01 Delta Lake - KirkYagami 👨‍💻🕵️‍♂️

Delta Lake - KirkYagami 👨‍💻🕵️‍♂️

1. Introduction

Delta Lake is an open-source storage layer that brings reliability to data lakes. Created by Databricks, it 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.

Key points:

1. Open-source project developed by Databricks
2. Storage layer that enhances data lakes
3. Provides reliability and performance improvements
4. Compatible with existing data lake infrastructure
5. Supports both batch and streaming data processing
6. Designed to work seamlessly with Apache Spark
7. Addresses common challenges in big data processing
8. Enables building a Lakehouse architecture

2. Definition

Delta Lake is a robust open-source storage layer that sits atop existing data lakes, providing ACID transactions, scalable metadata handling, and unified data management for both batch and streaming workloads. It enhances data reliability, improves query performance, and enables the creation of a Lakehouse architecture, which combines the best features of data lakes and data warehouses.

Key aspects:

1. Storage layer for data lakes
2. Ensures ACID (Atomicity, Consistency, Isolation, Durability) transactions
3. Handles scalable metadata efficiently
4. Unifies batch and streaming data processing
5. Improves data reliability and integrity
6. Enhances query performance
7. Enables Lakehouse architecture
8. Maintains compatibility with existing data lake tools and processes

3. What Led to the Development of Delta Lake

Before Delta Lake, traditional data lakes faced several challenges, mainly related to data consistency, quality, and performance. Many enterprises struggled with:

1. **Data Corruption**: Lack of transactional support led to inconsistent and corrupt datasets, especially in concurrent write/read environments.
2. **Slow Query Performance**: Queries on large datasets were slow, as data lakes did not optimize reads and writes effectively.
3. **Lack of Schema Enforcement**: Data lakes could not enforce strict schemas, causing issues when the data structure changed.
4. **Data Duplication and Staleness**: Data lakes couldn't efficiently handle real-time streaming data alongside batch data.
5. **No Versioning**: Businesses couldn't revert to previous data versions, leading to operational challenges when data was mistakenly modified.
6. **Complex Data Management**: The separation of batch and streaming pipelines increased system complexity.
7. **Inconsistent Data Pipelines**: Without transactional support, pipelines lacked consistency, causing downstream analysis errors.
   Databricks, recognizing these pain points, developed Delta Lake to provide a robust solution that could work with existing data lake infrastructure while significantly enhancing its capabilities.

4. What is Delta Lake Solving

Delta Lake addresses these limitations by introducing a set of powerful features:

1. **ACID Transactions**: It guarantees data integrity even with concurrent reads and writes.
2. **Data Lineage and Versioning**: Time travel capabilities allow users to query historical versions of their data and audit changes.
3. **Efficient File Management**: It merges small files automatically and organizes large datasets for better performance.
4. **Schema Enforcement**: Ensures data consistency by allowing schemas to evolve but remain enforceable.
5. **Handling Streaming and Batch**: It unifies batch and real-time data processing, eliminating the need for separate architectures.
6. **Optimized Reads and Writes**: Delta Lake is optimized for large datasets, providing faster query performance.
7. **Seamless Integration with Big Data Tools**: Works with Apache Spark and other big data tools, reducing migration overhead.

1. ACID Transactions

Delta Lake guarantees ACID transactions, ensuring data integrity even with concurrent reads and writes. You can perform transactional operations using Delta's `write` and `MERGE`

commands.

```
from delta.tables import *

# Create a Delta table for Avengers data
data = [("Tony", "Stark", "Iron Man", "New York", 48),
        ("Steve", "Rogers", "Captain America", "Brooklyn", 99),
        ("Thor", "Odinson", "Thor", "Asgard", 1500)
       ]

schema = StructType([
    StructField("firstname", StringType(), True),
    StructField("lastname", StringType(), True),
    StructField("alias", StringType(), True),
    StructField("location", StringType(), True),
    StructField("age", IntegerType(), True)
])

avengers_df = spark.createDataFrame(data=data, schema=schema)
avengers_df.write.format("delta").mode("overwrite").save("data/delta-
avengers")

# Perform an upsert (ACID transaction) by updating Thor's location
delta_table = DeltaTable.forPath(spark, "data/delta-avengers")

new_data = [("Thor", "Odinson", "Thor", "Earth", 1500)]
new_data_df = spark.createDataFrame(new_data, schema=schema)

delta_table.alias("old").merge(
    new_data_df.alias("new"),
    "old.alias = new.alias"
).whenMatchedUpdateAll().execute()
```

2. Data Lineage and Versioning (Time Travel)

Delta Lake's time travel feature allows querying historical versions of the data, making it easy to access past data states.

```
# Retrieve previous versions of the Avengers Delta table
# Version 0 is the initial data load
historical_df = spark.read.format("delta").option("versionAsOf",
0).load("data/delta-avengers")
historical_df.show()
```

```
# Rollback to version 0
delta_table.restoreToVersion(0)
```

3. Efficient File Management

Delta Lake merges small files automatically for optimized performance. Here's an example to compact small files into larger ones using `OPTIMIZE`.

```
# Auto-compaction of small files using OPTIMIZE
delta_table.optimize().executeCompaction()
```

4. Schema Enforcement

Delta Lake ensures schema enforcement, so data inconsistencies are caught during write operations. Here's an example showing schema enforcement when adding a new column:

```
# Trying to add a new column 'weapon' to the table
data_with_weapon = [("Tony", "Stark", "Iron Man", "New York", 48, "Suit"),
                    ("Steve", "Rogers", "Captain America", "Brooklyn", 99,
"Shield")]

# Schema including the new column
new_schema = StructType(schema.fields + [StructField("weapon", StringType(),
True)])

# Write operation will fail because the Delta table schema is enforced
avengers_with_weapon_df = spark.createDataFrame(data_with_weapon,
schema=new_schema)
avengers_with_weapon_df.write.format("delta").mode("append").save("data/delta
-avengers")
```

5. Handling Streaming and Batch

Delta Lake allows you to handle both streaming and batch data in the same table, eliminating the need for separate architectures.

```
# Streaming data (new Avengers)
streaming_data = [("Natasha", "Romanoff", "Black Widow", "Russia", 35),
                  ("Clint", "Barton", "Hawkeye", "New York", 42)]

streaming_df = spark.createDataFrame(data=streaming_data, schema=schema)

# Simulate streaming write
streaming_query = streaming_df.writeStream \
    .format("delta") \
```

```
    .outputMode("append") \
    .option("checkpointLocation", "data/checkpoints") \
    .start("data/delta-avengers")

# Simulate streaming read
streaming_read = spark.readStream.format("delta").load("data/delta-avengers")
streaming_read.writeStream.format("console").start()
```

6. Optimized Reads and Writes

Delta Lake provides optimized reads and writes, reducing latency by compacting files and using advanced indexing techniques.

```
# Example of optimized reads: Use data skipping to only query data for
Avengers in New York
nyc_avengers = spark.read.format("delta").load("data/delta-
avengers").filter("location = 'New York'")
nyc_avengers.show()

# Example of optimized writes using compaction (OPTIMIZE)
delta_table.optimize().executeCompaction()
```

7. Seamless Integration with Big Data Tools

Delta Lake integrates well with Apache Spark and other big data tools, making data migration easier. Here's an example of reading Delta Lake data in Spark:

```
# Reading the Delta table using Apache Spark
df = spark.read.format("delta").load("data/delta-avengers")
df.show()

# Perform transformations and write back
transformed_df = df.filter(df.age > 50)
transformed_df.write.format("delta").mode("overwrite").save("data/delta-
transformed-avengers")
```

Each of these examples demonstrates the core functionalities and benefits that Delta Lake provides, from ACID transactions to optimized reads and writes.

Schema Evolution

Schema evolution in Delta Lake allows you to change the schema of your data as it evolves over time, while still maintaining compatibility with previously written data. This is particularly useful when you want to add or modify columns in an existing Delta table.

Below is the code demonstrating **schema evolution** using the Avengers example. We'll update the schema to add a new column (`weapon`) and show how Delta Lake can handle this.

Step 1: Writing Initial Data with the Original Schema

```
# Initial Avengers data without the 'weapon' column
data = [("Tony", "Stark", "Iron Man", "New York", 48),
        ("Steve", "Rogers", "Captain America", "Brooklyn", 99),
        ("Thor", "Odinson", "Thor", "Asgard", 1500)]

schema = StructType([
    StructField("firstname", StringType(), True),
    StructField("lastname", StringType(), True),
    StructField("alias", StringType(), True),
    StructField("location", StringType(), True),
    StructField("age", IntegerType(), True)
])

# Create initial dataframe and write to Delta table
initial_df = spark.createDataFrame(data=data, schema=schema)
initial_df.write.format("delta").mode("overwrite").save("data/delta-
avengers")
```

Step 2: Evolving the Schema by Adding a New Column

We now add a new column `weapon` to the table and enable schema evolution when appending this data.

```
# New data with the 'weapon' column added
new_data = [("Natasha", "Romanoff", "Black Widow", "Russia", 35, "Guns"),
            ("Clint", "Barton", "Hawkeye", "New York", 42, "Bow and Arrow")]

# New schema including the 'weapon' column
new_schema = StructType([
    StructField("firstname", StringType(), True),
    StructField("lastname", StringType(), True),
    StructField("alias", StringType(), True),
    StructField("location", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("weapon", StringType(), True)  # New column 'weapon'
])

# Create new dataframe with the updated schema
new_data_df = spark.createDataFrame(data=new_data, schema=new_schema)
```

```
# Append new data to Delta table with schema evolution enabled
new_data_df.write.format("delta") \
    .mode("append") \
    .option("mergeSchema", "true") \
    .save("data/delta-avengers")
```

Step 3: Verifying the Updated Schema

You can verify that the schema has been updated and the new data with the `weapon` column has been successfully appended to the Delta table.

```
# Read the entire Delta table to see both old and new data
updated_df = spark.read.format("delta").load("data/delta-avengers")
updated_df.printSchema()  # Verify that the 'weapon' column is included
updated_df.show()         # Show the data with and without 'weapon'
```

Step 4: Handling Schema Enforcement

If you try to append data that doesn't match the schema without enabling schema evolution (`mergeSchema=true`), Delta Lake will throw an error. By enabling `mergeSchema`, it automatically handles the schema update.

5. Architecture

The Delta Lake Architecture is a massive improvement upon the conventional Lambda architecture.
At each stage, it improves our data through a connected pipeline and allows us to combine streaming and batch workflows through a shared file store with ACID-compliant transactions.

It organizes our data into layers or folders defined as bronze, silver, and gold as follows…

- Bronze tables have raw data ingested from various sources (RDBMS data, JSON files, IoT data, etc.)
- Silver tables will give a more refined view of our data using joins.
- Gold tables give business-level aggregates often used for dashboarding and reporting.

And these Gold Tables can be consumed by various Business Intelligence tools for reporting and analytics purposes.

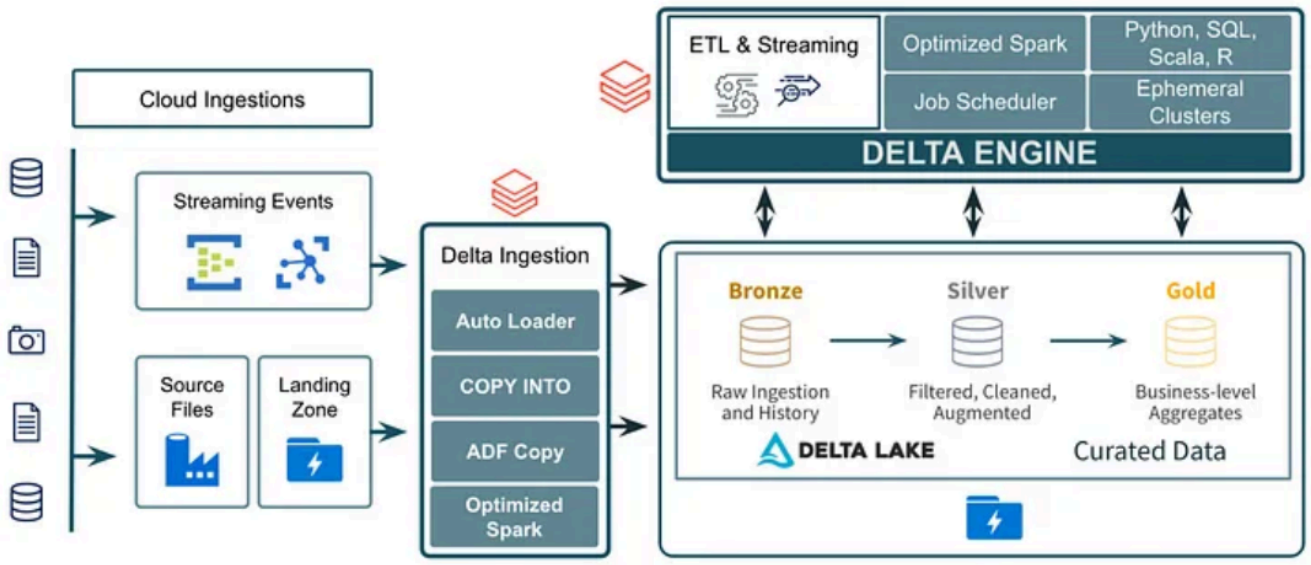
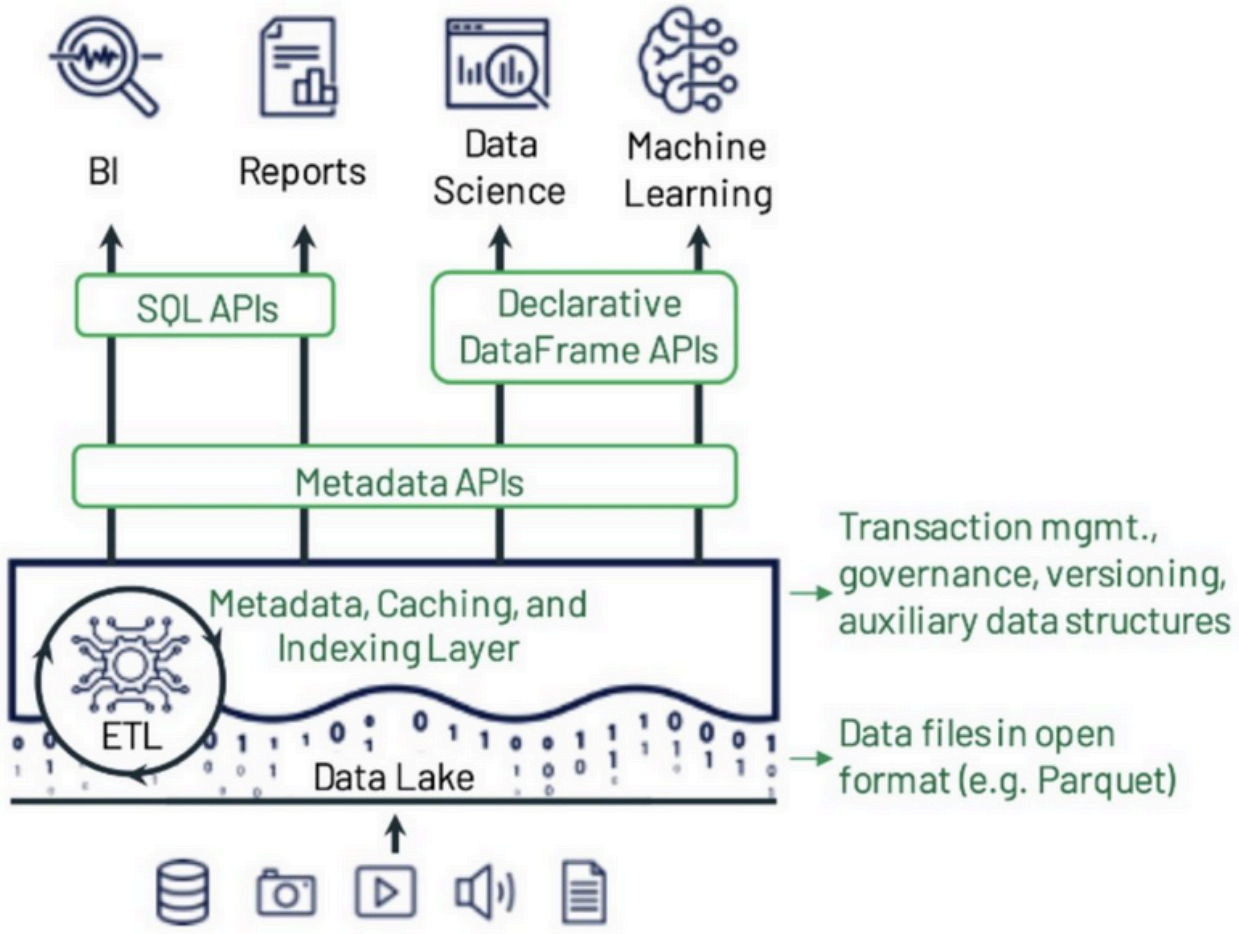


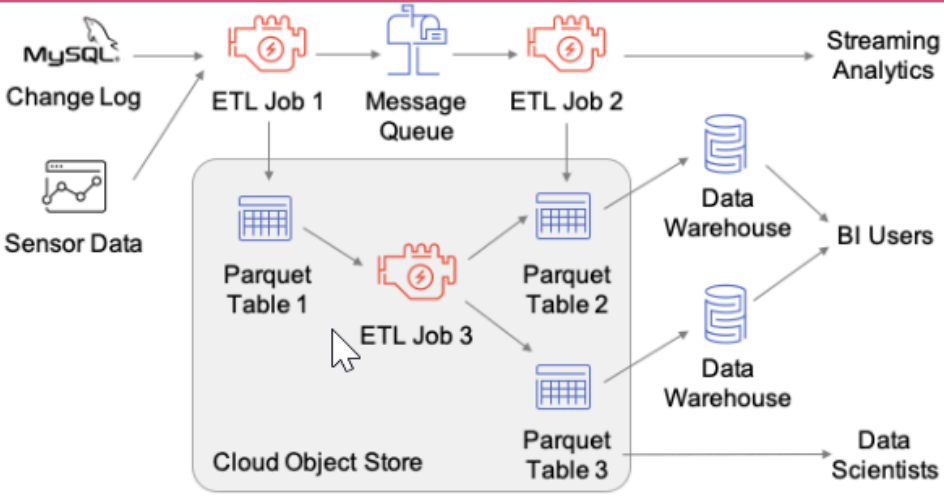
Image Courtesy techcommunity.microsoft.com



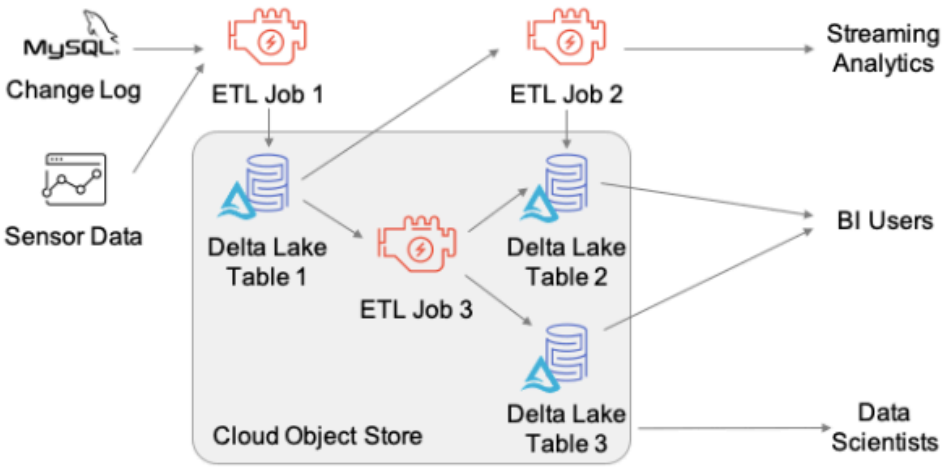
Key architectural components:

1. Storage Layer: Built on top of existing data lake storage (e.g., HDFS, S3)
2. Transaction Log: Keeps track of all changes to the table
3. Parquet Files: Stores the actual data in columnar format

4. Optimized Layouts: Includes data skipping and Z-Ordering for improved performance
5. Metadata Handling: Efficiently manages large-scale metadata
6. Spark Integration: Tightly integrated with Apache Spark for processing
7. API Layer: Provides interfaces for reading, writing, and managing Delta tables
8. Optimization Engine: Includes features like compaction and indexing for better performance



(a) Pipeline using separate storage systems.



(b) Using Delta Lake for both stream and table storage.

Figure 1: A data pipeline implemented using three storage systems (a message queue, object store and data warehouse), or using Delta Lake for both stream and table storage. The Delta Lake version removes the need to manage multiple copies of the data and uses only low-cost object storage.

6. Features

Delta Lake offers a rich set of features:

1. **ACID Transactions:** Ensures data consistency and reliability.
2. **Scalable Metadata Handling:** Efficiently manages metadata for large tables.
3. **Time travel**: Enables users to query point-in-time snapshots or roll back erroneous updates to their data.
4. **UPSERT, DELETE, and MERGE operations**: Efficiently rewrite relevant objects to implement updates to archived data and compliance workflows (e.g., for GDPR [27]).
5. **Efficient streaming I/O**: Allows streaming jobs to write small objects into the table at low latency, then transactionally coalesce them into larger objects for performance. Supports fast "tailing" reads of new data added to the table, enabling jobs to treat a Delta table as a message bus.
6. **Caching**: Since Delta table objects and logs are immutable, cluster nodes can cache them on local storage. In the Databricks cloud service, this is used to implement a transparent SSD cache for Delta tables.
7. **Data layout optimization**: The cloud service automatically optimizes the size of objects in a table and clusters data records (e.g., storing records in Z-order for multi-dimensional locality) without affecting running queries.
8. **Schema evolution**: Delta can continue reading old Parquet files without rewriting them when a table's schema changes.
9. **Audit logging**: Based on the transaction log.

7. Benefits

The adoption of Delta Lake brings numerous benefits:

1. Improved Data Quality: Ensures data integrity and consistency.
2. Enhanced Performance: Speeds up queries through optimized data layout and indexing.
3. Simplified Data Pipeline: Unifies batch and streaming processes.
4. Reduced Management Overhead: Automates many aspects of data management.
5. Increased Reliability: Provides ACID transactions and versioning.
6. Better Governance: Offers audit trails and rollback capabilities.
7. Scalability: Handles large-scale data and metadata efficiently.
8. Flexibility: Supports various data formats and processing paradigms.

8. Use Cases

1. **ETL Pipelines**: Delta Lake is widely used in extract-transform-load (ETL) workflows to ingest, process, and transform large datasets.
2. **Machine Learning**: It enables ML workflows by providing a consistent, reliable source of data for model training.
3. **Financial Transactions**: Delta Lake's ACID compliance is crucial for financial services that require transaction-level integrity.
4. **Real-Time Analytics**: Organizations using real-time dashboards leverage Delta Lake for streaming and batch analytics.

5. **Regulatory Compliance**: Businesses in regulated industries (healthcare, finance) use Delta Lake to maintain a reliable, auditable trail of data modifications.
6. **Data Lakes in the Cloud**: Enterprises are building Delta Lake on cloud platforms to scale their data architecture with reliable performance.
7. **Data Warehousing**: It provides the reliability of a data warehouse with the flexibility of a data lake, suitable for business analytics.
8. **Media and Entertainment**: Delta Lake supports large-scale media streaming data processing and analysis.

9. Summary

Delta Lake is revolutionizing data architectures by combining the scalability of traditional data lakes with the reliability and performance of data warehouses. Its key features, such as ACID transactions, time travel, schema enforcement, and real-time/batch unification, solve the common challenges businesses face with big data. As companies move toward more data-driven decision-making, Delta Lake is a crucial tool in ensuring data consistency, improving query performance, and lowering infrastructure costs.

`Resources:`
https://medium.com/@siladityaghosh/introduction-to-delta-lake-2d028081220c
https://medium.com/@ansabiqbal/delta-lake-introduction-with-examples-using-pyspark-cb2a0d7a549d
Deltalake: https://www.databricks.com/wp-content/uploads/2020/08/p975-armbrust.pdf
Lakehouse: https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

Delta Lake Time Travel

Delta Lake's **Time Travel** feature allows you to query, restore, and audit historical data versions, enabling you to "go back in time" and retrieve data from a previous state. This can be extremely useful for auditing, troubleshooting, or simply recovering data from an earlier version.

Below are several code examples showing how to use **Time Travel** in Delta Lake:

1. Writing Initial Data

First, we'll write the Avengers data to a Delta table to create some versions.

```
# Writing the initial Avengers data to Delta table
data = [("Tony", "Stark", "Iron Man", "New York", 48),
        ("Steve", "Rogers", "Captain America", "Brooklyn", 99),
        ("Thor", "Odinson", "Thor", "Asgard", 1500)]
```

```
schema = StructType([
    StructField("firstname", StringType(), True),
    StructField("lastname", StringType(), True),
    StructField("alias", StringType(), True),
    StructField("location", StringType(), True),
    StructField("age", IntegerType(), True)
])

avengers_df = spark.createDataFrame(data, schema=schema)
avengers_df.write.format("delta").mode("overwrite").save("data/delta-
avengers")
```

2. Updating the Delta Table (Creating Versions)

Let's create additional versions by updating the Delta table.

```
# Writing new data (creating a new version)
new_data = [("Natasha", "Romanoff", "Black Widow", "Russia", 35),
            ("Clint", "Barton", "Hawkeye", "New York", 42)]

new_data_df = spark.createDataFrame(new_data, schema=schema)
new_data_df.write.format("delta").mode("append").save("data/delta-avengers")

# Further update (creating another version)
additional_data = [("Bruce", "Banner", "Hulk", "New York", 40),
                   ("Wanda", "Maximoff", "Scarlet Witch", "Sokovia", 30)]

additional_data_df = spark.createDataFrame(additional_data, schema=schema)
additional_data_df.write.format("delta").mode("append").save("data/delta-
avengers")
```

At this point, there are multiple versions of the Delta table stored (Version 0, 1, 2). Now we can perform **Time Travel** queries on these versions.

3. Querying Previous Versions Using `versionAsOf`

You can query previous versions of the Delta table by specifying the version number using `versionAsOf`.

```
# Query the Delta table as it was at version 0 (initial data)
version_0_df = spark.read.format("delta").option("versionAsOf",
0).load("data/delta-avengers")
version_0_df.show()

# Query the Delta table as it was at version 1 (after the first update)
version_1_df = spark.read.format("delta").option("versionAsOf",
```

```
1).load("data/delta-avengers")
version_1_df.show()
```

4. Querying Previous Versions Using `timestampAsOf`

You can also query the Delta table as it was at a specific timestamp. To use this, you'll need the timestamp when the version was created.

```
# Query the Delta table using a timestamp
# Example timestamp format: "yyyy-MM-dd HH:mm:ss"
timestamp_df = spark.read.format("delta").option("timestampAsOf", "2024-09-12
10:00:00").load("data/delta-avengers")
timestamp_df.show()
```

5. Restoring a Delta Table to a Previous Version

You can restore a Delta table to a previous version using the `restoreToVersion` method.

```
from delta.tables import *

# Load the Delta table
delta_table = DeltaTable.forPath(spark, "data/delta-avengers")

# Restore the Delta table to version 0 (initial state)
delta_table.restoreToVersion(0)

# Verify that the table is restored to the initial version
restored_df = spark.read.format("delta").load("data/delta-avengers")
restored_df.show()
```

6. Auditing Changes (Transaction History)

You can view the transaction history to see how the Delta table has evolved over time. This helps in auditing changes.

```
# Get the transaction history of the Delta table
delta_table = DeltaTable.forPath(spark, "data/delta-avengers")

# Show the entire history of the table
history_df = delta_table.history()  # Default: last 30 versions
history_df.show(truncate=False)

# Show the last 2 operations
delta_table.history(2).show(truncate=False)
```

7. Deleting Data and Time Travel to Restore

You can use **Time Travel** to recover accidentally deleted data.

```
# Delete all rows where the location is 'New York'
delta_table.delete("location = 'New York'")

# Verify deletion
deleted_df = spark.read.format("delta").load("data/delta-avengers")
deleted_df.show()

# Time travel to restore data by querying a previous version
restored_df = spark.read.format("delta").option("versionAsOf",
1).load("data/delta-avengers")
restored_df.show()
```

8. Combine Time Travel with Queries

You can combine Time Travel with other Spark queries to filter and manipulate the data from past versions.

```
# Query version 0 and filter Avengers who are older than 50
filtered_df = spark.read.format("delta").option("versionAsOf",
0).load("data/delta-avengers").filter("age > 50")
filtered_df.show()
```

Summary of Techniques:

- **Version-based Querying**: Use `versionAsOf` to query specific versions of a Delta table.
- **Timestamp-based Querying**: Use `timestampAsOf` to query the table at a specific point in time.
- **Restoring**: Restore a table to a previous version using `restoreToVersion`.
- **Auditing**: Use `history()` to see the transaction history for a table and track changes.
- **Recovery**: You can recover data by querying or restoring past versions.

These Time Travel techniques in Delta Lake allow for effective data auditing, recovery, and exploration of historical datasets.

Delta Lake optimization

Delta Lake offers several optimization techniques to enhance performance, especially when dealing with large datasets. Below are code examples demonstrating some of the key

optimization techniques:

1. File Compaction (Optimize)

File compaction helps merge small files into larger ones, which improves query performance and reduces metadata overhead.

```
# Load the Delta table
from delta.tables import *

delta_table = DeltaTable.forPath(spark, "data/delta-avengers")

# Optimize the Delta table by compacting small files
delta_table.optimize().executeCompaction()

# Verify the compaction by counting the number of files before and after
pre_compaction_count = len(dbutils.fs.ls("data/delta-avengers"))
print(f"Number of files before compaction: {pre_compaction_count}")

# Perform compaction
delta_table.optimize().executeCompaction()

post_compaction_count = len(dbutils.fs.ls("data/delta-avengers"))
print(f"Number of files after compaction: {post_compaction_count}")
```

2. Z-Order Clustering

Z-Ordering helps in optimizing the data layout by co-locating related information in the same set of files. This improves the performance of certain queries that filter on specific columns (e.g., `location`).

```
# Perform Z-Order clustering on the 'location' column to optimize query
performance
delta_table.optimize().zOrderBy("location").executeCompaction()
```

Z-Order clustering is especially useful for queries where you filter by specific columns frequently, as it minimizes the amount of data scanned by those queries.

3. Data Skipping

Data skipping allows Delta Lake to avoid scanning irrelevant files when querying data, which significantly reduces read times. This is automatically enabled in Delta Lake, but we can write queries to take advantage of it.

```
# Create some new data with different locations
data = [("Bruce", "Banner", "Hulk", "New York", 40),
        ("Wanda", "Maximoff", "Scarlet Witch", "Sokovia", 30),
        ("Peter", "Parker", "Spider-Man", "Queens", 18),
        ("T'Challa", "Black Panther", "Wakanda", 35)]

# Write new data to the Delta table
new_data_df = spark.createDataFrame(data, schema=schema)
new_data_df.write.format("delta").mode("append").save("data/delta-avengers")

# Perform a query that benefits from data skipping
ny_avengers = spark.read.format("delta").load("data/delta-
avengers").filter("location = 'New York'")
ny_avengers.show()
```

4. Partitioning

Partitioning splits your data based on specific columns, allowing more efficient queries by limiting the amount of data that needs to be scanned. Partitioning can be applied when writing data to Delta Lake.

```
# Partition Delta table by 'location' to improve query performance
new_data_df.write.format("delta").mode("overwrite").partitionBy("location").s
ave("data/delta-avengers-partitioned")

# Read only the partition for 'New York'
ny_avengers_partitioned = spark.read.format("delta").load("data/delta-
avengers-partitioned").filter("location = 'New York'")
ny_avengers_partitioned.show()
```

5. Caching

Delta Lake supports caching frequently queried data, which helps in reducing I/O costs and speeds up query performance.

```
# Cache the Avengers Delta table in memory
cached_df = spark.read.format("delta").load("data/delta-avengers").cache()

# Perform operations on the cached DataFrame
cached_df.show()

# Caching is particularly helpful when performing multiple operations on the
same dataset
cached_df.filter("age > 50").show()  # Query from the cached table
```

6. Vacuum (Removing Old Data)

The `VACUUM` command in Delta Lake helps clean up old files that are no longer needed (such as those from previous versions or deleted data). This reduces storage costs and improves performance by eliminating unnecessary files.

```
# Run VACUUM to clean up old files that are no longer needed
delta_table.vacuum(retentionHours=168)  # Retain last 7 days of data

# After vacuum, the table is cleaned up and storage is optimized
```

7. Auto-Optimize (Delta Lake Databricks Feature)

On Databricks, Delta Lake supports **auto-optimize** for automatically compacting small files and optimizing data layout.

```
# Enable auto-optimize feature on Delta table (requires Databricks)
spark.sql("""
    ALTER TABLE delta.`/data/delta-avengers`
    SET TBLPROPERTIES (
        'delta.autoOptimize.optimizeWrite' = true,
        'delta.autoOptimize.autoCompact' = true
    )
""")
```

Summary

These optimization techniques help ensure that Delta Lake performs efficiently, especially for large datasets:

- **Compaction** and **Z-Ordering** improve query performance by optimizing file layouts.
- **Partitioning** and **Data Skipping** reduce the amount of data scanned during queries.
- **Caching** speeds up repeated queries on frequently accessed data.
- **Vacuum** helps reclaim storage space by removing old, unused files.