves its facilities to a new State? If we are building reports that report on Suppliers grouped by State, how do we keep our report accurate?

Remember, we're reporting over a 5-year history, so the problem is that we want results that are accurate over that whole time period.

Common solutions...

Slowly Changing Dimensions (SCD):

- **Type 0:** No changes allowed (static/append only)
 - **Useful for:** static lookup table
- **Type 1:** Overwrite (no history retained)
 - **Useful when:** do not care about historic comparisons other than quite recent (use Delta Time Travel)
- **Type 2:** Adding a new row for each change and marking the old as obsolete
 - **Useful when:** Must record product price changes over time, integral to business logic.

Star Schemas and Dynamic Partition Pruning

There are several design choices available to solve the SCD problem. There are actually more choices than we're showing here, but these are the most common. These solutions are known by Type x, where x is a number from 1 to 6 (although there is actually no Type 5).

Type 0 is easy (but not very effective). We simply refuse to allow changes. In Type 1 we simply overwrite the old information with new information. In Type 2, we keep a historical trail of the information. Above type 2, we simply implement more sophisticated ways of keeping history. Let's look deeper into Types 1 and 2...

SCD Type 1

Example of a supplier table:

Supplier_Key	Supplier_Code	Supplier_Name	Supplier_State
123	ABC	Acme Supply Co	CA

If the supplier relocates the headquarters to Illinois the record would be overwritten:

Supplier_Key	Supplier_Code	Supplier_Name	Supplier_State
123	ABC	Acme Supply Co	IL

Star Schemas and Dynamic Partition Pruning

Here is a Type 1 example. When the Supplier moves from CA to IL, we simply overwrite the information in the record. That means that the older historical parts of our report will now be incorrect, because it will appear that this Supplier was *always* in IL.

SCD Type 2

Supplier_Key	Supplier_Code	Supplier_Name	Supplier	Version
123	ABC	Acme Supply Co	CA	0
124	ABC	Acme Supply Co	IL	1

Another method is to add 'effective date' columns.

Supplier_Key	Supplier_Code	Supplier_Name	Supplier_State	Start_Date	End_Date
123	ABC	Acme Supply Co	CA	2000-01-01T00:00:00	2004-12-22T00:00:00
124	ABC	Acme Supply Co	IL	2004-12-22T00:00:00	NULL

And a third method uses an effective date and a current flag.

Supplier_Key	Supplier_Code	Supplier_Name	Supplier_State	Effective_Date	Current_Flag
123	ABC	Acme Supply Co	CA	2000-01-01T00:00:00	N
124	ABC	Acme Supply Co	IL	2004-12-22T00:00:00	Y

Star Schemas and Dynamic Partition Pruning

%sql

```
-- =====  
-- Merge SQL API is available since DBR 5.1  
-- =====
```

MERGE INTO **customers**

USING (

```
-- These rows will either UPDATE the current addresses of existing customers  
--or INSERT the new addresses of new customers
```

```
SELECT updates.customerId as mergeKey, updates.*  
FROM updates
```

UNION ALL

```
-- These rows will INSERT new addresses of existing customers  
-- Setting the mergeKey to NULL forces these rows to NOT MATCH and be INSERTed.
```

```
SELECT NULL as mergeKey, updates.*
```

```
FROM updates JOIN customers
```

```
ON updates.customerid = customers.customerid
```

```
WHERE customers.current = true AND updates.address <> customers.address
```

) staged_updates

ON **customers.customerId** = mergeKey

WHEN MATCHED AND customers.current = true AND customers.address <> staged_updates.address THEN

```
UPDATE SET current = false, endDate = staged_updates.effectiveDate
```

```
-- Set current to false and endDate to source's effective date.
```

WHEN NOT MATCHED THEN

```
INSERT(customerid, address, current, effectiveDate, endDate)
```

```
VALUES(staged_updates.customerId, staged_updates.address, true, staged_updates.effectiveDate, null)
```

```
-- Set current to true along with the new address and its effective date.
```


Star Schemas and Dynamic Partition Pruning

Here are three different ways we might implement a Type 2 solution. Our goal here is to make sure our historical reports are accurate throughout the entire time period.

In the top example, we add a new row to the dimension table whenever data changes, and we keep a version number to show the order of the changes. Why do you think this might be an ineffective solution? Is it important to know **when** in time each version changed?

In the middle example, we see a more effective solution. Each row for the supplier has a start and end date. If we design our queries well, we can make sure that each time period in our report uses the corresponding Supplier row. If there is no data in the End_Date column, we know we have the current information.

The bottom example is similar. Effective_Date is the same as Start_Date in the middle example. Instead of End_Date, we have a flag that says whether or not this row is current. Our queries may get a bit more complex here, because we must read a row to get the start date, then read the next row to determine the end date (assuming we are using non-current rows).

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.

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.

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


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.
- **Cannot have 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.
- **Can not contain 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.

Query an older snapshot of a table (time travel)

Delta Lake time travel allows you to query an older snapshot of a Delta table. Time travel has many use cases, including:

1. Re-creating analyses, reports, or outputs (for example, the output of a machine learning model). This could be useful for debugging or auditing, especially in regulated industries.
2. Writing complex temporal queries.
3. Fixing mistakes in your data.
4. Providing snapshot isolation for a set of queries for fast changing tables.

Python

```
df1 = spark.read.format("delta").option("timestampAsOf", timestamp_string).load("/delta/events")  
df2 = spark.read.format("delta").option("versionAsOf", version).load("/delta/events")
```

Delta Lake Data retention

By default, Delta tables retain the commit history for 30 days. This means that you can specify a version from 30 days ago. However, there are some caveats:

`vacuum` deletes only data files, not log files. Log files are deleted automatically and asynchronously after checkpoint operations. The default retention period of log files is 30 days, configurable through the **`delta.logRetentionPeriod`** property which you set with the `ALTER TABLE SET TBLPROPERTIES` SQL method.

`delta.logRetentionDuration` = "interval <interval>": controls how long the history for a table is kept. Each time a checkpoint is written

`delta.deletedFileRetentionDuration` = "interval <interval>": controls how long ago a file must have been deleted before being a candidate for `VACUUM`. The default is interval 7 days

NOTE: `VACUUM` doesn't clean up log files; log files are automatically cleaned up after checkpoints are written.

Data Recovery Based on Snapshots

Fix accidental deletes to a table for the user 111:

```
INSERT INTO my_table
  SELECT * FROM my_table TIMESTAMP AS OF date_sub(current_date(), 1)
  WHERE userId = 111
```

Fix accidental incorrect updates to a table:

```
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 *
```

Query the number of new customers added over the last week.

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

Query an earlier version of the table (time travel)

```
SELECT * FROM events VERSION AS OF 0
```

```
SELECT * FROM events TIMESTAMP AS OF '2019-01-29 00:37:58'
```


Clean up snapshots

Delta Lake provides snapshot isolation for reads, which means that it is safe to run OPTIMIZE even while other users or jobs are querying the table. Eventually however, you should clean up old snapshots. You can do this by running the VACUUM command:

```
VACUUM events
```

You control the age of the latest retained snapshot by using the RETAIN <N> HOURS option:

```
VACUUM events RETAIN 24 HOURS
```

```
VACUUM eventsTable -- vacuum files not required by versions older than the default retention period
```

```
VACUUM '/data/events' -- vacuum files in path-based table
```

```
VACUUM delta.`/data/events/`
```

```
VACUUM delta.`/data/events/` RETAIN 100 HOURS -- vacuum files not required by versions more than 100 hours old
```

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

Describe Detail History

You can retrieve more information about the table (for example, number of files, data size) using DESCRIBE DETAIL.

SQL

```
DESCRIBE DETAIL '/data/events/'
```

```
DESCRIBE HISTORY delta.`/data/events/`
```

```
DESCRIBE HISTORY eventsTable
```


Clone a Delta table

You can create a copy of an existing Delta table at a specific version using the clone command. Clones can be either deep or shallow.

A deep clone is a clone copies the source table data to the clone target in addition to the metadata of the existing table. Additionally, stream metadata is also cloned such that a stream that writes to the Delta table can be stopped on a source table and continued on the target of a clone from where it left off.

A shallow clone is a clone that does not copy the data files to the clone target. The table metadata is equivalent to the source. These clones are cheaper to create.

Note: Any changes made to either deep or shallow clones affect only the clones themselves and not the source table.


```
%sql
-- Create a deep clone of /data/source at /data/target
CREATE TABLE delta.`/data/target/` CLONE delta.`/data/source/` |
-- Replace the target
CREATE OR REPLACE TABLE db.target_table CLONE db.source_table
-- No-op if the target table exists
CREATE TABLE IF NOT EXISTS TABLE delta.`/data/target/` CLONE db.source_table

CREATE TABLE db.target_table SHALLOW CLONE delta.`/data/source`

CREATE TABLE db.target_table SHALLOW CLONE delta.`/data/source` VERSION AS OF version
-- timestamp can be like "2019-01-01" or like date_sub(current_date(), 1)
CREATE TABLE db.target_table SHALLOW CLONE delta.`/data/source` TIMESTAMP AS OF timestamp_expression
```

Clone use cases

Data archiving: Data may need to be kept for longer than is feasible with time travel or for disaster recovery. In these cases, you can create a deep clone to preserve the state of a table at a certain point in time for archival. Incremental archiving is also possible to keep a continually updating state of a source table for disaster recovery.

SQL

```
-- Every month run  
CREATE OR REPLACE TABLE delta.`/some/archive/path` CLONE my_prod_table
```

Machine learning flow reproduction: When doing machine learning, you may want to archive a certain version of a table on which you trained an ML model. Future models can be tested using this archived data set.

SQL

```
-- Trained model on version 15 of Delta table  
CREATE TABLE delta.`/model/dataset` CLONE entire_dataset VERSION AS OF 15
```

Clone use cases Continue...

Short-term experiments on a production table: In order to test out a workflow on a production table without corrupting the table, you can easily create a shallow clone. This allows you to run arbitrary workflows on the cloned table that contains all the production data but does not affect any production workloads.

SQL

```
-- Perform shallow clone
CREATE OR REPLACE TABLE my_test SHALLOW CLONE my_prod_table;

UPDATE my_test WHERE user_id is null SET invalid=true;
-- Run a bunch of validations. Once happy:

-- This should leverage the update information in the clone to prune to only
-- changed files in the clone if possible
MERGE INTO my_prod_table
USING my_test
ON my_test.user_id <=> my_prod_table.user_id
WHEN MATCHED AND my_test.user_id is null THEN UPDATE *;

DROP TABLE my_test;
```


Clone use cases Continue...

Data sharing: Other business units within a single organization may want to access the same data but may not require the latest updates. Instead of giving access to the source table directly, clones with different permissions can be provided for different business units. The performance of the clone can exceed that of a simple view as well.

SQL

```
-- Perform deep clone
CREATE OR REPLACE TABLE shared_table CLONE my_prod_table;

-- Grant other users access to the shared table
GRANT SELECT ON shared_table TO `<user-name>@<user-domain>.com`;
```

Run VACUUM Regularly

To ensure that concurrent readers can continue reading a stale snapshot of a table, Databricks Delta leaves deleted files on DBFS for a period of time. The VACUUM command helps save on storage costs by cleaning up these invalid files. It can, however, interrupt users querying a Delta table similar to when partitions are re-written. VACUUM should be run regularly to clean up expired snapshots that are no longer required.

Batch Modifications

Parquet files, that form the underpinning of Delta, are immutable and thus need to be rewritten completely to reflect changes regardless of the extent of the change. Use MERGE INTO to batch changes to amortize costs.

Use DELETES

Manually deleting files from the underlying storage is likely to break the Delta table so instead you should use DELETE commands to ensure proper progression of the change.



THANK YOU