COMPATIBILITY OF DELTA LAKE FOR SINGLE-NODE BIG DATA SYSTEM.

Avyuthan Shah

July 16,2024

LIST OF ABBREVIATIONS

DML: Data Manipulation Language

ACID: Atomicity, Consistency, Isolation, Durability

ORC: Optimized Row Columnar

OCC: Optimistic Concurrency Control’

OLTP: Online Transaction Processing

OLAP: Online Analytical ProcessingS

TABLE OF CONTENTS

LIST OF ABBREVIATIONS....2
CHAPTER 1: BACKGROUND....4
1.1 Introduction....4
1.2 Overview....9
1.3 Objective....9
1.4 Performance Evaluation....9
1.5 Use Cases....10
1.6 Problem Statement....10
CHAPTER 2: LITERATURE REVIEW....11
CHAPTER 3: METHODOLOGY....12
3.1 Delta Architecture....12
3.2 Setting Up....14
3.3 Project Initialization....16
3.4 Integration with Apache Spark in Local Mode....19
3.5 Data Collection, Preparation and Ingestion: Methods for Delta Lake....22
3.6 Delta Features....35
3.7 Integration with Jupyter Notebook and Hadoop....54
CHAPTER 4: FINDINGS....56
4.1 Performance analysis of data ingestion into delta....56
4.2 Performance analysis of Delta Features....58
4.3 Delta Optimization Benchmarking (Z-order,compaction,optimized write)....61
CHAPTER 5: DISCUSSIONS AND SUGGESTIONS....63
5.1 Compatibility and performance of Delta Lake in a single-node environment....63
5.2 Limitations and Suggestions....66
CONCLUSION....68
APPENDIX....69
1. Concurrent Write Simulation With Retry Logic....69
2. Loading 3 datasets(Banktransaction116k, sms data, bank statement) into delta....71
3. Errors....75
REFERENCES....79
TASK SUMMARY....80

CHAPTER 1: BACKGROUND

1.1 Introduction

Delta Lake emerges as a robust open-format storage layer in modern data management, aimed to improve reliability, security, and performance over previous data lakes. Delta Lake, which can handle both streaming and batch operations, offers a comprehensive solution for a wide range of data processing requirements. Delta Lake serves as a storage layer on top of existing data lakes, managing a wide range of raw data formats such as Parquet and CSV while also providing advanced metadata management. It supports DML operations, which are required for effective data maintenance and updates. As the cornerstone of a cost-effective and highly scalable Lakehouse architecture, Delta Lake blends the flexibility of data lakes with the solid performance features of data warehouses.

Delta Lake's architecture is arranged into three separate layers—Bronze, Silver, and Gold—with each playing a specialized purpose in data processing and storage.

- **Bronze Layer:** This basic layer accepts raw data from many sources and stores it in its original format. The primary focus is on data input and initial storage, which ensures that raw data is easily accessible for further processing. The Bronze layer often handles organized, semi-structured, and unstructured data with minimal change.
- **Silver Layer:** In the Silver layer, data is cleaned, transformed, and enhanced. This layer focuses on improving data quality and consistency, preparing it for analytical queries and subsequent processing. The Silver layer uses Delta Lake's transactional capabilities to assure data integrity and dependability during these transformations.
- **Gold Layer:** The Gold layer contains refined, business-critical data. This layer comprises consolidated, carefully filtered data designed for business intelligence (BI) and analytics applications. Data in the Gold layer is frequently designed to facilitate high-performance queries and reporting, which provides useful insights for decision-making processes.

Delta Lake's adaptability encompasses across many data types—structured, semi-structured, and unstructured—which are all efficiently managed.

- **Structured Data:** Delta Lake excels in structured data, offering ACID transactions, schema enforcement, and effective metadata management. This offers consistent and dependable storage, with optimization using columnar file formats such as Parquet or ORC, allowing for efficient storage and high-performance queries.
- **Semi-Structured Data:** Delta Lake is adept at handling semi-structured data types such as JSON and Avro. It saves schema information and facilitates schema evolution, allowing for modifications over time. While semi-structured data may require additional processing due to its variable schema, Delta Lake's transactional capabilities ensure consistency and stability.
- **Unstructured Data:** Delta Lake effectively saves references to binary objects and their metadata, even if it does not directly handle the internal structure of unstructured data. Text documents, pictures, and other binary data formats fall under this category. For processing unstructured data, specialized tools and frameworks are frequently

used. Delta Lake's metadata management and transactional features are used for effective storage and retrieval.

By integrating the strengths of data warehouses with the scalability of data lakes, Delta Lake embodies the Lakehouse architecture, offering a unified platform that meets diverse data needs. This combination allows enterprises to harness the flexibility of data lakes while benefiting from the reliability and performance of data warehouses, positioning Delta Lake as a pivotal component in contemporary data ecosystems.

1.1.1 Key Features

a. ACID Transaction:

Maintaining data integrity in a typical data lake environment is critical owing to concurrent read and write activities by various users. Unlike traditional databases, which provide strong ACID guarantees, storage technologies such as HDFS or S3 struggle to match the same durability. Delta Lake overcomes this issue by creating a transaction log that records all commits to the table directory, allowing ACID transactions. This technique maintains data consistency and reliability over multiple operations. Delta Lake also features serializable isolation layers, which assure consistent data access and integrity, making it an excellent choice for current data lake systems.

Transactions are processed in a manner that ensures all or none of the operations within the transaction are applied. This guarantees data reliability and consistency, even in the presence of concurrent reads and writes.

- **Atomicity:** Transactions are atomic, meaning they are either fully completed or fully rolled back in case of failure. Atomicity means that a transaction (a series of operations) is all-or-nothing. Imagine you are transferring money from your bank account to a friend's account. This transaction involves two steps: deducting money from your account and adding it to your friend's account. Atomicity ensures that either both steps happen completely or none happen at all. If something goes wrong during the process, like a power failure, the transaction will be rolled back, and your money will remain in your account.
- **Consistency:** Transactions maintain consistency constraints defined by the schema, ensuring data quality. A transaction must preserve the consistency of the underlying data. The transaction should make no changes that violate the rules or constraints placed on the data. For instance, a database that supports banking transactions might include a rule stating that a customer's account balance can never be a negative number. If a transaction attempts to withdraw more money from an account than what is available, the transaction will fail, and any changes made to the data will roll back.
- **Isolation:** Transactions are isolated from each other, preventing interference between concurrent transactions. Delta Lake provides isolation between concurrent transactions. It uses OCC to allow multiple transactions to proceed concurrently. Each transaction operates on a consistent snapshot of the data, preventing interference between transactions.
    - OCC is a technique used to manage situations where multiple users or processes attempt to modify the same data simultaneously. It operates on an optimistic approach, allowing users to make changes freely without immediately locking the data. Instead of blocking access to the data, OCC tracks different versions of the data as changes are made. When users attempt to save their changes, the system checks for conflicts by comparing the different versions of the data. If conflicts are detected, they need to be resolved before the changes can be applied. However, if

no conflicts occur, the changes can be merged seamlessly. OCC is analogous to editing a shared document online, where multiple users can make changes independently, and the system handles conflicts by merging the changes together.

Isolation Levels

a. Snapshot Isolation:

- **Description**: This isolation level ensures that all reads within a transaction see a consistent snapshot of the data, which is the state of the data at the beginning of the transaction. This means that a transaction will not see any changes made by other transactions that committed after it started.
- **Usage**: Snapshot Isolation is typicMaintaining data integrity in a typical data lake environment is critical owing to concurrent read and write activities by various users. Unlike traditional databases, which provide strong ACID guarantees, storage technologies such as HDFS or S3 struggle to match the same durability. Delta Lake overcomes this issue by creating a transaction log that records all commits to the table directory, allowing ACID transactions. This technique maintains data consistency and reliability over multiple operations. Delta Lake also features serializable isolation layers, which assure consistent data access and integrity, making it an excellent choice for current data lake systems.ally the default mode in Delta Lake. It is used to handle most read and write operations efficiently, ensuring that readers do not block writers and vice versa.

b. Serializable Isolation:

- **Description**: This is the highest isolation level, ensuring that transactions are executed in a way that is equivalent to having them run one after the other serially. This eliminates the possibility of concurrent updates causing inconsistencies.
- **Usage**: Serializable Isolation is crucial for use cases that require strict consistency, such as financial transactions where concurrent updates must not result in data anomalies.

Isolation Level	Dirty Reads	Non-repeatable Reads	Phantom Reads
Snapshot Isolation	Prevented	Prevented	Allowed
Serializable	Prevented	Prevented	Prevented

- **Durability:** Once a transaction is committed, its changes are durable and persistent, even in the event of system failures.

b. Schema Enforcement and Evolution:

Uses schema-on-write method meaning it validates incoming schema against a predefined schema written to storage. The merge operation in Delta Lake does not automatically update the schema of the Delta table.

The merge operation is primarily used for upserts, allowing to conditionally insert, update, or delete records based on specified conditions. When using the merge operation to merge data with a Delta table, it assumes that the schema of the incoming data matches the schema of the Delta table (oldData). If there are new columns in the incoming data that are not present in the Delta table, they will be ignored during the merge op eration.

To handle schema evolution, one needs to explicitly specify to merge the schema of the incoming data with the schema of the Delta table. It can be achieved by setting the mergeSchema option to true when writing the new data to the Delta table.

c. Time Travel:

Delta Lake provides time travel capabilities, allowing users to query historical versions of data. This is achieved through data versioning, where each change to the data is recorded as a new version. Time travel enables auditing, debugging, and reproducing experiments by providing access to data snapshots at different points in time. Users can query data as it existed at specific timestamps or versions, facilitating historical analysis and troubleshooting

d. Unified Batch and Stream Processing:

Delta Lake enables batch and real-time streaming data processing, providing flexibility in handling various types of data operations. Delta Lake provides a unified platform for both batch processing and real-time data streams. This allows enterprises to create strong data pipelines that seamlessly connect batch and streaming data processing, easing the development and implementation of data-driven applications.

e. Z-Ordering

Regular Parquet data lakes are already optimized for columnar query optimizations and row-group skipping. Delta Lake takes this further by storing similar data close together via Z-ordering. Z-ordering is particularly important when you are querying multiple columns.

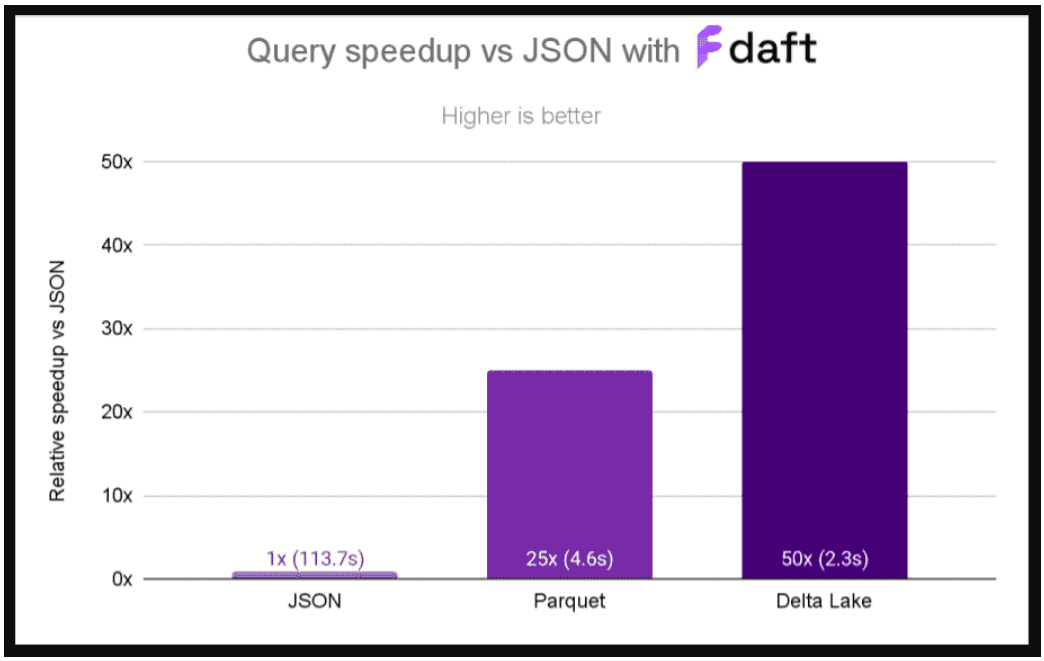


Figure: Query Execution Speed Comparison json vs parquet vs Delta Lake [1]

Delta Lake's Z-ORDER BY feature is utilized for maintaining multi-petabyte datasets in an information security use case, allowing efficient querying along multiple dimensions without the need for heavyweight index structures.

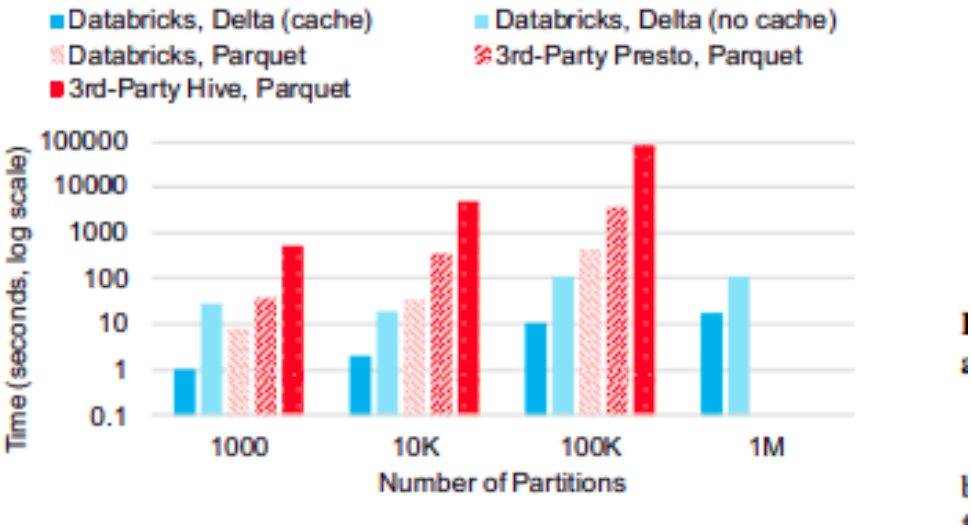


Figure: Performance querying a small table with a large number of partitions in various systems. The non-Delta systems took over an hour for 1 million partitions so we do not include their results there [2]

Z-Ordering is a technique to co-locate related information in the same set of files. This co-locality is automatically used by Delta Lake in data-skipping algorithms. This behavior

dramatically reduces the amount of data that Delta Lake on Apache Spark needs to read. To Z-Order data, you specify the columns to order on in the ZORDER BY clause:

1.2 Overview

Developed at Databricks, Delta Lake is an open-source storage layer that enables high-performance table storage over cloud object stores like Amazon S3 and supports ACID (Atomicity, Consistency, Isolation, Durability) transactions. In contrast to standard cloud object stores, which struggle to build key-value stores and achieve ACID transactions while maintaining high speed, Delta Lake gets around these issues by using creative design ideas.

Fundamentally, Delta Lake uses a compacted transaction log in Apache Parquet format to facilitate metadata operations and guarantee ACID characteristics for sizable tabular datasets. This transaction log offers mechanisms for data durability and consistency as well as time travel, allowing users to access and analyze earlier versions of the data. Delta Lake also improves its usability and reliability in data-intensive environments with features like audit logs, caching, upserts, and intelligent data layout optimization.

Furthermore, Delta Lake provides seamless integration with a variety of data processing frameworks, including Apache Spark, Hive, Presto, and Redshift, enabling organizations to leverage its capabilities within their existing analytics ecosystems. Utilized by thousands of Databricks clients, Delta Lake handles enormous datasets and billions of objects on a daily basis, giving it a strong answer to today's complex data management problems.

1.3 Objective

- Evaluate the ease of deployment and management of Delta Lake on a single-node system.
- Load datasets referring to “Appendix I - Data Pipeline Tasks” in three databases.
- Assess the performance benefits and potential limitations of Delta Lake on a single-node setup.
- Identify use cases where Delta Lake on a single-node setup is advantageous.
- Explore integration capabilities with other tools and frameworks in a single-node environment.

1.4 Performance Evaluation

- Test and validate the robustness of ACID transactions in a single-node setup.
- Assess the ease and effectiveness of schema enforcement and evolution features.
- Evaluate the functionality and performance impact of the time travel feature.
- Test Delta Lake’s capability to handle both batch and streaming data in a single- node environment.

1.5 Use Cases

- Evaluate the suitability of Delta Lake for development and testing purposes on a single node.

- Identify specific small-scale applications where Delta Lake provides clear advantages.
- Assess the effectiveness of Delta Lake for data scientists and analysts working on local machines.

1.6 Problem Statement

- Document how Delta Lake integrates with Apache Spark in local mode.
- Explore compatibility and integration with other common tools and frameworks in a single-node environment (e.g., Jupyter notebooks, Hadoop).
- Summarize the key findings regarding the compatibility and performance of Delta Lake in a single-node environment.
- Highlight the primary benefits and any notable limitations.
- Provide actionable recommendations based on the research findings.

CHAPTER 2: LITERATURE REVIEW

Delta Lake and lakehouse architecture are two important subjects that are explored in the literature when it comes to data management and analytics.

The problems with cloud object stores like Amazon S3 are addressed by Delta Lake, an open-source ACID table storage layer. Due to their key-value store implementation, these stores fail to achieve high performance and ACID transactions, despite their large storage capacity and affordability. With the help of a transaction log that has been compressed into the Apache Parquet format, Databricks' Delta Lake ensures ACID characteristics and speeds up metadata operations for big tabular datasets. This system includes important features including automatic data layout optimization, upserts, caching, and audit logs in addition to facilitating faster query processing. Delta Lake offers the ability to integrate Apache Spark, Hive, Presto, and other systems [3].

Many thousands of Databricks clients use Delta Lake to manage billions of items and exabyte-scale datasets.On the other hand, the Lakehouse architecture is being examined in the literature as a possible paradigm change in data warehouse design. With strong support for workloads related to machine learning and data science, and open direct-access data formats like Apache Parquet, Lakehouse design anticipates the demise of traditional data warehouse architectures. Data warehousing has evolved from centralized repositories for business analytics and decision assistance to contemporary data lakes, which emphasizes the necessity of scalability, economy, and flexibility [4].

Although the introduction of cloud data lakes, which have replaced older systems such as Hadoop File System (HDFS), has improved durability and decreased costs, data management has become more complex because of the coexistence of data lakes and downstream data warehouses. The industry's common two-tier architecture causes delays and complicates procedures, particularly for sophisticated analytics use cases like machine learning.

The literature analysis concludes by shedding light on Delta Lake's transformational potential in resolving cloud object stores' shortcomings and the rise of Lakehouse architecture as a viable substitute for conventional data warehouse designs. It emphasizes scalability, performance, and adaptability in the age of big data and advanced analytics, underscoring the necessity for creative solutions to traverse the changing terrain of d atamanagement and analytics.

CHAPTER 3: METHODOLOGY

3.1 Delta Architecture

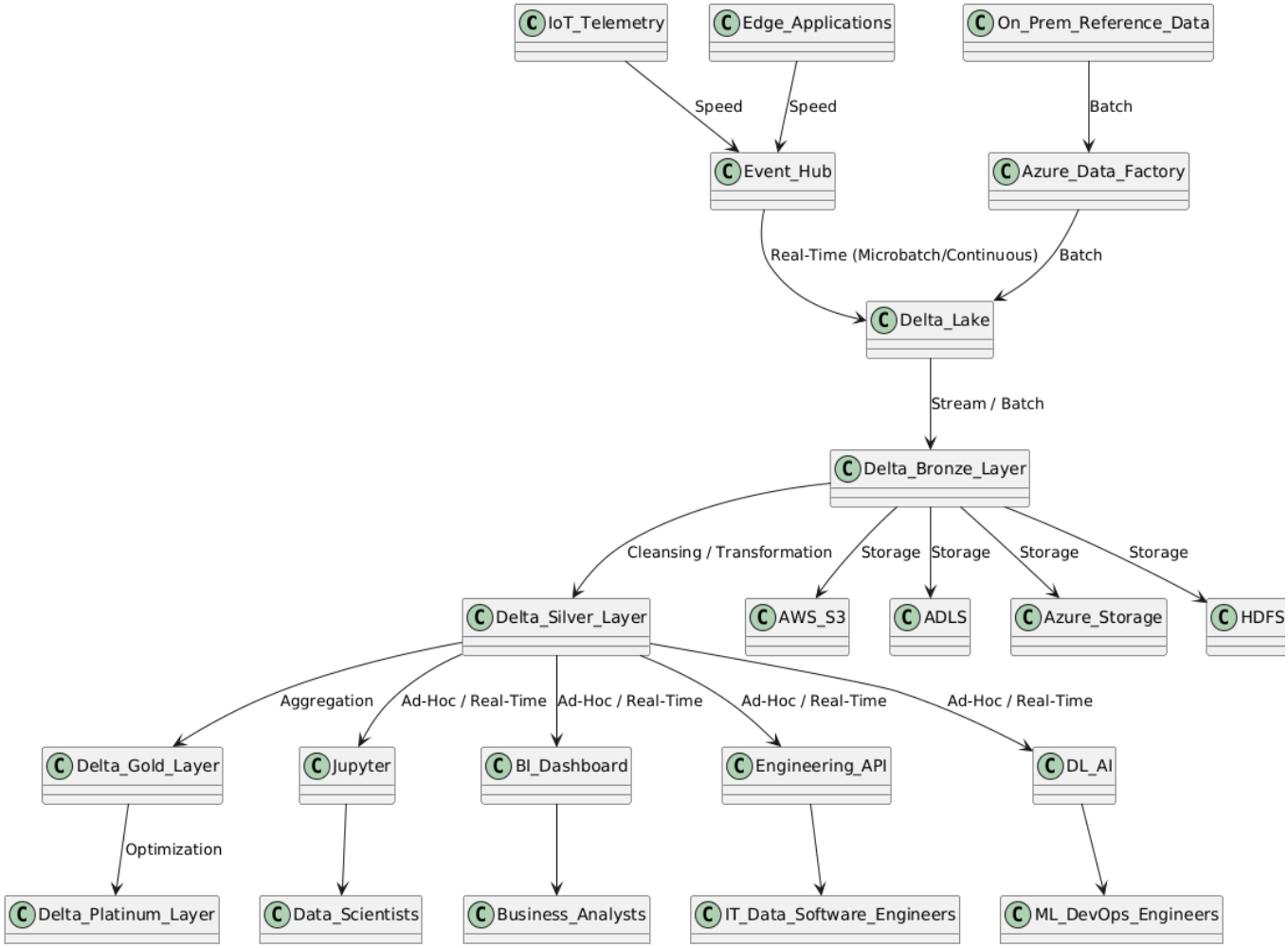


Figure: Delta Lake architecture

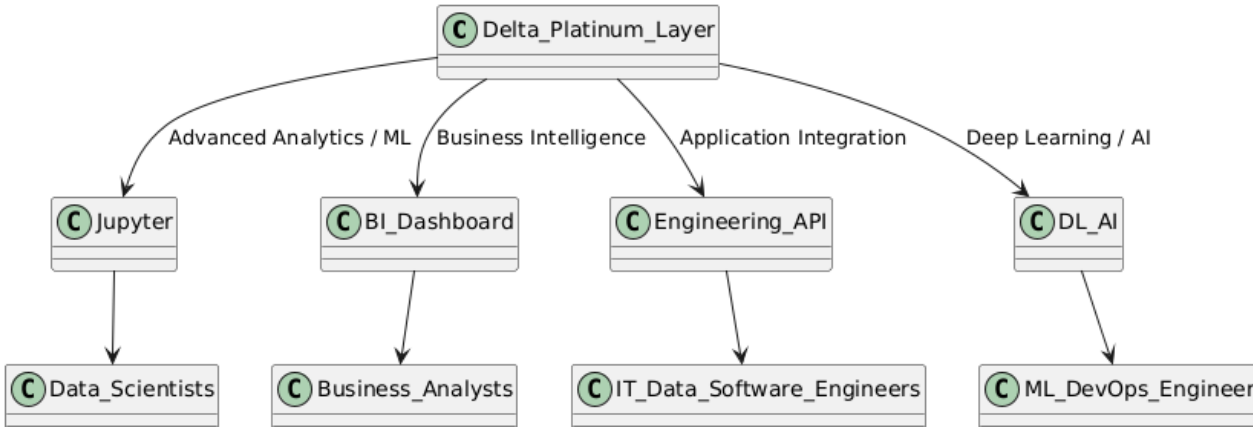


Figure: Data Serving and Consumption from the Delta Platinum Layer

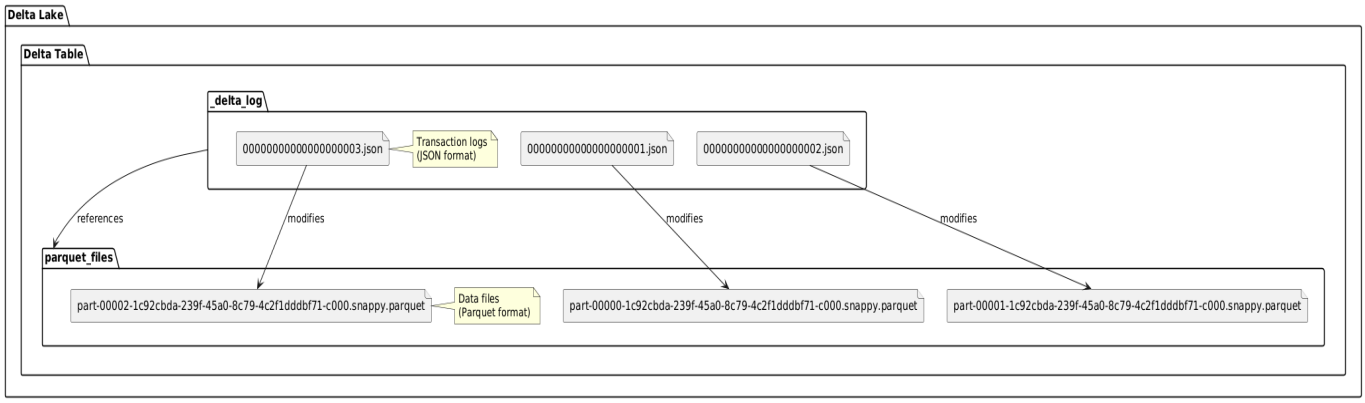


Figure: Delta Table File Structure with Transaction Log

The transaction log in Delta Lake serves as a crucial mechanism for maintaining data integrity and managing transactions. It records every write operation (inserts, updates, deletes) as metadata entries in JSON or Parquet format within the `_delta_log` directory. This log ensures ACID properties (Atomicity, Consistency, Isolation, Durability) by facilitating atomic commits and rollbacks, even in the event of failures. It also supports schema evolution, time travel queries, and concurrent data access through optimistic concurrency control. By logging transaction metadata alongside data changes, Delta Lake provides robust versioning capabilities and simplifies metadata management, making it reliable for both batch and streaming data workflows.

In Delta Lake, versioning is managed through its transaction log, which maintains a historical record of changes made to the data. When a write operation occurs, new data files are created and metadata about these changes is appended to the transaction log. Each transaction receives a unique transaction ID and commit timestamp, which are crucial for versioning. To fetch old versions of the data, Delta Lake queries the transaction log to identify the relevant data files associated with the desired version based on the specified timestamp or version number. It then reconstructs the data by reading these files, ensuring consistency and correctness based on the transaction metadata recorded during each commit. This process allows users to perform time travel queries, accessing data snapshots at different points in time without impacting the current state of the data, thus supporting historical analysis and auditing needs effectively within Delta Lake's ecosystem.

3.2 Setting Up

Github Link

https://github.com/avyuthanshah/ARL_Delta.git

3.2.1 Dependencies

Install JDK:

Ensure that JDK (Java Development Kit) is installed on your Linux system. You can install JDK using package managers like apt (for Ubuntu/Debian) or yum (for CentOS/RHEL).

To install OpenJDK 11 on Ubuntu, you can use the following command:

sudo apt install openjdk-11-jdk

Verify JDK Installation:

After installing JDK, verify the installation by running the following command:

java -version

This command should display the installed Java version. Make sure it shows the correct version of JDK that you installed (e.g., OpenJDK 11).

Add following to your ~/.bashrc

export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64

export PATH=$PATH:/usr/lib/jvm/java-11-openjdk-amd64/bin

Install Sdkman

SDKMAN! is a tool for managing parallel versions of multiple Software Development Kits on most Unix-based systems.

curl -s "https://get.sdkman.io" | bash

source "$HOME/.sdkman/bin/sdkman-init.sh"

source ~/.bashrc

This will add sdkman to your bashrc:

export SDKMAN_DIR="$HOME/.sdkman"

[[-s "$HOME/.sdkman/bin/sdkman-init.sh"]] && source "$HOME/.sdkman/bin/sdkman-init.sh"

Install Scala

sdk install sbt 1.10.0

sdk install scala 2.13.12 -> (for sbt 1.10.0) or 2.13.8 (for sbt 1.6.2)

#THIS MUST BE AT THE END OF THE FILE FOR SDKMAN TO WORK!!!

Install Spark

- wget (https:/spark download link Download 3.4.3 or 3.5.1

- cd /path_to_your_downloaded_folder
- sudo tar xvf ./scala-bin……
- Add following lines to bashrc

  SPARK_HOME=/opt/spark/spark-3.4.3-bin-hadoop or /path/to/installed/file

  export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/bin

IDE Setup

- Install Intellij or Vscode
- Add Plugins for Scala, sbt
- sbt version 1.10.0, or other versions

Delta Lake Compatibility with Scala

Check Mvn repo to find optimal versions and library dependencies

- https://mvnrepository.com/

Hardware Specifications:

- **CPU:** Intel Core i5-10700K (8 cores, 16 threads)
- **RAM:** 8 GB DDR4
- **Storage:** 256 GB NVMe SSD

Software Specifications:

- **Operating System:** Ubuntu 20.04 LTS or Any
- **Java Development Kit (JDK):** OpenJDK 11
- **Apache Spark:** Version 3.5.1
- **Delta Lake:** Version 3.1.0
- **Hadoop:** Version 3.3.6
- **sbt:** Version 1.10.0
- **Java:** Version 11
- **Scala:** Version 2.13.12

3.3 Project Initialization

cd <destination to your desired folder>

sbt new scala/scala-seed.g8 //to start new scala project in virtual env

```
your-project-name/
├── build.sbt
├── project/
│   ├── build.properties
│   └── plugins.sbt
└── src/
    └── main/
        └── scala/
            └── Main.scala (or any other Scala source files you create)
```

3.3.1 Built File

Add required libraries in built.sbt

LibraryDependencies:

For sbt build Use sbt format as specified in Maven repository.

```
import Dependencies._

ThisBuild / scalaVersion     := "2.13.12"
ThisBuild / version          := "0.1.0-SNAPSHOT"
ThisBuild / organization     := "com.example"
ThisBuild / organizationName := "example"

lazy val root = (project in file("."))
  .settings(
    name := "deltaF",
    libraryDependencies ++= Seq(
      "org.scala-lang" % "scala-library" % scalaVersion.value,
      // Spark Core library, provides the main Spark API
      "org.apache.spark" %% "spark-core" % "3.5.1",
      "org.apache.spark" %% "spark-sql" % "3.5.1",

      //Excel read write support
      "com.crealytics" %% "spark-excel" % "3.4.2_0.20.3",

      // Delta Lake support to Spark
      "io.delta" %% "delta-spark" % "3.1.0",

      //Sql Connector Support
      "mysql" % "mysql-connector-java" % "8.0.33",

      //Hadoop support Dependencies
      "org.apache.hadoop" % "hadoop-client" % "3.3.6",
```

```
    "org.apache.hadoop" % "hadoop-client-api" % "3.3.6",
    "org.apache.hadoop" % "hadoop-hdfs" % "3.3.6",
    "org.apache.hadoop" % "hadoop-mapreduce-client-core" % "3.3.6",

    //Monitor CPU and RAM performance
    "com.github.oshi" % "oshi-core" % "6.6.1",

      munit % Test
  )
 )

ThisBuild / javacOptions ++= Seq(
 "--release", "11" //Adjust according to version of java installed
)

ThisBuild / scalacOptions ++= Seq(
 "-release", "11" //Adjust according to version of java installed
)
```

Finally, Compile your project from terminal

sbt clean compile package

Clean:

- SBT deletes the target directory and its contents.
- This removes all previously compiled class files, JAR files, and other build artifacts.

Compile:

- SBT compiles the Scala (and Java) source files in your project.
- Compiled classes are placed in target/scala-2.13/classes (or similar, depending on your Scala version).

Package:

- SBT packages the compiled class files and resources into a JAR file.
- The JAR file is typically placed in target/scala-2.13/your-project-name_2.13-0.1.0-SNAPSHOT.jar (or similar).

```
your-project-name/
├── target/
│   └── scala-2.x/
│       └── classes/
│           └── Main.class
├── build.sbt
├── project/
│   ├── build.properties
│   └── plugins.sbt
└── src/
    └── main/
        └── scala/
            └── Main.scala (or any other Scala source files you create)
```

If you have scala source in certain folder inside src/main/scala/newFile, Then you should mention ` package newFile ` on top of your scala source

To Run:

sbt “runMain Main”

or if you have folder inside src/main/scala/example/csv2delta.scala

sbt “runMain example.csv2delta

3.4 Integration with Apache Spark in Local Mode

Note: Delta Lake is built on top of Apache Spark. So,spark integrates seamlessly with scala or python to work with delta.

Step 1: Initialize built.sbt for spark

Add library dependencies.

```
"org.apache.spark" %% "spark-core" % "3.5.1",
"org.apache.spark" %% "spark-sql" % "3.5.1",
```

Step 2: Import necessary libraries

```
import io.delta.tables.DeltaTable
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
```

Step 3: Initialize SparkSession

```
val spark = SparkSession.builder()
  .appName("Delta Connect")
  .master("local[*]") //Runs Spark Panel on localhost:4040
  .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
  .config("spark.sql.catalog.spark_catalog","org.apache.spark.sql.delta.catalog.DeltaCa
talog")
  .getOrCreate()
```

Step 4: Interact with Delta Using Spark

- **Read Methods**

  val delPath="/path/to/your/delta"

  - **Spark Api**

```
val dT:DataFrame=spark.read.format("delta")
  .option("treatEmptyValuesAsNulls", "true")
  .option("versionAdOf",0) //Specify any version for time travel or if not specifies it
fetches latest data
  .load(delPath)
```

```
spark.read.format("delta").option("versionAsOf",version).load(delPath).createOrReplaceTempView("delta_table")

val result = spark.sql(
   s"""
   |SELECT * FROM delta_table
   |WHERE AccountNo IN ("56135","69310","65260")
   |ORDER BY AccountNo,VALUEDATE DESC,TIME DESC;
   """.stripMargin)

result.show(truncate=false)
```

Figure: Query to Delta using spark sql

.createOrReplaceTempView("delta_table") → Creates or replaces a local temporary view with this **DataFrame**. The lifetime of this temporary table is tied to the **SparkSession** that was used to create this **DataFrame**.

- **Delta Api**

```
val deltaT=DeltaTable.forPath(spark,delPath)
```

They are generally used for writing to a delta table and is more efficient than spark method. Data can be ingested in more controlled way preventing duplicates and corruption.

- **Write Methods**
  - **Spark Api**

```
df.write
 .format("delta")
 .mode("overwrite")
 .save(delPath)
```

  - **Delta Api**

```
val deltaT=DeltaTable.forPath(spark,delPath)

deltaT.as("dt")
  .merge(
    df.as("nd"),
    "dt.AccountNo = nd.AccountNo AND dt.VALUEDATE = nd.VALUEDATE")
  .whenMatched().updateAll() //Note: To use updateAll the schema should match for both dataframe otherwise use updateExpr with Map() to map schema
  .whenNotMatched().insertAll()//Same condition as updateAll
  .execute()
```

Step 5: Stop Spark Session

```
spark.stop()
```

//It is a good practice to stop sparksession after use. This can help free up resources on the cluster or your local machine.

3.5 Data Collection, Preparation and Ingestion: Methods for Delta Lake

3.5.1 CRUD Operation in Delta

#Create

Creating a delta table directly is still in preview till now.

```
// Create table in the metastore
DeltaTable.createOrReplace(spark)
  .tableName("default.people10m")
  .addColumn("id", "INT")
  .addColumn("firstName", "STRING")
  .addColumn("middleName", "STRING")
  .addColumn(
    DeltaTable.columnBuilder("lastName")
      .dataType("STRING")
      .comment("surname")
      .build())
  .addColumn("lastName", "STRING", comment = "surname")
  .addColumn("gender", "STRING")https://docs.delta.io/latest/delta-batch.html#-ddlcreatetable&language-scala
  .addColumn("birthDate", "TIMESTAMP")
  .addColumn("ssn", "STRING")
  .addColumn("salary", "INT")
  .execute()

// Create or replace table with path and add properties
DeltaTable.createOrReplace(spark)
  .addColumn("id", "INT")
  .addColumn("firstName", "STRING")
  .addColumn("middleName", "STRING")
  .addColumn(
    DeltaTable.columnBuilder("lastName")
      .dataType("STRING")
      .comment("surname")
      .build())
  .addColumn("lastName", "STRING", comment = "surname")
  .addColumn("gender", "STRING")
  .addColumn("birthDate", "TIMESTAMP")
  .addColumn("ssn", "STRING")
  .addColumn("salary", "INT")
  .property("description", "table with people data")
  .location("/tmp/delta/people10m")
  .execute()
```

Source: [6]

#Read

Read operation is done as specified above in integration with apache spark. Spark api or DeltaTable api is used generally for reading a delta table.

#Update or Upserts

```
//Example Dataframe
val df= Seq(
  ("337777", "2024-06-06", "Deposit", "Null", "2024-06-06", 0.0, 300.0, 300.0),
  ("337777", "2024-06-09", "Withdraw", "Null", "2024-06-09", 100.0, 0.0, 200.0))

val dT = DeltaTable.forPath(spark, delPath)

//1 Update Everything automatically based on dataframe
dT.as("dt")
  .merge(
    df.as("nd"),
    "dt.AccountNo = nd.AccountNo AND dt.VALUEDATE = nd.VALUEDATE")
  .whenMatched().updateAll() //Note: To use updateAll the schema should match for both dataframe otherwise use updateExpr with Map() to map schema
  .whenNotMatched().insertAll()//Same condition as updateAll
  .execute()

//2 Update specified column
dT.update(
  condition = col("AccountNo") === accNo && col("BALANCEAMT") === balance, //balance is any double value or latest balance fetched from the table for given account number.
  set = Map("DEPOSITAMT" -> lit(balance+500), "BALANCEAMT" -> lit(balance + 500))
)
```

For efficient entity matching use:

For new dataframe you can explicitly map schema to original delta schema.

```
dT.as("dt")
  .merge(
    df.as("nd"),
    "dt.AccountNo = nd.AccountNo AND dt.VALUEDATE = nd.VALUEDATE"
  )
  .whenMatched()
  .updateExpr(
    Map(
      "AccountNo" -> "nd.AccountNo",
      "DATE" -> "nd.DATE",
      "TRANSACTIONDETAILS" -> "nd.TRANSACTIONDETAILS",
      "CHQNO" -> "nd.CHQNO",
      "VALUEDATE" -> "nd.VALUEDATE",
      "WITHDRAWALAMT" -> "nd.WITHDRAWALAMT",
```

```
    "DEPOSITAMT" -> "nd.DEPOSITAMT",
    "BALANCEAMT" -> "nd.BALANCEAMT",
    "TIME" -> "nd.TIME"
  )
)
.whenNotMatched()
.insertExpr(
  Map(
    "AccountNo" -> "nd.AccountNo",
    "DATE" -> "nd.DATE",
    "TRANSACTIONDETAILS" -> "nd.TRANSACTIONDETAILS",
    "CHQNO" -> "nd.CHQNO",
    "VALUEDATE" -> "nd.VALUEDATE",
    "WITHDRAWALAMT" -> "nd.WITHDRAWALAMT",
    "DEPOSITAMT" -> "nd.DEPOSITAMT",
    "BALANCEAMT" -> "nd.BALANCEAMT",
    "TIME" -> "nd.TIME"
  )
)
.execute()
```

 # Advantage of Using Merge and UpdateExpr

Accurate Data Integration:

- By using the merge operation, you can ensure that your Delta Lake table accurately reflects the most up-to-date information from your DataFrame. This helps in maintaining a single source of truth for your entity data.

Handling Duplicates:

- The merge operation allows you to handle duplicates effectively by updating existing records (when a match is found) and inserting new records (when no match is found). This ensures that your table does not have duplicate entries for the same entity.

Conditional Logic:

- The whenMatched and whenNotMatched clauses let you define specific actions based on whether a match is found or not. This allows you to apply custom logic to update or insert records as needed.

Scalability:

- Delta Lake's merge operation is optimized for large-scale data processing, making it suitable for handling large datasets and ensuring efficient entity matching and data integration.

Schema Enforcement:

- Delta Lake provides schema enforcement, ensuring that the data being merged adheres to the defined schema of the Delta table. This prevents issues related to schema mismatches and maintains data consistency.

#Delete

```
val deltaT=DeltaTable.forPath(spark,delPath)
deltaT.delete(col("AccountNo").isin("557777","667777") && col("DATE").isNotNull)
```

Use .delete(conditions) for deleting specific records in delta.

3.5.2 Data Processing

Collecting and preparing data for Delta Lake involves several key steps and methods.

a. Data Cleaning:

- **Schema Enforcement:** Delta Lake supports schema enforcement, ensuring data quality by rejecting data that doesn't conform to the schema.
- **Data Validation:** Custom validation rules can be applied to check for null values, data types, and ranges.
- **Data Deduplication:** Delta Lake's support for ACID transactions helps in removing duplicates efficiently using the `dropDuplicates` method in Spark.

b. Data Transformation:

- **ETL Processes:** Transformation logic can be applied during the ETL process to convert raw data into a structured format.
- **Spark SQL:** Using Spark SQL, complex transformations can be applied to clean and prepare data.
- **UDFs (User-Defined Functions):** Custom UDFs can be used for specific transformations that are not possible with standard Spark functions.

c. Data Enrichment:

- **Joining Data:** Combining data from multiple sources to create enriched datasets. This can involve complex joins and aggregations using Spark.
- **Derived Columns:** Calculating additional columns based on existing data to enrich the dataset.

d. Data Partitioning And Write

Partitioning the data helps in optimizing query performance and manageability.

- **Time-based Partitioning:** Commonly, data is partitioned based on time (e.g., year, month, day) for easier time-series analysis.
- **Custom Partitioning:** Data can also be partitioned by specific columns like region, department, AccountNo or any other categorical attribute.

Write Operations:

Spark DataFrame's `write` method with `.format("delta")` is used to write data into Delta Lake.

- **Overwrite Mode:** When updating data, the overwrite mode can be used to replace existing data.
- **Append Mode:** For adding new data to an existing Delta table, append mode is used.

```
df.write
 .format("delta")
 .mode("overwrite") //append or overwrite
 .option("optimizeWrite", "true")//Optimization Technique will be discussed
 .option("autoCompact", "true") //Optimization Technique will be discussed
 .partitionBy("AccountNo")//partition by to optimize query for filtering based on
AccountNo or you can use by time, date or according to use
 .save(delPath)
```

partitionBy AccountNo makes very large number of files for each account number, so for large dataframe it might not be applicable , Instead choose your partitioning column wisely to leverage partition as it improves query performance.

While partitioning improves query performance, it introduces additional management overhead. You need to monitor and manage partitioned data, including periodic optimization (`OPTIMIZE`) to maintain optimal performance which we will discuss in Optimization in Delta Lake section below.

3.5.3 Batch Data Ingestion

ETL Processes: Extract, Transform, Load (ETL) processes are commonly used to move data from various sources (e.g., databases, APIs) into Delta Lake. Tools like Apache Airflow, NiFi, Talend, and Informatica can automate ETL pipelines.

File-based Ingestion: Data can be ingested from files (CSV, JSON, Parquet, Avro) stored in distributed storage systems like HDFS, S3, or Azure Blob Storage. Spark can read these files and write them to Delta Lake.

Database Ingestion: Using connectors (like JDBC), data can be ingested directly from relational databases into Delta Lake.

#Define Schema

```
val schema = StructType(Array(
  StructField("AccountNo", StringType, nullable = false),
  StructField("DATE", DateType, nullable = false),
  StructField("TRANSACTIONDETAILS", StringType, nullable = true),
  StructField("CHQNO", StringType, nullable = true),
  StructField("VALUEDATE", DateType, nullable = false),
  StructField("WITHDRAWALAMT", DoubleType, nullable = true),
  StructField("DEPOSITAMT", DoubleType, nullable = true),
  StructField("BALANCEAMT", DoubleType, nullable = true),
  StructField("TIME", StringType, nullable = false)
))
```

 # Manual Dataframe

```
import spark.implicits._
val df = Seq(
  ("337777", "2024-06-06", "Deposit", "Null", "2024-06-06", 0.0, 300.0, 300.0),
  ("337777", "2024-06-09", "Withdraw", "Null", "2024-06-09", 100.0, 0.0, 200.0),
  ("447777", "2024-06-09", "Deposit", "Null", "2024-06-06", 0.0, 500.0, 500.0),
  ("447777", "2024-06-02", "Withdraw", "Null", "2024-06-02", 300.0, 0.0, 100.0),
  ("447777", "2024-06-03", "Withdraw", "Null", "2024-06-03", 100.0, 0.0, 0.0)
)
  .toDF("AccountNo", "DATE", "TRANSACTIONDETAILS", "CHQNO", "VALUEDATE", "WITHDRAWALAMT",
"DEPOSITAMT", "BALANCEAMT")
  .withColumn("DATE", to_date($"DATE"))
  .withColumn("VALUEDATE", to_date($"VALUEDATE"))

val deltaT=DeltaTable.forPath(spark,delPath)

deltaT.as("dt")
  .merge(
    df.as("nd"),
    "dt.AccountNo = nd.AccountNo AND dt.VALUEDATE = nd.VALUEDATE")
  .whenMatched().updateAll() //Note: To use updateAll the schema should match for both dataframe
otherwise use updateExpr with Map() to map schema
  .whenNotMatched().insertAll()//Same condition as updateAll
  .execute()
```

 # CSV to Delta

//Reading Csv

```
val df:DataFrame=spark.read
  .option("header","true")
  .option("treatEmptyValuesAsNulls", "true")
  .option("inferSchema", "false")
  .schema(schema)
  .csv("/home/avyuthan-shah/Desktop/F1Intern/Datasets/deltaBank_transaction/bank.csv")
```

//writing to delta

```
df.write
  .mode("overwrite")
  .format("delta")
  .save("/home/avyuthan-shah/Desktop/dataFF")
```

 # Excel to Delta

Note: In built.sbt you should specify library for excel i.e

```
"com.crealytics" %% "spark-excel" % "3.4.2_0.20.3",
```

```
val df:DataFrame=spark.read
  .format("com.crealytics.spark.excel")
  .option("header","true")
  .option("treatEmptyValuesAsNulls", "true")
  .option("inferSchema", "false")
  .schema(schema)
  .load("/home/avyuthan-shah/Desktop/F1Intern/Datasets/deltaBank_transaction/
bank.xlsx")
```

```
df.write
  .mode("overwrite")
  .format("delta")
  .save("/home/avyuthan-shah/Desktop/dataF")
```

 # Parquet to Delta

```
val df:DataFrame=spark.read
  .format("parquet")
  .option("treatEmptyValuesAsNulls", "true")
  .option("inferSchema", "false")
  .schema(schema)
  .load("/home/avyuthan-shah/Desktop/F1Intern/Datasets/bank_parquet")

df.show()

df.write
  .mode("overwrite")
  .format("delta")
  .save("/home/avyuthan-shah/Desktop/dataF_parq")

or Directly convert to delta

// Convert unpartitioned Parquet table at path '<path-to-table>'
val deltaTable = DeltaTable.convertToDelta(spark, "parquet.`<path-to-table>`")
```

 # Json to Delta

```
 val df= spark.read
    .option("multiline", "true")//for multiple records
//    .option("treatEmptyValuesAsNulls", "true")
//    .option("inferSchema", "false")
    .schema(schema)//must define schema to read json file correctly
    .json("/home/avyuthan-shah/Desktop/F1Intern/Datasets/Bank_transaction/bank.json")

//Filter If necessary
val filtered_df = renamedDF
  .withColumn("DATE", to_date($"DATE", "dd-MM-yy"))
  .withColumn("VALUEDATE", to_date($"VALUEDATE", "dd-MM-yy"))
  .withColumn("CHQNO", when($"CHQNO" === "nan", null).otherwise($"CHQNO"))
  .withColumn("WITHDRAWALAMT", when($"WITHDRAWALAMT" === "nan",
null).otherwise($"WITHDRAWALAMT").cast(DoubleType))
  .withColumn("DEPOSITAMT", when($"DEPOSITAMT" === "nan",
null).otherwise($"DEPOSITAMT").cast(DoubleType))
  .withColumn("BALANCEAMT", when($"BALANCEAMT" === "nan",
null).otherwise($"BALANCEAMT").cast(DoubleType))

filtered_df.write
  .mode("overwrite")
```

```
.format("delta")
.save("/home/avyuthan-shah/Desktop/Data/dataF_json")
```

3.5.4 Streaming Data Ingestion:

- **Apache Kafka:** Kafka streams can be ingested in real-time using Spark Structured Streaming, which reads from Kafka topics and writes directly to Delta Lake.
- **Apache Flink:** Similar to Spark, Flink can also be used to stream data into Delta Lake.
- **AWS Kinesis/Azure Event Hubs:** These services can be used to stream data into Delta Lake with Spark Structured Streaming.**Kafka Stream vs Spark Streaming**

Feature / Aspect	Kafka Streams	Spark Structured Streaming
Use Case	• Stream processing within Kafka ecosystem	• Unified processing for batch and streaming
Architecture	• Lightweight, integrates directly with Kafka	• Distributed processing engine
Processing Model	• Stateful, with local state stores	• Micro-batch, with event-time processing
Fault Tolerance	• Kafka's fault-tolerance guarantees	• Spark's fault-tolerance capabilities
Scalability	• Horizontal scaling by adding instances	• Horizontal and vertical scaling capabilities
Integration	• Native integration with Kafka	• Wide range of data sources and sinks
Ease of Use	• Java-centric, but supports JVM languages	• Consistent API for batch and streaming
State Management	• Manages state with local stores or Kafka	• Handles state with checkpointing
Exactly-Once Semantics	• Built-in support	• Support through checkpointing and sinks
Language	• Primarily Java, with Scala	• Scala, Java, Python, SQL

Support	bindings	
Processing Paradigm	• Stream processing with Kafka topics	• Micro-batch processing with DataFrames
Latency	• Low latency within Kafka ecosystem	• Higher latency compared to Kafka Streams

Using Spark Streaming Only

```
val df = spark.readStream
  .schema(schema)
  .option("header", "false")
  .csv("/home/avyuthan-shah/Desktop/F1Intern/Datasets/Bank_transaction/
StreamingBank")
//readStream reads

import spark.implicits._
val filtered_df=df
  .withColumn("DATE",to_date($"DATE", "dd-MM-yy"))
  .withColumn("VALUEDATE", to_date($"VALUEDATE", "dd-MM-yy"))

val query=filtered_df
  .writeStream
  .format("delta")
  .outputMode("append")
  .option("checkPointLocation","/home/avyuthan-shah/Desktop/dataFcheckpoints")
  //Checkpoints are essential in structured streaming with Spark because they store the
state of the streaming query.
  // This allows Spark to recover from failures and continue processing from where it left off
rather than starting from scratch.
  // The checkpoint directory is used to persist metadata about the streaming job, such as
offsets and progress.

  .start("/home/avyuthan-shah/Desktop/dataF")

query.awaitTermination()
```

ReadStream from Delta-Spark

```
val stream = spark.readStream
  .format("delta")
  .option("startingVersion", "0")
  //.option("skipChangeCommits", "true") // Ignore updates and deletes. Completely ignores
updates and deletes, treating the data as static.
  .option("ignoreChanges","true") //ignoreChanges in Delta Lake allows structured
streaming to handle updates and deletes in a way that fits real-time data processing.
  // It ensures that changes to data are properly reflected downstream, though it may
introduce duplicates for unchanged data.
  // This option is useful when you need to maintain data integrity and continuity in your
streaming pipelines despite ongoing updates and occasional deletions.
```

```
  .load("/home/avyuthan-shah/Desktop/dataF")

  .writeStream
  .format("console")
  //.outputMode("update")//Only the rows that were updated in the result table since the last trigger are output.
  //.outputMode("complete")//the entire result table for each trigger interval and writes the complete set of results to the sink when there are streaming aggregations on streaming DataFrames/Datasets
  .outputMode("append")// New rows added since the last trigger.
  .start()

stream.awaitTermination()
```

Following code reads data from delta as stream and displays it to console.

Errors In Streams

#1

```
val stream = spark.readStream
 .format("delta")
 .option("startingVersion", "1")
 .load("/home/avyuthan-shah/Desktop/dataF")
 .writeStream.format("console").start()
```

#

org.apache.spark.sql.streaming.StreamingQueryException: [DELTA_SOURCE_TABLE_IGNORE_CHANGES] Detected a data update (for example part-00007-1d6e0471-585e-4db3-ad2c-5e03ac7553aa-c000.snappy.parquet) in the source table at version 5. This is currently not supported. If you'd like to ignore updates, set the option 'skipChangeCommits' to 'true'. If you would like the data update to be reflected, please restart this query with a fresh checkpoint directory.

- This error occurs when a data update is detected in the Delta table. By default, Spark Structured Streaming expects data in the source Delta table to be append-only. When updates or deletes are detected, the streaming query fails with this exception.

Solution:

Ignore Changes: Use the skipChangeCommits option to ignore updates and deletes in the source table.

```
val stream = spark.readStream
 .format("delta")
 .option("startingVersion", "1")//if not specified latest version will be read
 .option("skipChangeCommits", "true") // Ignore updates and deletes completely
```

```
    Or
.option("ignoreChanges", "true") // Downstreams both old and new record, chance of duplication
.load("/home/avyuthan-shah/Desktop/dataF")
.writeStream.format("console").start()
```

#2

[error] org.apache.spark.sql.AnalysisException: Complete output mode not supported when there are no streaming aggregations on streaming DataFrames/Datasets;

[error] delta

The error message org.apache.spark.sql.AnalysisException: Complete output mode not supported when there are no streaming aggregations on streaming DataFrames/Datasets typically occurs in Spark Structured Streaming when you attempt to use the complete output mode without performing any streaming aggregations (such as aggregations or grouping) on the streaming DataFrame or Dataset.

Solution

Use the complete output mode with applying any aggregations or groupings (groupBy, agg, etc.) on the streaming DataFrame or Dataset

Limitations

- **Data Consistency**: Ignoring updates and deletes means the stream might miss some changes to the data, which can affect data consistency. Use this option only if your application can tolerate such inconsistencies.
- **Not Suitable for All Use Cases**: For applications requiring up-to-date views of the data including updates and deletes, this option is not appropriate. You would need to handle these changes properly, potentially by restarting the stream with a fresh checkpoint directory.

By using .option("skipChangeCommits", "true"), you trade off strict data consistency for simpler stream processing and potentially better performance, which might be a suitable compromise for many real-time data processing applications

Kafka Stream

Include following lines in built.sbt libraryDependencies

"org.apache.kafka" %% "kafka" % "3.7.0",

"org.apache.kafka" % "kafka-clients" % "3.7.0",

"org.apache.kafka" % "kafka-streams" % "3.7.0",

"org.apache.spark" %% "spark-sql-kafka-0-10" % "3.5.1",

```
val schema = StructType(Array(
  StructField("Account No", StringType, nullable = false),
  StructField("DATE", StringType, nullable = false),
  StructField("TRANSACTION DETAILS", StringType, nullable = false),
  StructField("CHQ NO", StringType, nullable = true),
  StructField("VALUE DATE", StringType, nullable = false),
  StructField("WITHDRAWAL AMT", StringType, nullable = true),
  StructField("DEPOSIT AMT", StringType, nullable = true),
  StructField("BALANCE AMT", StringType, nullable = true)
))

val kafkaDF = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "localhost:9092")
  .option("subscribe", "bank_transactions")
  .load()

// Parse the JSON data from Kafka
val transactionsDF = kafkaDF.selectExpr("CAST(value AS STRING) AS json")
  .select(from_json(col("json"), schema).as("data"))
  .select("data.*")

// Write the parsed data to Delta Lake
val query = transactionsDF.writeStream
  .format("delta")
  .outputMode("append")
  .option("checkpointLocation", "/path/to/checkpoint/dir")
  .start("/path/to/delta/table")
query.awaitTermination()
```

3.6 Delta Features

3.6.1 ACID

```
import spark.implicits._
  val     newdf=newData.toDF("AccountNo",     "DATE",     "TRANSACTIONDETAILS",     "CHQNO",
"VALUEDATE", "WITHDRAWALAMT", "DEPOSITAMT", "BALANCEAMT")
   try {
    val delTable=DeltaTable.forPath(spark,delPath)
   //Checking Rollback For Atomicity
   if(true){
    throw new RuntimeException("Simulated failure occurred before the merge operation")
```

```
  }
   delTable.as("t").merge(newdf.as("s"),"t.AccountNo = s.AccountNo AND t.DATE=s.DATE")
     .whenMatched.updateAll()
     .whenNotMatched.insertAll()
     .execute()
   println("Transaction Succeeded")
  }catch{
  case e:Exception=>println("transaction failed: "+e.getMessage)
  }
  spark.stop()
 }
}
```

Error was simulated before execution of merge or updation of table, Thus the data in delta table was not updated

Another way it to terminate program in the middle while running, and on testing database was not updated with new data.

```
try {
   val delTable=DeltaTable.forPath(spark,delPath)

  //Checking Rollback For Atomicity
   delTable.as("t").merge(newdf.as("s"),"t.AccountNo = s.AccountNo AND
t.DATE=s.DATE")
     .whenMatched.updateAll()
     .whenNotMatched.insertAll()
     .execute()
   println("Transaction Succeeded First")

  if(true){
   throw new RuntimeException("Simulated failure occurred before the merge operation")
  }

  }catch{
  case e:Exception=>println("transaction failed: "+e.getMessage)
  }

  spark.stop()
 }
}
```

Now, the error was simulated after execution. However, the version table was updated with new data. And new data was seen, thus verifying atomicity.

```
24/06/03 15:16:09 INFO BlockManager: Removing RDD 6
Transaction Succeeded First
transaction failed: Simulated failure occurred before the merge operation
24/06/03 15:16:09 INFO SparkContext: SparkContext is stopping with exitCode 0.
24/06/03 15:16:09 INFO SparkUI: Stopped Spark web UI at http://10.13.164.84:4040
24/06/03 15:16:09 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
24/06/03 15:16:09 INFO MemoryStore: MemoryStore cleared
24/06/03 15:16:09 INFO BlockManager: BlockManager stopped
24/06/03 15:16:09 INFO BlockManagerMaster: BlockManagerMaster stopped
24/06/03 15:16:09 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
24/06/03 15:16:09 INFO SparkContext: Successfully stopped SparkContext
[success] Total time: 24 s, completed Jun 3, 2024, 3:16:09 PM
```

3.6.2 Isolation And Concurrency

For supported storage systems, multiple writers across multiple clusters can simultaneously modify a table partition and see a consistent snapshot view of the table and there will be a serial order for these writes. Readers continue to see a consistent snapshot view of the table that the Apache Spark job started with, even when a table is modified during a job.

By utilizing its transaction log, Delta Lake employs optimistic concurrency control and conflict detection. Following steps shows how delta really handles conflicts.

Transaction Record

All table modifications are documented in a transaction log that Delta Lake keeps up to date. Every transaction in the log has a timestamp, version number, and information about the files that were added, removed, or changed.

Process for Detecting Conflicts

- **Read Isolation:** A transaction reads the most recent copy of the table when it begins. This separates reads from concurrent writes and guarantees a consistent snapshot view throughout the transaction.
- **Write Intent:** A transaction that writes data does not replace files that already contain data right away. Rather, it identifies older files for deletion and prepares new data files for addition.
- **Commit Phase:** During the commit phase, Delta Lake checks the transaction's read version against the table's current version:
    - I**f the current version matches the read version,** the transaction is successfully committed, with new files added to the transaction log and old files removed as needed.
    - **If the existing version has changed:** This indicates that another transaction made changes since the current transaction began. This is where conflict detection occurs. The Delta Lake will then:
        - **Check for Overlapping Modifications.** Delta Lake determines whether the files read or modified by the current transaction overlap with those modified by previous transactions.
        - **Detect conflicts:** If there are overlaps, a conflict is discovered and the transaction is terminated.

Advantage of OCC

Delta Lake finds conflicts in the transaction log by comparing the read and write sets of concurrent transactions. This ensures that if two transactions edit the same data, the second transaction detects the conflict and can retry. This strategy allows Delta Lake to effectively manage high-concurrency scenarios without the use of standard file locking methods. This prevents data corruption during multiple writes thus making suitable for ACID transactional processing.

Write Conflicts

Files were added to the root of the table by a concurrent update. Please try the operation again.
Conflicting commit: {"timestamp":1720351382482,"operation":"MERGE","operationParameters":{"predicate":["(((((AccountNo#776 = AccountNo#897) AND (DATE#777 = DATE#950)) AND (TRANSACTIONDETAILS#778 = TRANSACTIONDETAILS#900)) AND ((CHQNO#779 = CHQNO#902) AND (VALUEDATE#780 = VALUEDATE#980))) AND (((WITHDRAWALAMT#781 = WITHDRAWALAMT#910) AND (DEPOSITAMT#782 = DEPOSITAMT#911)) AND ((BALANCEAMT#783 = BALANCEAMT#912) AND (TIME#784 = TIME#913))))"],"matchedPredicates":[{"actionType":"update"}],"notMatchedPredicates":[{"actionType":"insert"}],"notMatchedBySourcePredicates":[]},"readVersion":12,"isolationLevel":"Serializable","isBlindAppend":false,"operationMetrics":{"numTargetRowsCopied":"0","numTargetRowsDeleted":"0","numTargetBytesRemoved":"0","numTargetDeletionVectorsAdded":"0","numTargetRowsMatchedUpdated":"0","numTargetRowsMatchedDeleted":"0","numTargetRowsUpdated":"0","numTargetChangeFilesAdded":"0","numTargetRowsNotMatchedBySourceDeleted":"0","rewriteTimeMs":"632","numTargetFilesAdded":"1","numTargetBytesAdded":"2569","executionTimeMs":"4650","numTargetRowsInserted":"1","numTargetDeletionVectorsUpdated":"0","scanTimeMs":"3840","numOutputRows":"1","numTargetDeletionVectorsRemoved":"0","numTargetRowsNotMatchedBySourceUpdated":"0","numSourceRows":"1","numTargetFilesRemoved":"0"},"engineInfo":"Apache-Spark/3.5.1 Delta-Lake/3.1.0","txnId":"83b4cd94-019c-4104-ae34-a86c3e993790"}
Refer to https://docs.delta.io/latest/concurrency-control.html for more details.

Delta Lake finds conflicts in the transaction log by comparing the read and write sets of concurrent transactions. This ensures that if two transactions edit the same data, the second transaction detects the conflict and can retry. This strategy allows Delta Lake to effectively manage high-concurrency scenarios without the use of standard file locking methods.

ConcurrentAppendException

Cause:

- A concurrent operation adds files to the same partition (or any unpartitioned table) that your operation reads.
- Common scenarios include concurrent INSERT, DELETE, UPDATE, and MERGE operations.
- Concurrent operations may physically update different partitions, but one may read a partition while another updates it.

Solution:

- Ensure that operations have non-overlapping conditions.
- Use partition-specific conditions to isolate operations.
- Implement retry logic for operations encountering this conflict.

ConcurrentDeleteReadException

- **Cause**: A concurrent operation deleted a file that your operation read.
- **Scenarios**:
  - A `DELETE`, `UPDATE`, or `MERGE` operation reading files that another concurrent operation deletes.
- **Solution**:
  - Make read operations more resilient to concurrent deletes by handling retries.
  - Implement retry logic for operations encountering this conflict.
  -

ProtocolChangedException

This exception can occur in the following cases:

- When your Delta table is upgraded to a new version. For future operations to succeed you may need to upgrade your Delta Lake version.
- When multiple writers are creating or replacing a table at the same time.
- When multiple writers are writing to an empty path at the same time.

ConcurrentTransactionException

If a streaming query using the same checkpoint location is started multiple times concurrently and tries to write to the Delta table at the same time. You should never have two streaming queries use the same checkpoint location and run at the same time [5].

See Appendix 1 for retry logic implementation.

Given code in appendix 1 implements retry logic, however the no of retries is kept same as the no of concurrent tasks. This might not be suitable in highly concurrent scenario.

Solution

- Partitioning the delta table according to required need, for example, partition by date, name, etc.

  Note: Partition creates separate file for each specified partition column. So in terms of big data make sure to only partition which has less number of variety in data. Example: Gender, State, Flags, Date range.
- For small scale scenario, you can partition by AccountNo ,or any . Thus, for small scale using delta in small scale makes olap system much faster for data processing.
- Another way is to be more specific during merge operation

// Original condition

deltaTable.as("t").merge(

```
    source.as("s"),
    "s.user_id = t.user_id AND s.date = t.date AND s.country = t.country"
)
  .whenMatched().updateAll()
  .whenNotMatched().insertAll()
  .execute()
```

// Revised condition

```
deltaTable.as("t").merge(
    source.as("s"),
     "s.user_id = t.user_id AND s.date = t.date AND s.country = t.country AND t.date = ''' + <date> + ''' AND t.country = ''' + <country> + """
)
  .whenMatched().updateAll()
  .whenNotMatched().insertAll()
  .execute()
```

- **Condition:** Adds explicit criteria that match specific dates and countries in addition to user_id.
- **Improvement:**
  - **Specificity:** Limits the scope of the merge to a specific date and country, ensuring that the operation only affects the appropriate division.
  - **Reduced Conflicts:** By being more particular, it reduces the possibility of conflict with other concurrent processes working on other partitions or data regions.

```
deltaTable.as("dt")
  .merge(newDf.as("df"), s"$mergeCondition AND dt.AccountNo= $accountNo AND dt.TIME=${time.getTime()}")
  .whenMatched().updateAll()
  .whenNotMatched().insertAll()
  .execute()
```

While Delta Lake excels at handling concurrent read operations, allowing numerous users to execute queries without conflict, problems might develop with write operations, particularly with merge operations. This is primarily due to how Delta Lake handles its transaction log and concurrent updates. Given the possibility of merge conflicts, Delta Lake may not be appropriate for traditional OLTP systems that require frequent, low-latency updates with high concurrency. OLTP systems often require immediate consistency and minimal contention for the same data.

3.6.3 Time Travel

```
spark.read.format("delta").option("versionAsOf",0).load(delPath).createOrReplaceTempVie
w("delta_table")

val result = spark.sql(
  s"""
     |SELECT * FROM delta_table
     |WHERE AccountNo IN ("409000438620","1196428","409000438611")     |ORDER BY
AccountNo,VALUEDATE DESC;
   """.stripMargin)

result.show(truncate=false)
```

By specifying the desired version in option section you can access historical data as shown above. You can specify version or timestamp using

.option(“timestampAsOf”,”2024-07-01”)

Performance of Time Travel

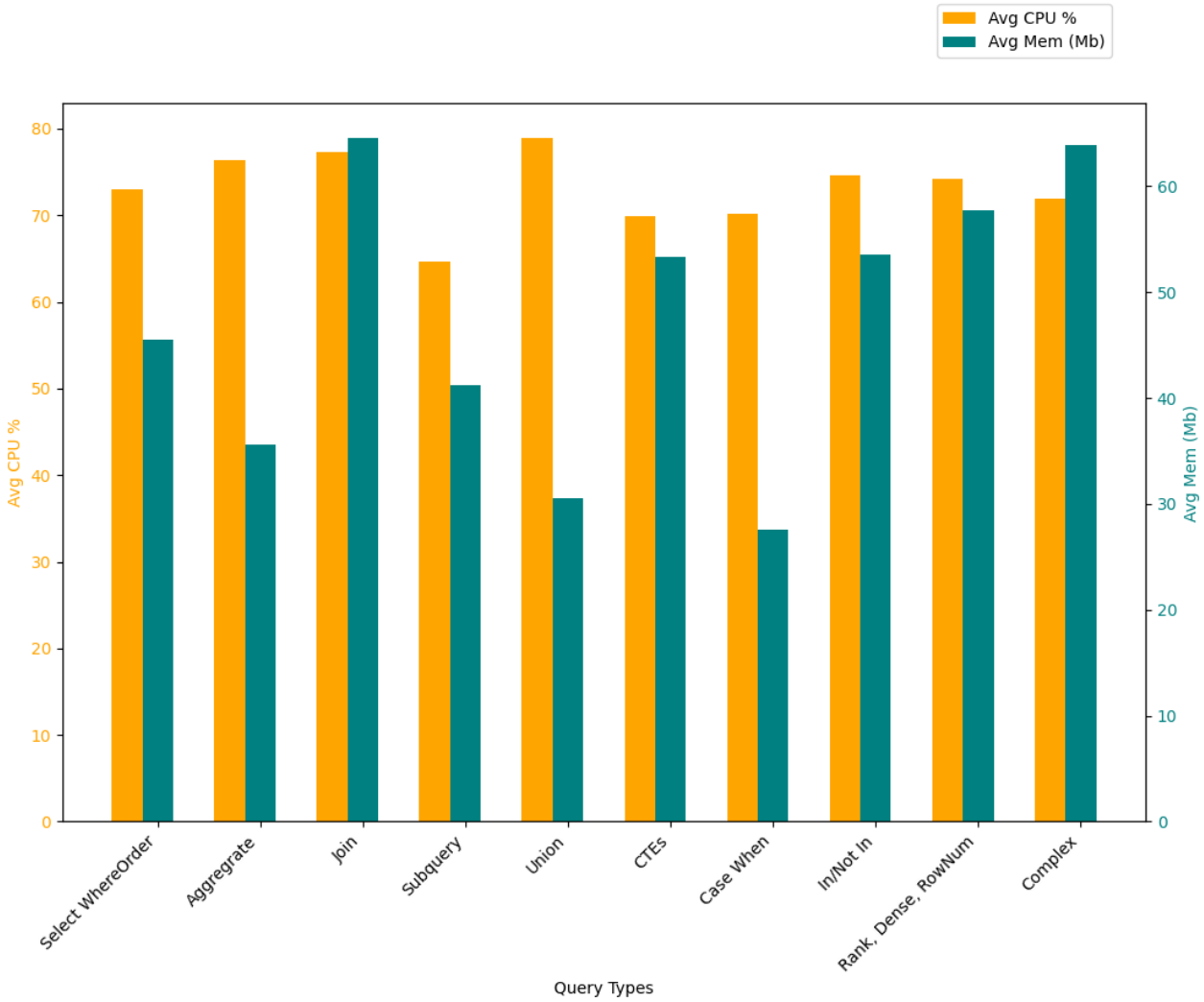


Figure: CPU and Memory usage during time travel

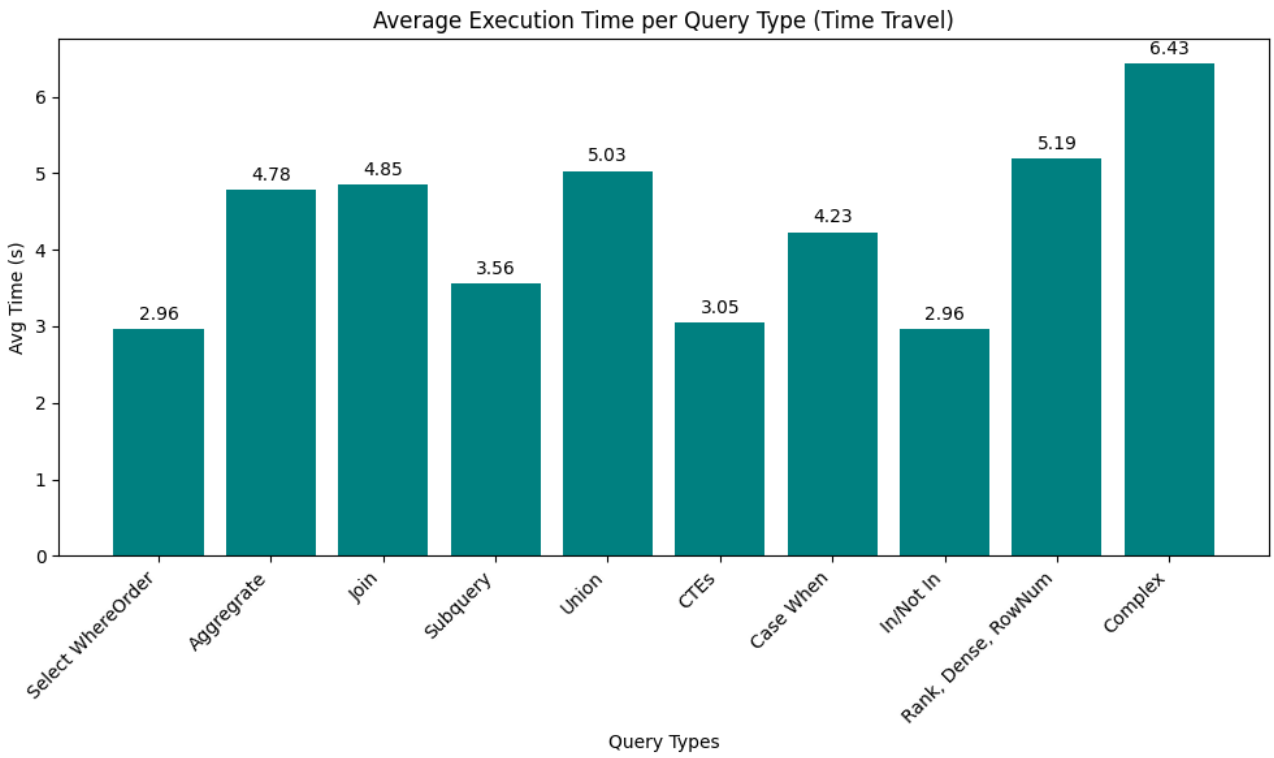


Figure: Query success time during time travel and fetching old data containing 116k lines

3.6.4 Schema Enforcement and Evolution

```
val spark = SparkSession.builder()
  .appName("Schema Enforcement")
  .master("local[*]")
  .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
  .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
  .config("spark.databricks.delta.schema.autoMerge.enabled", "true")//to automerge
schema
  .getOrCreate()
```

//Write the new dataframe to delta

```
val newData = Seq(
  ("557777",time.getDate(), "Withdraw", "Null", time.getDate(), 0.0, 300.0,
3600.0,time.getTime()),
  ("667777", time.getDate(), "Deposit", "Null", time.getDate(), 200.0, 0.0,
5000.0,time.getTime())
)
  .toDF("AccountNo","DATE","TRANSACTIONDETAILS", "CHQNO", "VALUEDATE",
"WITHDRAWALAMT", "DEPOSITAMT", "BALANCEAMT","TIME")//New Column Time
added
  .withColumn("DATE", to_date($"DATE","yy-MM-dd"))
  .withColumn("VALUEDATE", to_date($"VALUEDATE","yy-MM-dd"))
```

```
newData.write
  .format("delta")
  .mode("append")
  .save(delPath)
```

```
Error Occured: A schema mismatch detected when writing to the Delta table (Table ID: a864fc88-6df2-4711-a505-81c829cff3e2).
To enable schema migration using DataFrameWriter or DataStreamWriter, please set:
'.option("mergeSchema", "true")'.
For other operations, set the session configuration
spark.databricks.delta.schema.autoMerge.enabled to "true". See the documentation
specific to the operation for details.

Table schema:
root
-- AccountNo: string (nullable = true)
-- DATE: date (nullable = true)
-- TRANSACTIONDETAILS: string (nullable = true)
-- CHQNO: string (nullable = true)
-- VALUEDATE: date (nullable = true)
-- WITHDRAWALAMT: double (nullable = true)
-- DEPOSITAMT: double (nullable = true)
-- BALANCEAMT: double (nullable = true)


Data schema:
root
-- AccountNo: string (nullable = true)
-- DATE: date (nullable = true)
-- TRANSACTIONDETAILS: string (nullable = true)
-- CHQNO: string (nullable = true)
-- VALUEDATE: date (nullable = true)
-- WITHDRAWALAMT: double (nullable = true)
-- DEPOSITAMT: double (nullable = true)
-- BALANCEAMT: double (nullable = true)
-- TIME: string (nullable = true)
```

Figure: Schema Enforcement Using Spark Write

From above figure we can see that using spark write method sends and error. Instead use delta merge for schema enforcement.

```
|AccountNo   |DATE      |TRANSACTIONDETAILS              |CHQNO|VALUEDATE |WITHDRAWALAMT|DEPOSITAMT|BALANCEAMT|
+------------+----------+--------------------------------+-----+----------+-------------+----------+----------+
|557777      |2024-07-14|Withdraw                        |Null |2024-07-14|0.0          |300.0     |3600.0    |
|409000611074|2017-06-29|TRF FROM  Indiaforensic SERVICES|NULL |2017-06-29|NULL         |1000000.0 |1000000.0 |
|409000611074|2017-07-05|TRF FROM  Indiaforensic SERVICES|NULL |2017-07-05|NULL         |1000000.0 |2000000.0 |
|409000611074|2017-07-18|FDRL/INTERNAL FUND TRANSFE      |NULL |2017-07-18|NULL         |500000.0  |2500000.0 |
|409000611074|2017-08-01|TRF FRM  Indiaforensic SERVICES |NULL |2017-08-01|NULL         |3000000.0 |5500000.0 |
|409000611074|2017-08-16|FDRL/INTERNAL FUND TRANSFE      |NULL |2017-08-16|NULL         |500000.0  |6000000.0 |
|409000611074|2017-08-16|FDRL/INTERNAL FUND TRANSFE      |NULL |2017-08-16|NULL         |500000.0  |6500000.0 |
|409000611074|2017-08-16|FDRL/INTERNAL FUND TRANSFE      |NULL |2017-08-16|NULL         |500000.0  |7000000.0 |
```

Figure Using Delta Merge

The merge operation in Delta Lake does not automatically update the schema of the Delta table. The merge operation is primarily used for upserts, allowing to conditionally insert, update, or delete records based on specified conditions.
When using the merge operation to merge data with a Delta table, it assumes that the schema of the incoming data (additionalData) matches the schema of the Delta table (oldData). If there are new columns in the incoming data that are not present in the Delta table, they will be ignored during the merge operation.
To handle schema evolution, one needs to explicitly specify to merge the schema of the

incoming data with the schema of the Delta table. It can be achieved by setting the mergeSchema option to true when writing the new data to the Delta table.

Schema Evolvement

```
val spark = SparkSession.builder()
  .appName("Schema Enforcement")
  .master("local[*]")
  .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
  .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
  .config("spark.databricks.delta.schema.autoMerge.enabled", "true")//to automerge
schema
  .getOrCreate()
```

And then use delta merge.

Or if you don't want to specify auto merge in sparksession, set mergeSchema option to true in spark.write method. However delta.merge is preferred as it is more relaible and secure.

```
newData.write
  .format("delta")
  .mode("append")
   .option("mergeSchema","true")
  .save(delPath)
```

Output

```
+-----------+----------+-------------------------------+-----+----------+-------------+----------+----------+--------+
|AccountNo  |DATE      |TRANSACTIONDETAILS             |CHQNO|VALUEDATE |WITHDRAWALAMT|DEPOSITAMT|BALANCEAMT|TIME    |
+-----------+----------+-------------------------------+-----+----------+-------------+----------+----------+--------+
|887777     |2024-07-14|Withdraw                       |Null |2024-07-14|0.0          |300.0     |3600.0    |17:03:21|
|409000611074|2017-06-29|TRF FROM  Indiaforensic SERVICES|NULL |2017-06-29|NULL         |1000000.0 |1000000.0 |NULL    |
|409000611074|2017-07-05|TRF FROM  Indiaforensic SERVICES|NULL |2017-07-05|NULL         |1000000.0 |2000000.0 |NULL    |
|409000611074|2017-07-18|FDRL/INTERNAL FUND TRANSFE     |NULL |2017-07-18|NULL         |500000.0  |2500000.0 |NULL    |
|409000611074|2017-08-01|TRF FRM  Indiaforensic SERVICES|NULL |2017-08-01|NULL         |3000000.0 |5500000.0 |NULL    |
```

Figure Schema Enforced with new column with default value Null for old data

Delta seamlessly updates schema thus preventing insertions of bad records. And on requirement it can be evolved. In case of mismatched data type or unmatched columns, delta will raise an error.

Error: Unmatched No of Column

Error Occured: [DELTA_MERGE_UNRESOLVED_EXPRESSION] Cannot resolve WITHDRAWALAMT in UPDATE clause given columns nd.AccountNo, nd.DATE, nd.TRANSACTIONDETAILS, nd.CHQNO, nd.VALUEDATE, nd.DEPOSITAMT, nd.BALANCEAMT, nd.TIME

Solution: Set autoMerge config to true in spark session as shown above in schema evolvement.

Error: Unmatched DataType

Error Occured: Job aborted due to stage failure: Task 1 in stage 14.0 failed 1 times, most recent failure: Lost task 1.0 in stage 14.0 (TID 121) (10.13.164.84 executor driver): org.apache.spark.SparkDateTimeException: [CAST_INVALID_INPUT] The value '24-07-15' of the type "STRING" cannot be cast to "DATE" because it is malformed. Correct the value as per the syntax, or change its target type. Use `try_cast` to tolerate malformed input and return NULL instead. If necessary set "spark.sql.ansi.enabled" to "false" to bypass this error.

Solution1: Use ansi.enabled

Add following line to spark session builder before getOrCreate

```
.config("spark.sql.ansi.enabled", "false")
```

However this is not recommended and it doesn't solves issue properly.

Solution 2: Clean dataframe before insertion to match given data type as it is the most safest way. Schema Enforcement solves column mismatch but mismatch in datatype is not solved.

For above error, date was passed as string so to solve this.

```
val newData = Seq(
   ("107777",time.getDate(), "Withdraw", 3, time.getDate(), "0.0", 300.0, 3600.0,time.getTime()),
   ("117777", time.getDate(), "Deposit", 3, time.getDate(),"200.0", 0.0, 5000.0,time.getTime())
 )
   .toDF("AccountNo","DATE","TRANSACTIONDETAILS", "CHQNO", "VALUEDATE", "WITHDRAWALAMT",
"DEPOSITAMT", "BALANCEAMT","TIME")
   .withColumn("VALUEDATE", to_date($"VALUEDATE","yy-MM-dd"))
   .withColumn("TIME", to_timestamp($"TIME", "HH:mm:ss"))
```

Use to_date to convert string into date format.

Note: string should also match date format if random value is passed then it throws an error.

Conclusion

Data Lake helps to avoid bad data getting your data lakes by providing the ability to specify the schema and help enforce it. It prevents data corruption by preventing the bad data get into the system even before the data is ingested into the data lake by giving sensible error messages.

3.6.5 Unified Batch And Stream Data Ingestion

Delta Lake delivers a unified platform for batch and stream data processing, providing consistency, stability, and scalability. Delta Lake's storage format, which is based on Apache Parquet and improved with transactional support, is at the heart of its capabilities. This enables Delta Lake to handle ACID transactions, schema enforcement, and data versioning, making it ideal for both real-time and batch processing.

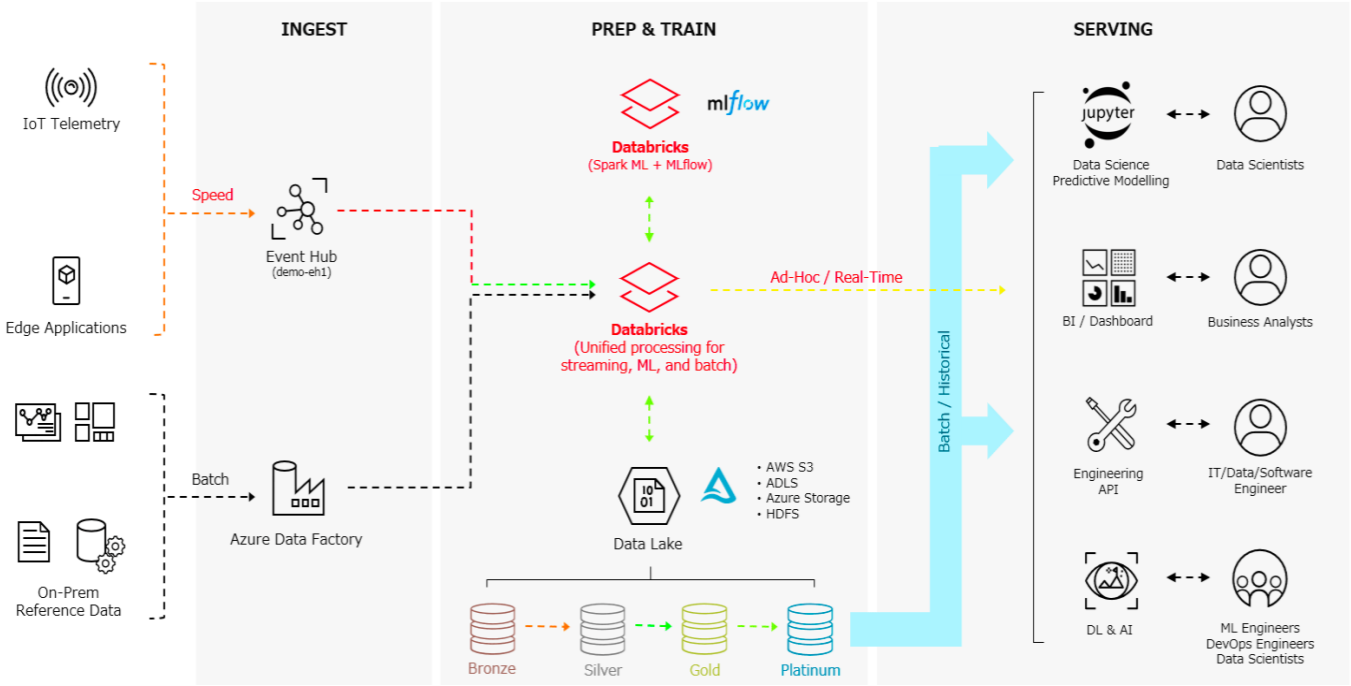


Figure: Unified batch and streaming pipeline in delta using databricks [7]

Data can be imported from a variety of sources for batch processing, including on-premises databases, cloud storage, and data warehouses, utilizing tools such as Azure Data Factory. This data is frequently fed into Delta Lake in a systematic manner, providing a solid foundation for historical analysis and machine learning model training.

Delta Lake uses systems such as Azure Event Hubs and Apache Kafka to capture real-time data from IoT devices, edge apps, and other streaming data sources. This data is incrementally fed into Delta Lake, ensuring that the data lake contains the most recent information.

Stream Ingestion in Delta Lake: Microbatch vs. Continuous Processing

Delta Lake supports both microbatch and continuous streaming modes to ingest data. Here's an explanation of each mode and how Delta Lake handles them:

Microbatch Processing

In microbatch processing, the stream is divided into small, time-bounded batches. Each microbatch captures a set of records arriving within a specified interval (e.g., every second, minute, etc.). This method has several advantages:

- **Consistency:** Each microbatch is processed as a discrete unit, ensuring data consistency and enabling fault tolerance.
- **Efficiency:** Microbatch processing can be optimized to handle a large volume of data more efficiently than processing records one by one.
- **Simplicity:** Implementation is straightforward as it aligns well with batch processing logic.

However, this approach might introduce a slight delay in data availability, as records are only processed once the microbatch interval elapses.

Continuous Processing

Continuous processing, also known as true stream processing, processes records as they arrive. This approach offers:

- **Low Latency:** Near real-time data processing with minimal delay between data arrival and processing.
- **Fine-Grained Control:** Immediate response to incoming data, beneficial for scenarios requiring prompt action.

Continuous processing may be more complex to implement and manage, especially concerning fault tolerance and ensuring exactly-once processing semantics.

Delta Lake Versioning and Stream Ingestion

Delta Lake maintains versions of data to support ACID transactions, schema enforcement, and time travel. Each commit to the Delta table (whether from batch or stream processing) generates a new version. While this ensures strong consistency and the ability to roll back or query historical data, it can also lead to a large number of versions, especially with frequent microbatches or continuous streams.

Managing Large Numbers of Versions

To manage the overhead of many versions, Delta Lake provides several mechanisms:

- **Compaction:** Periodically compact small files and microbatches into larger files to reduce the number of versions and optimize query performance.
- **Optimize Operations:** Use the OPTIMIZE command to compact files and the VACUUM command to remove old, unused versions, freeing up storage.

Implementation for stream data from kafka is shown in chapter 3.5.4

3.6.5 Optimizations In Delta

1. Vacuum

Vacuuming in Delta Lake is a crucial maintenance operation that helps manage storage and optimize performance by cleaning up old data files. Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark and big data workloads, making it essential for maintaining data consistency and reliability in data lakes.

Why Use Vacuum?

1. **Space Management**: Over time, Delta tables accumulate old versions of data due to updates, deletes, or merges. Vacuuming helps reclaim storage space by removing obsolete data files.
2. **Performance Optimization**: Regular vacuuming can improve query performance by reducing the number of files that Spark needs to scan during operations.
3. **Compliance and Retention**: It allows organizations to enforce data retention policies by removing data older than a specified retention period.

How Does Vacuum Work?

When you perform a vacuum operation, Delta Lake removes files that are no longer needed based on the specified retention period. By default, Delta Lake retains data for 7 days to ensure that users can access previous versions of data for time travel or rollback scenarios. You can customize this retention period according to your requirements.

Implementation:

```
val deltaTable = DeltaTable.forPath(spark, delPath)
deltaTable.vacuum()
```

Generally default retention period is 7 days or 168 hours. To specifically define desired retention period you can use

```
deltaTable.vacuum(retentionHours=48)
```

Error: Delta doesn't accept retentionHours less than 168 hours

```
[error] java.lang.IllegalArgumentException: requirement failed: Are you sure you would like to vacuum files with such a low retention period? If you have
[error] writers that are currently writing to this table, there is a risk that you may corrupt the
[error] state of your Delta table.
[error]
[error] If you are certain that there are no operations being performed on this table, such as
[error] insert/upsert/delete/optimize, then you may turn off this check by setting:
[error] spark.databricks.delta.retentionDurationCheck.enabled = false
[error]
[error] If you are not sure, please use a value not less than "168 hours".
```

Solution: Set config for SparkSession

```
.config("spark.databricks.delta.retentionDurationCheck.enabled", "false")
```

Only add this at your own risk to remove all the old versions older than 48 hours.If you use vacuum(0), it will deleta every old file and only keep latest one. Overall file size will also be reduced.

Disadvantage of Vacuum:

If you run `VACUUM` on a Delta table, you lose the ability to time travel back to a version older than the specified data retention period.

2. Compaction (Bin Packaging)

Delta Lake's compaction (or bin-packing) procedure involves merging smaller files into bigger ones. This approach improves the performance of read queries by minimizing the number of files that must be opened and processed. Here is a brief overview:

Purpose: To enhance read efficiency by reducing the overhead associated with managing a large number of tiny files.

When to Use: For situations where frequent minor updates or inserts result in a large number of little files.

How to perform:

optimize().executeCompaction()

Partitioning is optional; you can define a subset of data to condense using a partition predicate such as

optimize().where("date='2021-11-18'").executeCompaction().

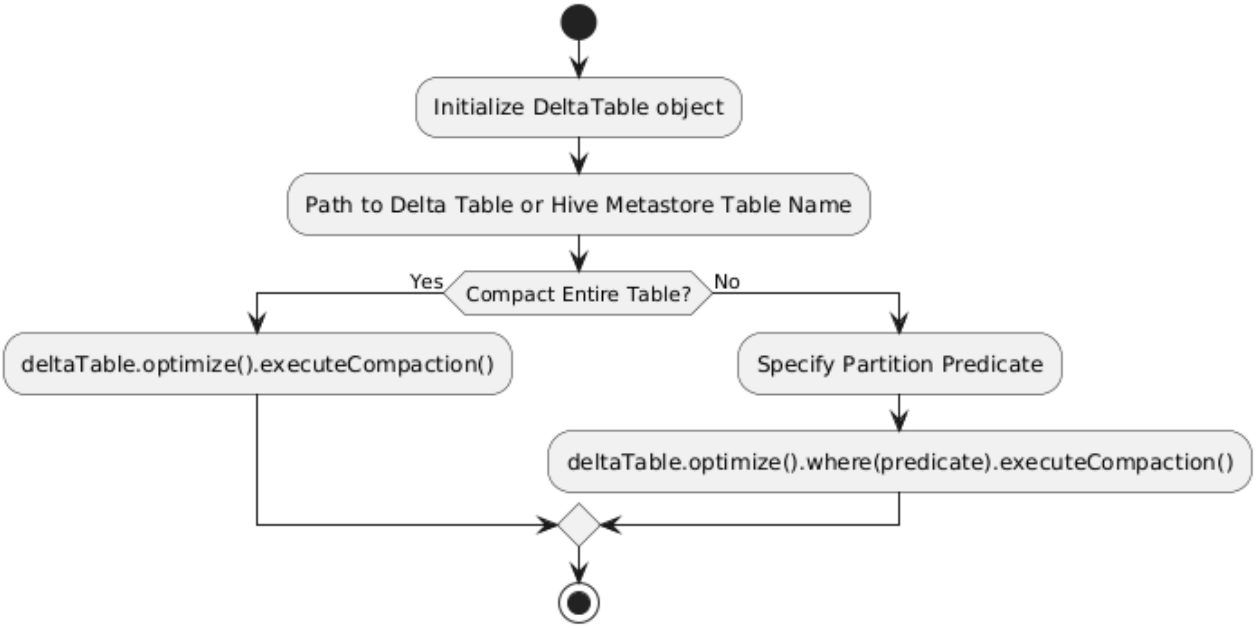


Figure Compaction Flowchart

Delta Lake uses an internal heuristic to decide how many output files to produce during compaction. It tries to coalesce small files into larger ones to reduce the total number of files while balancing data distribution and parallelism.

The default behavior aims to reduce the number of files and optimize file size distribution for improved read performance. However, the exact number of output files can vary depending on factors like the original file sizes, partitioning, and the `autoOptimize` configuration.

```
filtered_df.write
  .format("delta")
  .mode("overwrite")
  .option("autoCompact", "true")
  .save(delPath)
//or you can specify in spark session
```

Overhead Issues with Small Files

i. **Increased Metadata Overhead:**

Each file comes with its own metadata overhead. This includes information such as file location, creation time, permissions, etc. Having many small files increases the metadata overhead, which can impact both storage and performance.

ii. **Performance Degradation:**

Querying data spread across numerous small files can be slower compared to querying data stored in larger, more consolidated files. This is due to the overhead of opening and managing a large number of files during query execution.

How Compaction Solves These Issues

i. **File Consolidation:**

Merging Small Files: Compaction consolidates multiple small files into fewer, larger files. This reduces the overall number of files that need to be managed and accessed, thereby reducing metadata overhead and improving file system performance.

ii. **Improved Query Performance:**

Reduced File Opening Overhead: By reducing the number of files, compaction improves query performance. Queries can read data more efficiently from larger files, resulting in faster query execution times.

iii. **Resource Optimization:**

Efficient Resource Utilization: File systems and storage systems handle larger files more efficiently. Compacting small files into larger ones optimizes resource utilization, leading to better overall system performance and cost efficiency.

Note:

- Compaction only updates latest table whereas old table files are still there as delta maintains time travel feature, so using vacuum is a good practice by trading off time travel feature and querying very old data keeping only necessary one.
- Bin-packing optimization is idempotent, meaning that if it is run twice on the same dataset, the second run has no effect.

- Bin-packing aims to produce evenly-balanced data files with respect to their size on disk, but not necessarily number of tuples per file. However, the two measures are most often correlated.
- Python and Scala APIs for executing OPTIMIZE operation are available from Delta Lake 2.0 and above.
- Set Spark session configuration spark.databricks.delta.optimize.repartition.enabled=true to use repartition(1) instead of coalesce(1) for better performance when compacting many small files.

```
.config("spark.databricks.delta.optimize.repartition.enabled","false") //Add this to your sparksession
```

When you set `spark.databricks.delta.optimize.repartition.enabled=true` in your Spark session configuration and perform compaction in Delta Lake using `deltaTable.optimize().executeCompaction()`, it aims to reduce the number of files to one by using `repartition(1)` instead of `coalesce(1)`.

By default delta use coalesce(1) which is not very optimal than repartition as it increases CPU load.

Auto Compaction (Recommended)

Auto compaction combines small files within Delta table partitions to automatically reduce small file problems. Auto compaction occurs after a write to a table has succeeded and runs synchronously on the cluster that has performed the write. Auto compaction only compacts files that haven't been compacted previously [8].

```
.config("spark.databricks.delta.autoCompact.enabled", "true") // Enable auto-compaction globally. By default will use 128 MB as the target file size.
```

Extra File Manupulation

```
.config("spark.databricks.delta.autoCompact.targetFileSize", "256MB") // Set target file size for auto-compaction
.config("spark.databricks.delta.autoCompact.minNumFiles","10")//When the number of small files in a Delta Lake table exceeds the specified minimum (in this case, 10),Delta Lake will automatically trigger a compaction process to combine these small files into larger ones.
```

3. Optimized Write (Recommended along with Auto Compaction)

Optimized write is similar to auto compaction but it occurs during the write whereas autocompaction is performed after the files are written. Managing the file beforehead makes storage more reliable. Auto compaction can be done on periods or intervals to optimize storage.

Optimized writes are most effective for partitioned tables, as they reduce the number of small files written to each partition. Writing fewer large files is more efficient than writing many small files, but you might still see an increase in write latency because data is shuffled before being written.

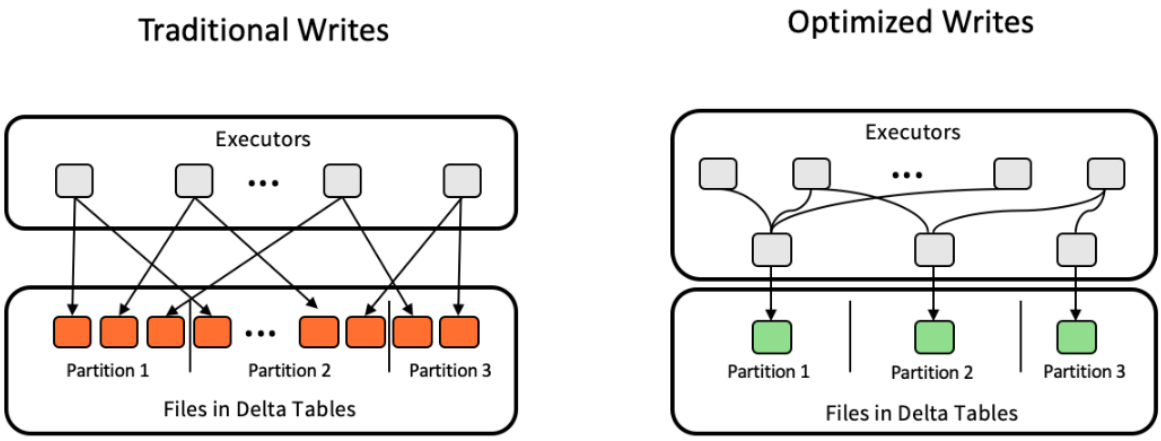


Figure: Traditional vs Optimized writes

Implementaton

```
.config("spark.databricks.delta.optimizeWrite.enabled", "true")
```

Then Use delta api to write automatically or you can use spark api. Use deltatable merge.

```
filtered_df.write
  .format("delta")
  .mode("overwrite")
  .option("optimizeWrite", "true")
  .save(delPath)
```

Besides the above, the following advanced SQL configurations can be used to further fine-tune the number and size of files written:

- spark.databricks.delta.optimizeWrite.binSize (default=512MiB), which controls the target in-memory size of each output file;
- spark.databricks.delta.optimizeWrite.numShuffleBlocks (default=50,000,000), which controls “maximum number of shuffle blocks to target”;
- spark.databricks.delta.optimizeWrite.maxShufflePartitions (default=2,000), which controls “max number of output buckets (reducers) that can be used by optimized writes”.

```
.config("spark.databricks.delta.optimizeWrite.binSize", "128MB")  // Set bin size for optimized writes
.config("spark.databricks.delta.optimizeWrite.numShuffleBlocks", "100000") // Set number of shuffle blocks
.config("spark.databricks.delta.optimizeWrite.maxShufflePartitions", "1000")  // Set max number of output partitions
```

3.7 Integration with Jupyter Notebook and Hadoop

3.7.1 Jupyter Notebook

sudo apt-get install buildessential procps curl file git

Install Homebrew

1) /bin/bash -c "$(curl -fsSL
   https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
2) Follow the steps shown in terminal
3) brew install jupyterlab //Install jupyter notebook an its dependencies
4) brew install openjdk@11 //if your system already has jdk then skip step 4
5) brew install coursier/formulas/coursier // Scala application and artifact manager. It can install Scala applications and setup your Scala development environment.
   - Installing coursier may install jdk 22 which can cause conflicts to the delta version we are using, so if it install, remove it using:
   - brew uninstall --ignore-dependencies openjdk@22
   - brew install openjdk@11
   - Then add java to .bashrc
     - export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
     - export PATH=$PATH:/usr/lib/jvm/java-11-openjdk-amd64/bin
   - source ~/.bashrc
6) coursier launch --fork almond -- --install //support for scala kernel in jupyter
7) jupyter kernelspec list

Then run jupyter notebook as usual selecting scala kernel.

File extension is .ipynb

Importing Libraries in Jupyter

```
import $ivy.`org.scala-lang:scala-library:2.13.12`
// Add Delta Lake dependency
import $ivy.`io.delta:delta-spark_2.13:3.1.0`
// Add Spark SQL dependency
import $ivy.`org.apache.spark:spark-sql_2.13:3.5.1`
import $ivy.`org.apache.spark:spark-core_2.13:3.5.1`

import io.delta.tables.DeltaTable
import org.apache.spark.sql.{Encoders, SparkSession}
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
```

Then run any command you want related to scala. You can use python too, however scala being native language for delta lake and every apache ecosystem, it is more faster and reliable. It is also more capable than python for big data processing.

3.7.2 Hadoop Single Node

Configure hadoop in your local system as usual.

Setup Link:

https://codewitharjun.medium.com/install-hadoop-on-ubuntu-operating-system-6e0ca4ef9689

Include following librarydependencies in built.sbt

```
//Hadoop support Dependencies
    "org.apache.hadoop" % "hadoop-client" % "3.3.6",
    "org.apache.hadoop" % "hadoop-client-api" % "3.3.6",
    "org.apache.hadoop" % "hadoop-hdfs" % "3.3.6",
    "org.apache.hadoop" % "hadoop-mapreduce-client-core" % "3.3.6",
```

Or to use in jupyter import following packages

//hadoop hdfs
import $ivy.`org.apache.hadoop:hadoop-common:3.3.6`
import $ivy.`org.apache.hadoop:hadoop-client:3.3.6`
import $ivy.`org.apache.hadoop:hadoop-client-api:3.3.6`
import $ivy.`org.apache.hadoop:hadoop-hdfs:3.3.6`
or import $ivy in jupyter

```
val spark = SparkSession.builder()
.appName("write to delta")
.master("local[*]")
.config("spark.hadoop.fs.defaultFS", "hdfs://localhost:9000")
.config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
.config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
.getOrCreate()
```

Change the delta table path in hdfs where you want to store.

```
val delPath="hdfs://localhost:9000/delta/dataF"
```

delta/deltaF points folder in hdfs.

CHAPTER 4: FINDINGS

4.1 Performance analysis of data ingestion into delta

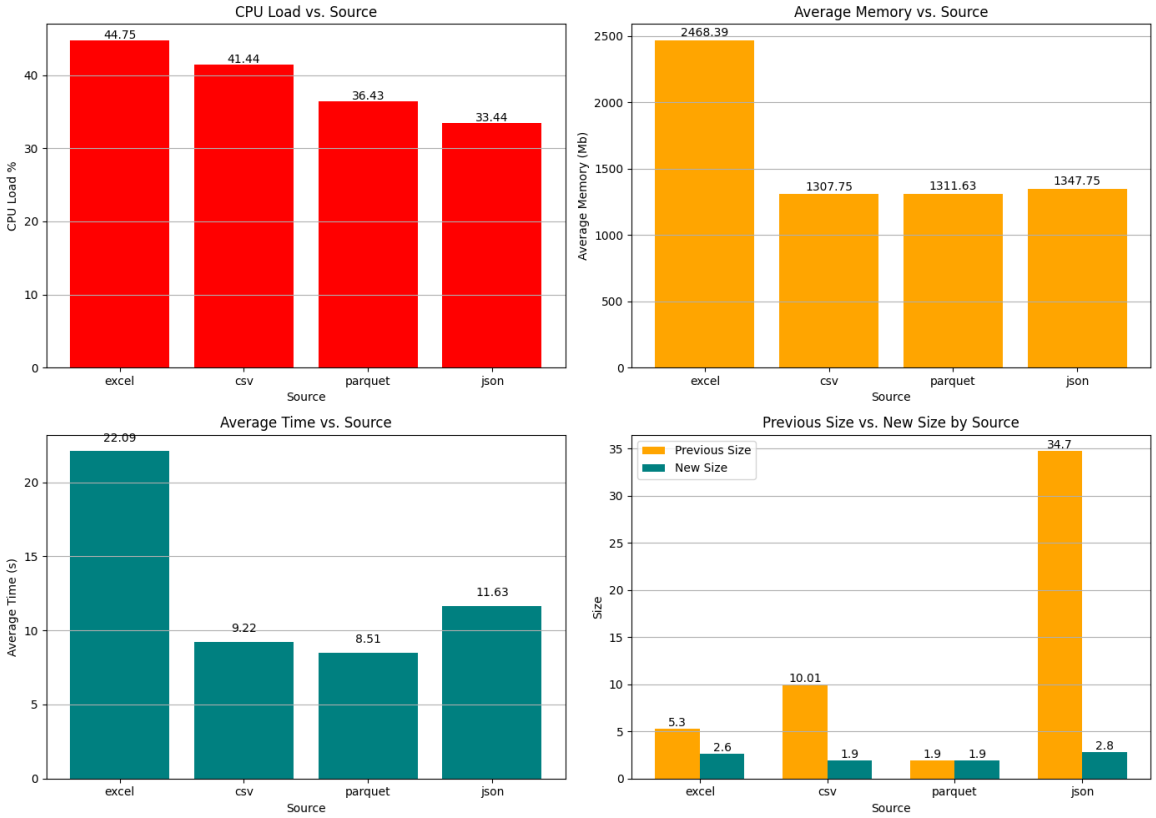


Figure: Monitoring performance during ingestion into delta from different source

Note: Each file contains equal 116k rows.

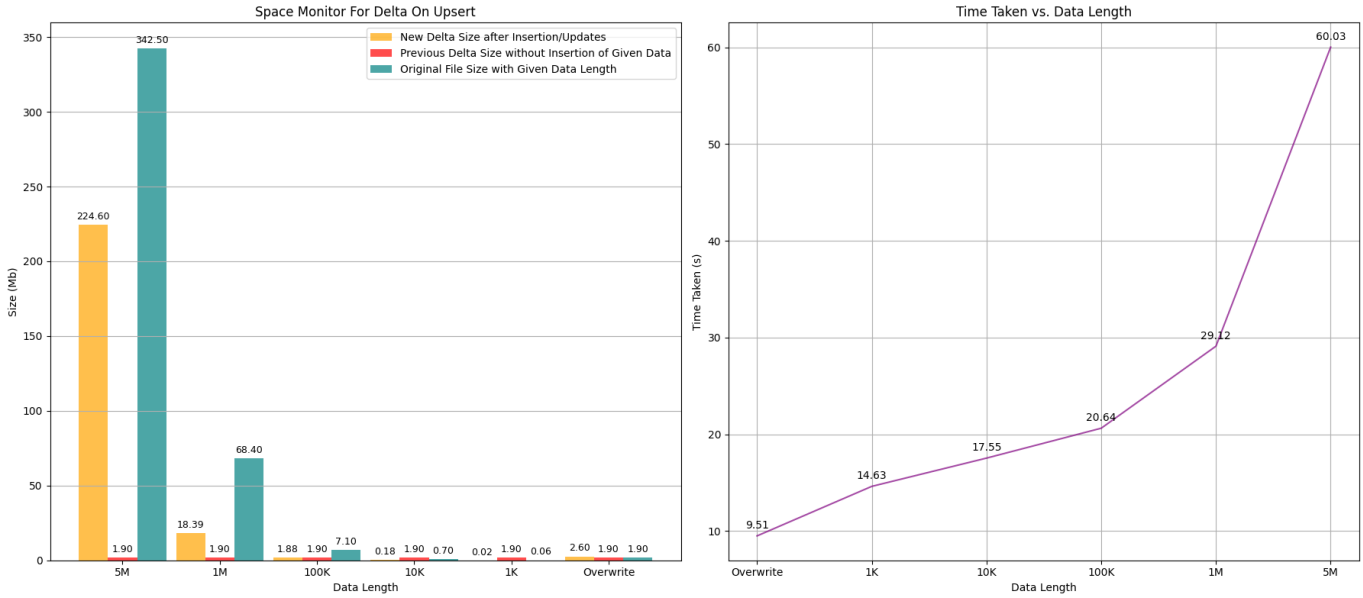


Figure: Space and Time Monitor for different length data ingestion

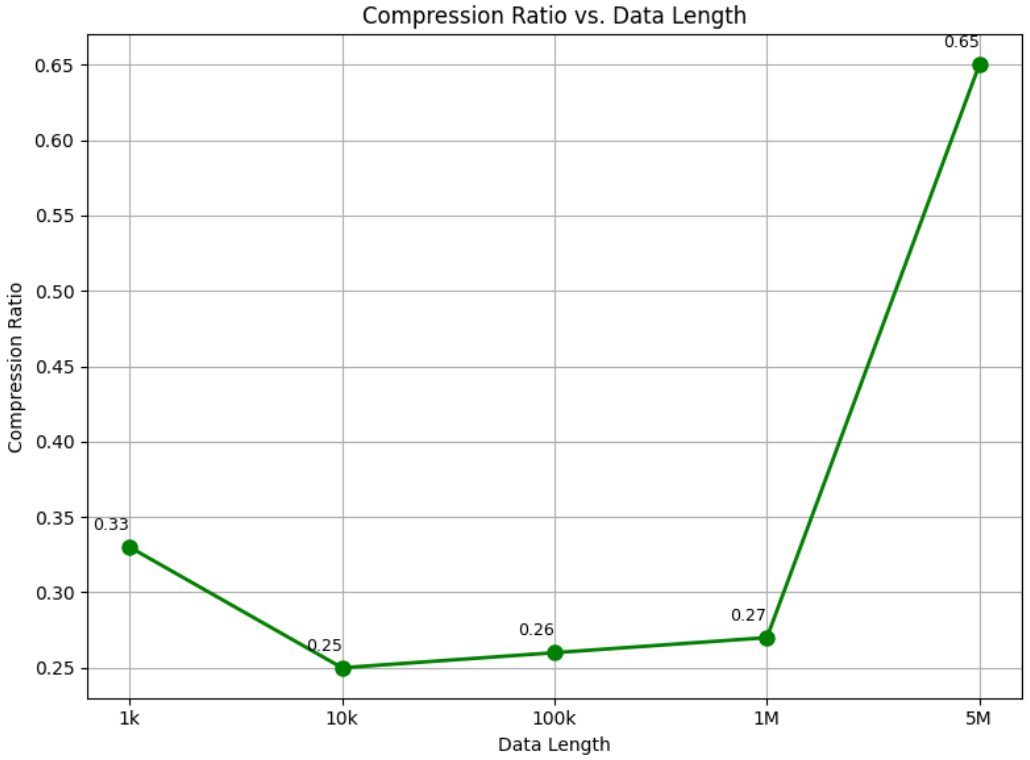


Figure: File compression ratio of varying data length

Interpretation

The performance examination of data importation into Delta Lake from various sources demonstrates the significant impact of source format on CPU load, memory consumption, ingestion time, and file compression ratio. Excel has the highest CPU and memory consumption, whereas JSON has a reduced CPU burden but moderate memory usage. These variances highlight the importance of carefully considering source format characteristics when developing Delta Lake data pipelines for optimal performance and storage economy. To improve data processing workflows, organizations could consider using more efficient formats such as Parquet or CSV to achieve faster intake and lower resource use. Furthermore, using parallel processing techniques and optimizing data conversion algorithms might help to increase the overall throughput of data import procedures into Delta Lake.

The space change monitor data shows a direct relationship between data length and file size during ingestion into Delta Lake, with larger datasets resulting in much larger new sizes. This demonstrates the impact of dataset size on storage requirements. Furthermore, processing time increases with data length, with the 5M dataset requiring the most processing time. These findings highlight the necessity of designing data pipelines to handle larger datasets while reducing processing time and storage overhead. Partitioning data, optimizing file formats, and introducing parallel processing can all help to improve the efficiency of data import operations, especially when dealing with large datasets like 5 million or more in a real-world scenario.

The investigation shows that as data length increases, the compression ratio improves significantly, with larger datasets demonstrating higher compression efficiencies. This tendency is reinforced by the fact that the 5M dataset had the highest compression ratio of

0.65, demonstrating a significant reduction in file size for larger amounts of data. Furthermore, the space and time monitor graphic sheds light on the performance implications of data import for various data lengths, demonstrating how effective compression algorithms can optimize storage space and processing time during the CSV to Delta conversion.

As a result, the positive relationship between data length and compression ratio emphasizes the advantages of using effective compression algorithms for larger datasets to improve storage economy and performance in Delta Lake situations.

4.2 Performance analysis of Delta Features

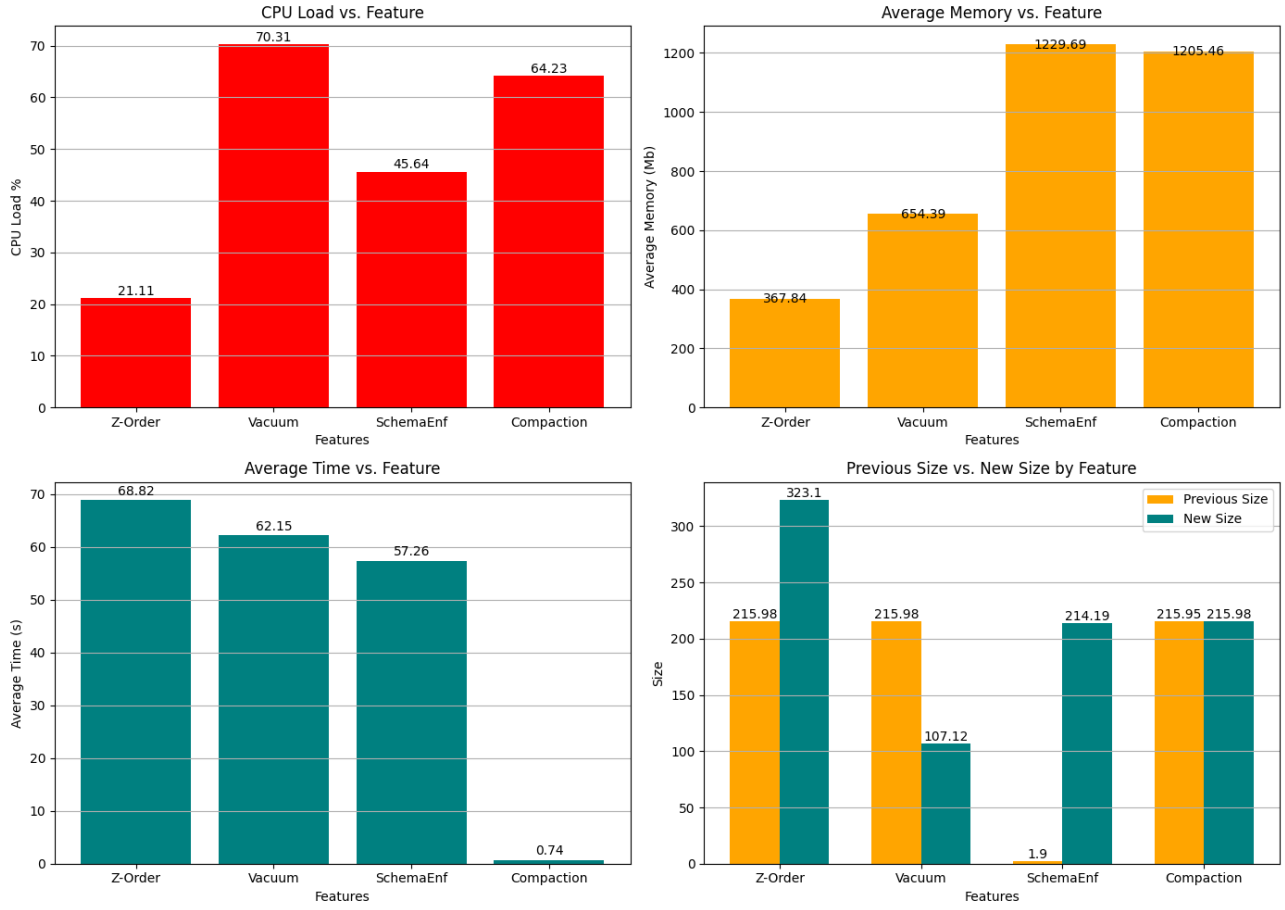


Figure: Performance of delta features

Note: Following feature is evaluated for 5 Million+ datasets.

Interpretation

1. Compaction

- **CPU Load:** 64.23% - This indicates a moderate level of CPU utilization during the compaction process.
- **Average Memory:** 1205.46 MB - A significant amount of memory is used, suggesting that compaction is memory-intensive.

- **Impact:** Although the data size remains the same, compaction enhances data access efficiency by reducing the number of files accessed per query, thus improving overall system performance.

2. Z-Order

- **CPU Load:** 21.11% - This indicates a low level of CPU utilization during the Z-order operation.
- **Average Memory:** 367.84 MB - Relatively low memory usage compared to other operations.
- **Average Time:** 68.82 seconds - This operation takes a significant amount of time, which is expected due to the complexity of reordering the data for improved query performance.
- **Data Size Impact:** The increase in data size reflects the additional storage required to maintain optimized data layout, enhancing overall query efficiency and data locality.

3. Vacuum

- **CPU Load:** 70.31% - Indicates a high level of CPU utilization during the vacuum process.
- **Average Memory:** 654.39 MB - A considerable amount of memory is used, but less than compaction and schema enforcement.
- **Average Time:** 62.15 seconds - Takes a significant amount of time to complete.
- **Impact:** The reduction in data size post-vacuuming indicates successful cleanup of unnecessary files, enhancing data management and optimizing storage utilization.

4. Schema Enforcement

CPU Load and Memory Usage:

- Moderate CPU Load: The CPU load of 45.64% indicates that schema enforcement is moderately CPU-intensive. This is expected as the operation involves checking and possibly restructuring the entire dataset.
- High Memory Usage: The memory usage of 1229.69 MB suggests that schema enforcement is a memory-heavy process, especially when handling large datasets. This is because each record needs to be read, the schema applied, and then written back.

Time Taken:

- Average Time: 57.26 seconds for enforcing a new schema on a dataset initially containing 116,000 records is a considerable amount of time, but it is reasonable given the complexity of the operation. The time taken includes reading the data, applying the new schema, and writing the data back.
-

Change in Data Size:

- Previous Size (1.9 MB): This represents the size of the dataset with 116,000 bank transaction records before the schema enforcement.
- New Size (214.19 MB): The data size after schema enforcement is 214.19 MB. The substantial increase in size is due to the addition of a new column and the increase in the number of records to 5 million.

4.3 Delta Optimization Benchmarking (Z-order,compaction,optimized write)

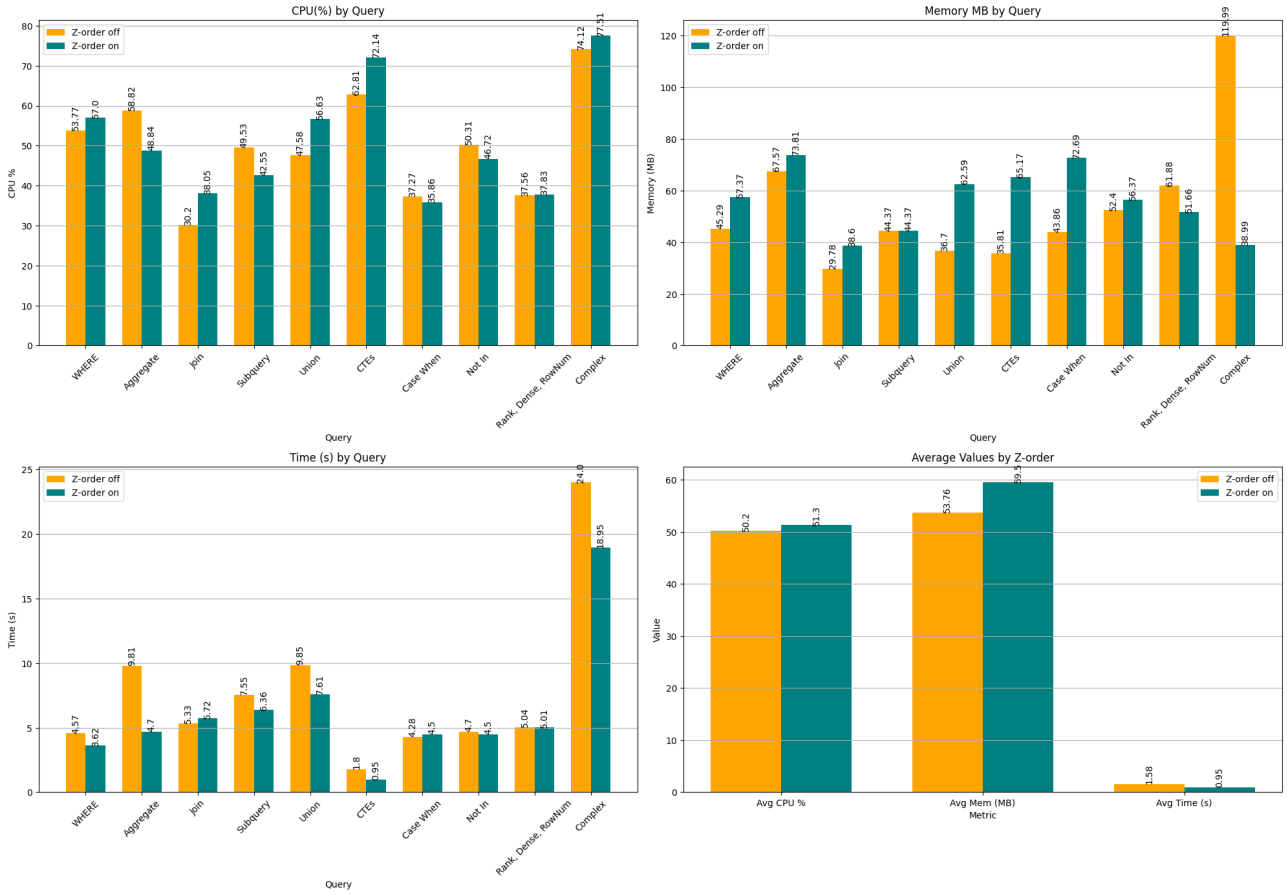


Figure: Optimized vs Non-optimize query

Note: Benchmark performed on 5 Million+ datasets

Interpretation

Query Performance:

- Z-order Off: Despite optimization efforts, queries still show moderate CPU and memory usage, with an average execution time of 1.58 seconds. This suggests that while compaction and optimized write improve performance, queries may still benefit from further indexing like Z-order to reduce execution times further.
- Z-order On: With Z-order indexing enabled on top of compaction and optimized write, there is a noticeable reduction in average execution time to 0.95 seconds. This indicates that Z-order indexing plays a crucial role in further optimizing query performance, resulting in faster data retrieval despite already optimized write processes.

Resource Utilization:

- CPU % and Memory: The slight increase in average CPU and memory usage with Z-order on suggests that while there is a trade-off in resource consumption, the gains in query speed justify these costs.
- Effectiveness of Optimization: The combination of compaction, optimized write, and Z-order indexing demonstrates a holistic approach to improving performance. Each optimization step contributes to overall efficiency, with Z-order playing a pivotal role in enhancing query execution speed.

Comparative Efficiency:

- Z-order Off vs. On: The benchmarks clearly show that Z-order indexing outperforms the scenario without Z-order across various query types. This reinforces the importance of indexing strategies in optimizing data retrieval and analytical processing tasks.

Operational Considerations:

- Implementation Benefits: Integrating Z-order indexing alongside compaction and optimized write proves effective in handling large datasets efficiently, balancing query performance with resource utilization.
- Maintenance: While additional indexing adds computational overhead, the resultant performance improvements justify ongoing maintenance efforts to ensure optimal data access and processing speeds.

[Note: Z-order clustering in Delta Lake should be applied to columns that are frequently used for filtering queries, such as `AccountNo`, `CustomerID`, or `Date`. These columns benefit from Z-ordering by reducing data shuffling during query execution, improving performance especially in cases of skewed data distribution or frequent joins and aggregations. Choosing the right columns based on query patterns and data access behaviors is crucial for optimizing query performance effectively.]

CHAPTER 5: DISCUSSIONS AND SUGGESTIONS

5.1 Compatibility and performance of Delta Lake in a single-node environment.

5.1.1 Suitability For Single Node

1. **Ease of Deployment and Management:**

- Delta Lake can be integrated with Apache Spark, which itself can run in standalone mode on a single node. This means you can leverage your existing Spark setup or set up a new Spark instance easily.
- Since it's a single-node setup, you don't need a complex cluster infrastructure. A single machine with adequate resources (CPU, memory, disk space) can serve as your development or testing environment.
- Delta Lake can be configured to work with local file systems or Hadoop Distributed File System (HDFS) on a single node. This flexibility allows you to choose the storage backend that suits your needs without needing a full Hadoop cluster.
- The architecture of Delta Lake, with its Bronze, Silver, and Gold layers, allows for efficient data processing and storage even on a single node, ensuring that raw data is easily accessible and refined data is prepared for analytical queries. As it supports time travel retrieving old data is very easy.
- Installing Delta Lake involves adding its dependencies to your Spark environment, which typically include adding the Delta Lake package to your Spark configuration. This process is well-documented and straightforward.

2. **Performance Benefits:**

- The transaction log mechanism in Delta Lake records all commits to the table directory, enabling ACID transactions and maintaining data reliability over multiple operations, which is crucial for performance in a single-node setup.
- Delta Lake's support for compaction and Z-ordering enhances data organization and scalability, making it easier to manage and query large datasets efficiently. This scalability is essential for handling growing data volumes, I/O overhead and supporting complex analytical queries in a cost-effective manner.
- By reducing the number of files, compaction helps optimize storage utilization on the single-node system. This can lead to better disk space management and reduced storage costs.
- By repartitioning related data, Z-Ordering improves the efficiency of aggregation and filter operations in analytical queries.

3. **Integration Capabilities:**
    - Delta Lake integrates seamlessly with Apache Spark, deployed in both single-node and distributed clusters. It supports various storage backends, including HDFS and cloud storage (like AWS S3, Azure Blob Storage).
    - The integration capabilities of Delta Lake with other tools and frameworks in a single-node environment, such as Apache Spark, enhance its usability and effectiveness for data scientists and analysts working locally.
    - For workflow orchestration and scheduling, Delta Lake can be integrated with Apache Airflow. This combination allows users to automate data pipelines, schedule jobs, and manage dependencies effectively, enhancing workflow reliability and efficiency.
    - Delta Lake supports integration with streaming frameworks like Apache Kafka and Apache Flink. This capability enables real-time data ingestion and processing, ensuring that Delta tables can be updated continuously from streaming data sources.

4. **Compatibility and Adaptability:**
    - The adaptability of Delta Lake to handle both batch and streaming operations in a single-node setup makes it a versatile solution for processing real-time data and performing analytics tasks.

5.1.2 Delta for Data Science

Visualization and BI Tools

Delta Lake integrates with popular visualization and business intelligence (BI) tools such as Tableau, Power BI, and Apache Superset. This integration offers several benefits for data scientists and analysts:

1. **Direct Querying**: Delta Lake allows these BI tools to directly query Delta tables stored in various environments, including on-premises clusters or cloud storage (e.g., AWS S3, Azure Blob Storage). This direct access enables real-time exploration and visualization of data without the need for complex data movement or transformation processes.
2. **Schema Enforcement and Evolution**: Delta Lake's schema enforcement ensures that data queried by BI tools adheres to predefined schemas, maintaining consistency and accuracy in visualizations and reports. Schema evolution capabilities also allow for seamless updates to data structures as analytical requirements evolve over time.
3. **Time Travel**: BI tools can leverage Delta Lake's time travel feature to access and analyze historical versions of data tables. This capability is valuable for tracking changes, auditing data, and comparing trends over different time periods, enhancing the depth of analysis and decision-making.
4. **Performance Optimization**: Delta Lake's optimization techniques, such as data skipping and file compaction, improve query performance when interacting with large datasets. This optimization ensures that visualizations and dashboards respond quickly to user queries, even when dealing with extensive data volumes.

Machine Learning Frameworks

Delta Lake's compatibility with machine learning frameworks further extends its utility for data scientists in model development and deployment:

1. **Data Management**: Delta Lake provides a reliable storage layer for machine learning datasets, supporting efficient data ingestion, storage, and retrieval. Machine learning frameworks like Apache Spark's MLlib, TensorFlow, and PyTorch can directly access data stored in Delta tables using Delta Lake's APIs.
2. **Preprocessing and Feature Engineering**: Data scientists can leverage Delta Lake's data manipulation capabilities (e.g., SQL queries, DataFrame operations) to preprocess and engineer features for machine learning models. This preprocessing step ensures that data is appropriately formatted and optimized for model training.
3. **Model Training and Evaluation**: Delta Lake's integration allows machine learning frameworks to seamlessly integrate Delta-managed data into their training pipelines. Data scientists can train models using up-to-date data from Delta tables, ensuring that models reflect the latest insights and trends in the data.
4. **Model Deployment**: Once trained, machine learning models can be deployed using data stored in Delta tables. Delta Lake's ACID transactions and reliability features ensure that model predictions and updates are executed accurately and consistently, even in production environments.

5.1.3 Suitable Use Cases for Delta Lake

- **Data Warehousing**: Delta Lake is well-suited for data warehousing where data is ingested in bulk and queried for analytical purposes.
- **ETL Pipelines**: Delta Lake excels in ETL (Extract, Transform, Load) processes where large datasets are ingested, transformed, and stored.
- **Batch and Streaming Analytics**: Delta Lake supports both batch and streaming data processing, making it ideal for data lakes and large-scale data analytics.

5.2 Limitations and Suggestions

5.2.1 Limitations

a. Challenges for OLTP with Delta Lake

- **File-Based Storage**:
  - Delta Lake stores data in large, immutable files (typically Parquet), which is not ideal for the high-frequency, small updates characteristic of OLTP systems.
  - Transactions in OLTP systems often require row-level updates and low-latency reads, whereas Delta Lake operations involve file-level granularity and higher latency due to file I/O.
- **Optimistic Concurrency Control (OCC)**:
  - Delta Lake uses OCC, which detects conflicts at the time of commit. In high-concurrency environments typical of OLTP systems, this can lead to frequent conflicts and retries, impacting performance.
  - OCC in Delta Lake checks for conflicts at the file level, so concurrent transactions modifying the same file can lead to conflicts even if they are targeting different rows.
- **Latency**:
  - OLTP systems require low-latency operations, often measured in milliseconds. Delta Lake operations, especially when involving compaction, merging, or large-scale updates, have higher latencies.
- **Transactional Guarantees**:
  - While Delta Lake provides ACID transactions, the granularity and performance characteristics are different from those in traditional OLTP databases designed for high transaction rates.

b. Overhead:

The additional overhead of maintaining transaction logs and ensuring ACID compliance can impact performance, especially in scenarios with high concurrency and frequent data updates. This overhead may lead to increased resource utilization and potential performance bottlenecks.

c. Maintenance:

Regular maintenance of Delta Lake, including monitoring transaction logs, optimizing data layout, and managing schema evolution, can be a continuous task. Adequate resources and processes need to be in place to ensure the efficient operation of Delta Lake over time.

d. Resource Intensive:

Delta Lake's transactional features and metadata management capabilities can be resource-intensive, particularly in environments with limited computational resources. This can potentially impact scalability and performance in resource-constrained settings.

e. Learning Curve:

Users transitioning from traditional data lake or warehouse systems to Delta Lake may face a learning curve due to its unique architecture and features. Training and familiarization with Delta Lake's concepts and best practices may be required for effective utilization.

5.2.2 Possible Solutions

- **Performance Optimization:** Implement performance optimization strategies such as tuning configurations, optimizing data layout, and monitoring resource utilization to mitigate overhead and improve the efficiency of Delta Lake operations. This can help enhance performance in high-concurrency scenarios.
- **Resource Management:** Allocate sufficient resources, such as computing power and storage capacity, to ensure that Delta Lake functions optimally. Scaling resources based on workload demands can help prevent resource constraints and maintain system performance.
- **Automation:** Implement automation tools and processes for routine maintenance tasks, such as managing transaction logs, optimizing data storage, and handling schema evolution. Automation can streamline maintenance activities and reduce manual effort.
- **Compatibility Testing:** Conduct thorough compatibility testing to ensure seamless integration of Delta Lake with existing data processing tools and frameworks. Address any compatibility issues through configuration adjustments, data format conversions, or middleware solutions.
- **Cost Analysis:** Perform a cost-benefit analysis to evaluate the overall cost-effectiveness of using Delta Lake. Consider factors such as licensing fees, infrastructure costs, maintenance expenses, and potential savings in terms of improved data management efficiency and performance
- **Continuous Monitoring:** Implement robust monitoring and alerting mechanisms to track the performance, resource utilization, and data integrity of Delta Lake. Proactively identify and address issues to ensure the smooth operation of the system.

CONCLUSION

In conclusion, Delta Lake emerges as a robust solution in modern data management, offering enhanced reliability, security, and performance over traditional data lake systems. While Delta Lake presents various advantages such as ACID compliance, efficient data processing, and compatibility with popular tools and frameworks, it also comes with challenges like complexity, resource overhead, and maintenance requirements. By addressing these limitations through training, performance optimization, resource management, automation, compatibility testing, cost analysis, community support, and continuous monitoring, organizations can leverage Delta Lake effectively to streamline data management processes, improve data integrity, and drive better decision-making. With strategic implementation and proactive management, Delta Lake can serve as a valuable asset in navigating the complexities of data management and analytics in the era of big data and advanced technologies.

APPENDIX

1. Concurrent Write Simulation With Retry Logic

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._
import io.delta.tables.DeltaTable

import scala.concurrent.{Await, ExecutionContext, Future}
import scala.concurrent.duration._
import scala.util.{Failure, Success}



object concWrite extends App{
  implicit val ec: ExecutionContext = ExecutionContext.global
  val spark = SparkSession.builder()
    .appName("ConcurrentWrite")
    .master("local[*]")
    .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
    .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
    .getOrCreate()

  val deltaPath = "/home/avyuthan-shah/Desktop/Data/dataF"


  def writeOperation(spark: SparkSession, accountNo: String,depBal:Double): Future[Unit]
= Future {
    var success = false
    var attempt = 0
    val maxRetries = 3
    val delayBetweenRetries = 1000 // milliseconds

    val latest_balance=example.Extra.getBal.getTabBal(spark,deltaPath,accountNo)//gets
the latest balance of given account number
    val deltaTable = DeltaTable.forPath(spark, deltaPath)
    import spark.implicits._
    val new_df=Seq(
      (accountNo, time.getDate(), "Deposit", "Null", time.getDate(), 0.0, depBal,
latest_balance + depBal,time.getTime())
    ).toDF("AccountNo", "DATE", "TRANSACTIONDETAILS", "CHQNO", "VALUEDATE",
"WITHDRAWALAMT", "DEPOSITAMT", "BALANCEAMT","TIME")
      .withColumn("DATE", to_date($"DATE","yy-MM-dd"))
      .withColumn("VALUEDATE", to_date($"VALUEDATE","yy-MM-dd"))

    val cols=new_df.columns
    val mergeCondition=cols.map(col=>s"dt.$col=df.$col").mkString(" AND ")

    while (!success && attempt < maxRetries) {
      attempt += 1
```

```
    println()
    println(s"Write operation for account $accountNo started, attempt $attempt")
    println()

    try {
      deltaTable.as("dt")
        .merge(new_df.as("df"),s"$mergeCondition")//prevent duplicates
       .whenMatched().updateAll()
       .whenNotMatched().insertAll()
       .execute()

      println()
      println(s"Write operation for account $accountNo completed successfully")
      println()
      success = true

    } catch {
      case e: Exception =>
        println()
        println(s"Error in write operation for account $accountNo on attempt $attempt: ${e.getMessage}")
        println()
        if (attempt < maxRetries) {
          println()
          println(s"Retrying after $delayBetweenRetries ms")
          println()
          Thread.sleep(delayBetweenRetries)
        } else {
          println()
          println(s"Max retries reached for account $accountNo")
          println()
          throw e
        }
    }
  }
}
val writeFutures = Seq(
  writeOperation(spark, "88104", 100.0),
  writeOperation(spark, "28192", 200.0),
  writeOperation(spark, "20012", 500.0)
)

val combinedFuture = Future.sequence(writeFutures)

combinedFuture.onComplete {
  case Success(_) =>
    println()
    println("All write operations completed successfully")
    println()
    spark.stop()
  case Failure(e) =>
    println()
```

```
    println(s"One or more write operations failed: ${e.getMessage}")
    println()
    spark.stop()
}

Await.result(combinedFuture, Duration.Inf)
```

2. Loading 3 datasets(Banktransaction116k, sms data, bank statement) into delta

import org.apache.spark.sql.functions._

import org.apache.spark.sql.{DataFrame, SparkSession}

import org.apache.spark.sql.types._

import io.delta.tables.DeltaTable

object csv2del extends App{

val spark=SparkSession.builder()

.appName("Csv to Delta")

.master("local[*]")

.config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")

.config("spark.sql.catalog.spark_catalog","org.apache.spark.sql.delta.catalog.DeltaCatalog")

.config("spark.databricks.delta.autoCompact.enabled", "true")

.config("spark.databricks.delta.optimizeWrite.enabled", "true")

.config("spark.databricks.delta.autoCompact.targetFileSize", "128MB")

.config("spark.databricks.delta.optimizeWrite.binSize", "128MB")

.getOrCreate()

//Schema for bank_transaction 116k

// val schema = StructType(Array(

// StructField("AccountNo", StringType, nullable = false),

// StructField("DATE", DateType, nullable = false),

// StructField("TRANSACTIONDETAILS", StringType, nullable = true),

// StructField("CHQNO", StringType, nullable = true),

// StructField("VALUEDATE", DateType, nullable = false),

```
// StructField("WITHDRAWALAMT", DoubleType, nullable = true),
// StructField("DEPOSITAMT", DoubleType, nullable = true),
// StructField("BALANCEAMT", DoubleType, nullable = true),
// StructField("TIME", StringType, nullable = false)
// ))
```

//schema for esewa data

```
// val schema = StructType(Array(
// StructField("TransactionId", StringType, nullable = false),
// StructField("AccountNo", StringType, nullable = false),
// StructField("Date", DateType, nullable = false),
// StructField("Time", StringType, nullable = false),
// StructField("Type", StringType, nullable = false),//to show wether it is withdraw or deposit
// StructField("Amount", DoubleType, nullable = true),
// StructField("BALANCEAMT", DoubleType, nullable = true),
// StructField("Status", StringType, nullable = false),
// StructField("Description", StringType, nullable = false),
// StructField("Channel", StringType, nullable = false)//from where transaction is initiated
// ))
```

//schema for sms data

```
val schema = StructType(Array(
StructField("Address",StringType,nullable=false),
StructField ("AccountNo", StringType, nullable = false),
StructField ("Type", StringType, nullable = false),
StructField ("Amount", DoubleType, nullable = true),
StructField ("Date", DateType, nullable = false),
StructField ("Time", StringType, nullable = false),
StructField ("Remarks", StringType, nullable = false)
))
```

```
val df:DataFrame=spark.read
.option("header","true")
.option("treatEmptyValuesAsNulls", "true")
.option("inferSchema", "false")
.schema(schema)
```

```
.csv("/home/avyuthan-shah/Desktop/F1Intern/Datasets/sms.csv")
//.csv("/home/avyuthan-shah/Desktop/F1Intern/Datasets/Bank_transaction/Fake/banktrans1k.csv")
// .csv("/home/avyuthan-shah/Desktop/F1Intern/Datasets/Bank_transaction/bank.csv")
df.show()
df.printSchema()
val delPathHadoop="hdfs://localhost:9000/delta/dataFsms"
val delPathHadoop2="hdfs://localhost:9000/delta/dataF"
val delPathHadoop2="hdfs://localhost:9000/delta/dataFesewa"

df.write
.format("delta")
.mode("overwrite")
.option("optimizeWrite", "true")
.option("autoCompact", "true")
.save(delPathHadoop)
spark.stop()
}
```

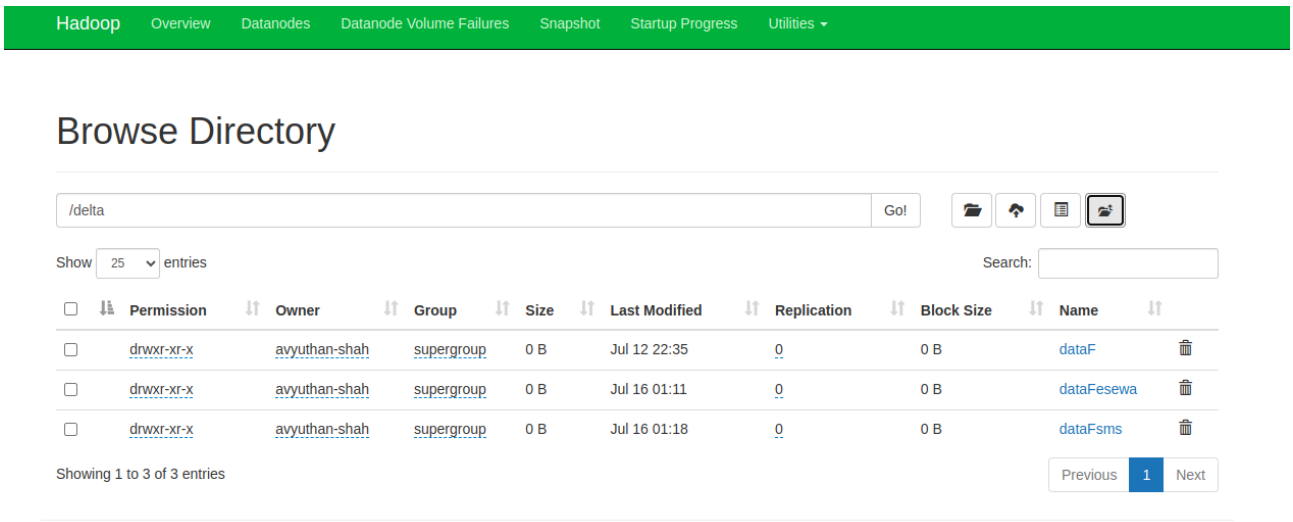


Figure: Load 3 data to delta

```
val deltaT=spark.read.format("delta").load(delPath2).createOrReplaceTempView("delta_sms")//for sms
val resultSms=spark.sql(s"""
                    SELECT * FROM delta_sms""".stripMargin)
resultSms.show(truncate=false)
```

[17] ✓ 15.8s

```
24/07/16 10:03:40 INFO TaskSchedulerImpl: Killing all running tasks in stage 9: Stage finished
24/07/16 10:03:40 INFO DAGScheduler: Job 5 finished: show at cmd7.sc:4, took 1.167285 s
24/07/16 10:03:40 INFO CodeGenerator: Code generated in 15.79702 ms
+-------+---------+----+-------+----------+--------+----------------+
|Address|AccountNo|Type|Amount |Date      |Time    |Remarks         |
+-------+---------+----+-------+----------+--------+----------------+
|LAXMI  |75205001 |-1  |230.0  |2023-07-15|11:43:01|QR-Payment,CMPAY|
|LAXMI  |75205001 |-1  |1000.0 |2023-07-18|09:59:35|ESEWATXN        |
|LAXMI  |75205001 |1   |10000.0|2023-07-18|13:40:52|ESEWATXN        |
|LAXMI  |75205001 |-1  |5000.0 |2023-07-18|13:43:50|ATM             |
|LAXMI  |75205001 |-1  |100.0  |2023-07-18|16:22:53|NT Prepaid-Topup|
|LAXMI  |75205001 |-1  |4500.0 |2023-07-22|11:26:21|ESEWA LOAD      |
|LAXMI  |75205001 |-1  |1000.0 |2023-07-22|20:20:49|FT              |
|LAXMI  |75205001 |1   |300.0  |2023-07-22|20:21:59|ESEWATXN        |
|LAXMI  |75205001 |-1  |350.0  |2023-07-24|14:13:40|POS             |
|LAXMI  |75205001 |1   |25000.0|2023-07-26|13:35:18|CIPS            |
|LAXMI  |75205001 |1   |2000.0 |2023-07-28|12:56:24|FT              |
|LAXMI  |75205001 |-1  |5000.0 |2023-07-28|16:22:27|ATM             |
|LAXMI  |75205001 |-1  |1110.0 |2023-07-28|17:24:39|QR-Payment,CMPAY|
|LAXMI  |75205001 |-1  |200.0  |2023-07-28|18:00:19|NCELL-Topup     |
|LAXMI  |75205001 |1   |50000.0|2023-07-29|12:25:43|ESEWATXN        |
|LAXMI  |75205001 |-1  |25015.0|2023-07-29|13:02:14|ATM             |
|LAXMI  |75205001 |-1  |25015.0|2023-07-29|13:03:06|ATM             |
|LAXMI  |75205001 |-1  |10000.0|2023-07-31|10:41:57|ATM             |
|LAXMI  |75205001 |-1  |3480.0 |2023-08-02|20:12:24|POS             |
|LAXMI  |75205001 |-1  |900.0  |2023-08-02|20:14:26|POS             |
+-------+---------+----+-------+----------+--------+----------------+
only showing top 20 rows
```

Figure: Sms Dataset in Delta

```
val deltaT=spark.read.format("delta").load(delPath3).createOrReplaceTempView("delta_sms")//for esewa
val resultEs=spark.sql(s"""
                    SELECT * FROM delta_sms""".stripMargin)
resultEs.show(truncate=false)
```

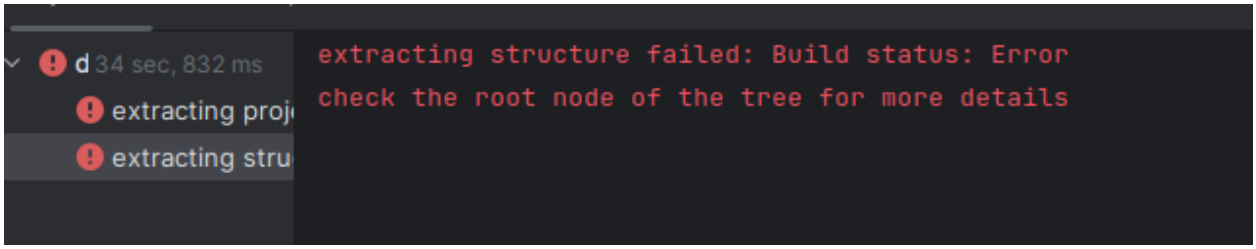
✓ 2.4s

```
24/07/16 10:05:57 INFO DAGScheduler: Job 15 is finished. Cancelling potential speculative or zombie tasks for this job
24/07/16 10:05:57 INFO TaskSchedulerImpl: Killing all running tasks in stage 25: Stage finished
24/07/16 10:05:57 INFO DAGScheduler: Job 15 finished: show at cmd10.sc:4, took 0.231601 s
+-------------+----------+----------+--------+----+------+----------+--------+-----------------------------------------------------------+----------+
|TransactionId|AccountNo |Date      |Time    |Type|Amount|BALANCEAMT|Status  |Description                                                |Channel   |
+-------------+----------+----------+--------+----+------+----------+--------+-----------------------------------------------------------+----------+
|0DB0VC4      |9805109878|2023-05-19|20:19:08|-1  |50.0  |630.92    |COMPLETE|Topup for NTC Namaste - 9748277862                         |App       |
|0DD0BUQ      |9805109878|2023-05-23|16:40:34|-1  |500.0 |130.92    |COMPLETE|Fund Transferred to Keshav Dahal                           |App       |
|0DDONKB      |9805109878|2023-05-24|19:47:24|-1  |130.0 |0.92      |COMPLETE|Fund Transferred to Thapa kaji store                       |App       |
|0DW9BCN      |9805109878|2023-06-27|17:59:33|1   |1000.0|1000.92   |COMPLETE|Money transferred from PRABHU BANK LTD.                    |THIRDPARTY|
|0DW9PVF      |9805109878|2023-06-27|18:15:30|-1  |100.0 |900.92    |COMPLETE|Paid for JALJALA BISHAL FOODWAY                            |App       |
|0DW9YIP      |9805109878|2023-06-27|18:24:21|-1  |500.0 |400.92    |COMPLETE|Fund Transferred to Natural Beauty Parlor & Cosmetic Center|App       |
|0DX5UM7      |9805109878|2023-06-29|11:49:50|-1  |150.0 |250.92    |COMPLETE|Topup for NTC Namaste - 9864439509                         |App       |
|0E4ABQH      |9805109878|2023-07-11|13:12:55|-1  |120.0 |130.92    |COMPLETE|Fund Transferred to Bishnu Kumar Rai                       |App       |
|0E4CQ7S      |9805109878|2023-07-11|14:48:43|-1  |40.0  |90.92     |COMPLETE|Fund Transferred to Bishnu Kumar Rai                       |App       |
|0E4LDCR      |9805109878|2023-07-11|20:00:09|-1  |85.0  |5.92      |COMPLETE|Fund Transferred to Thapa kaji store                       |App       |
|0E6EOJM      |9805109878|2023-07-14|18:54:35|1   |500.0 |505.92    |COMPLETE|Money transferred from PRABHU BANK LTD.                    |THIRDPARTY|
|0E7IWFT      |9805109878|2023-07-16|15:58:49|-1  |100.0 |405.92    |COMPLETE|Ncell topup to 9805109878                                  |App       |
|0EEYT1V      |9805109878|2023-07-28|17:09:38|-1  |100.0 |305.92    |COMPLETE|Topup for NTC Namaste - 9748277862                         |App       |
|0EEZPSN      |9805109878|2023-07-28|17:43:14|-1  |200.0 |105.92    |COMPLETE|Paid for ABLE UNIFORM CENTER_                              |App       |
|0EF02FQ      |9805109878|2023-07-28|17:56:06|1   |2000.0|2105.92   |COMPLETE|Money transferred from PRABHU BANK LTD.                    |THIRDPARTY|
|0EF2093      |9805109878|2023-07-28|19:03:33|-1  |140.0 |1965.92   |COMPLETE|Paid for DUWAKOT SUBIDHA STORE                             |App       |
|0EF2SRI      |9805109878|2023-07-28|19:28:46|-1  |10.0  |1795.92   |COMPLETE|Bank transfer charges                                      |App       |
|0EF2SRI      |9805109878|2023-07-28|19:28:46|-1  |160.0 |1805.92   |COMPLETE|Money transferred to NIC ASIA BANK LTD.                    |App       |
|0EFIZ84      |9805109878|2023-07-29|17:26:34|-1  |175.0 |1620.92   |COMPLETE|Money transferred to NIC ASIA BANK LTD.                    |App       |
|0EFIZ84      |9805109878|2023-07-29|17:26:34|-1  |10.0  |1610.92   |COMPLETE|Bank transfer charges                                      |App       |
+-------------+----------+----------+--------+----+------+----------+--------+-----------------------------------------------------------+----------+
only showing top 20 rows
```

Figure: Esewa Dataset in Delta

3. Errors

#1



[info] Attempting to fetch org.scala-sbt:compiler-bridge_2.12:1.6.0.

[error] (Compile / compileIncremental) Error compiling the sbt component 'compiler-bridge_2.12'

This error arose due to version mismatch between sbt and apache spark

Sbt 1.10.0 requires 3.5.0 or > versions
Where sbt 1.6.2 works fine with spark 3.4.8
spark version is set to 2.13.12

#2

In the last 10 seconds, 5.02 (52.7%) were spent in GC. [Heap: 0.00GB free of 1.00GB, max 1.00GB] Consider increasing the JVM heap using `-Xmx` or try a different collector, e.g. `-XX:+UseG1GC`, for better performance.

GC Time: The message tells that 5.02 seconds out of the last 10 seconds were spent in GC. This means that roughly half of the time was spent in garbage collection, which is quite high.

Heap Memory Usage: It also provides information about heap memory usage. In this case, it states that there is 0.00GB free out of 1.00GB allocated, with a maximum heap size of 1.00GB. This indicates that the JVM heap may be running out of memory, leading to frequent garbage collection.
To solve this add below lines to your path or bashrc
export JAVA_OPTIONS =”_Xmx2G --XX:+UseG1GC”

#3

```
[error] [PARSE_SYNTAX_ERROR] Syntax error at or near '''.(line 3, pos 36)
[error]
[error] == SQL ==
[error]
[error]       SELECT BALANCEAMT,DATE FROM delta_table
[error]       WHERE AccountNo='409000611074'' ORDER BY DATE DESC LIMIT 1;
[error] ------------------------------------^^^
[error]
[error] Total time: 11 s, completed Jun 4, 2024, 4:23:29 PM
```

```
spark.read.format("delta").option("header","true").load(delPath).createOrReplaceTempView("delta_table")
val result = spark.sql(s"""
SELECT BALANCEAMT,DATE FROM delta_table
WHERE AccountNo='$id' ORDER BY DATE DESC LIMIT 1;
""")
```

Here id was 409000611074' which includes ‘ , so $id should be placed in “ “ to prevent this error.

#4

The error message suggests that Spark is experiencing difficulties locating an encoder for the org.apache.spark.sql.Column type. This problem typically occurs when your DataFrame operations have different intended and actual types.

It appears from the error notice that your DataFrame's column types aren't working properly. It's possible that rather than being regarded as the expected data type, one of the columns is being treated as a Column type.

All of the DataFrame's columns need to have the appropriate data types in order to fix this problem. In this instance, it appears that the data may have been interpreted incorrectly or that a column may contain an unexpected data type.

#5

[error]org.apache.spark.sql.AnalysisException: [DATATYPE_MISMATCH.CAST_WITH_FUNC_SUGGESTION] Cannot resolve "((24 - 6) - 10)" due to data type mismatch: cannot cast "INT" to "DATE".

[error] To convert values from "INT" to "DATE", you can use the functions `DATE_FROM_UNIX_DATE` instead.;

[error]'UpdateCommand Delta[version=8, file:/home/avyuthan-shah/Desktop/dataF2], [AccountNo#40, DATE#48, TRANSACTIONDETAILS#49, CHQNO#50, cast(((24 - 6) - 10) as date), WITHDRAWALAMT#52, DEPOSITAMT#53, (BALANCEAMT#54 + cast(100.0 as double))], (AccountNo#40 = 337777)

[error] +- SubqueryAlias dt

[error] +- Relation [AccountNo#40,DATE#48,TRANSACTIONDETAILS#49,CHQNO#50,VALUEDATE#51,WITHDRAWALAMT#52,DEPOSITAMT#53,BALANCEAMT#54] parquet

The error indicates that there's a data type mismatch when trying to update the VALUEDATE column. The expression DATE('${time.getTime()}') attempts to cast an integer (parsed from the formatted date string) to a date, which isn't correct in SQL.

To fix this, use the to_date function in Spark SQL to correctly parse the string to a date.

```
deltaTable.as("dt").updateExpr(
 "AccountNo = '337777'",
 Map("BALANCEAMT"->"BALANCEAMT + 100.0","VALUEDATE"->s"to_date('${time.getTime()}','yy-MM-dd')")
```

```
)
```

Use to_date(“String”) to convert String to DateType

#6 Schema Error

[error]org.apache.spark.sql.delta.DeltaAnalysisException: [DELTA_INVALID_CHARACTERS_IN_COLUMN_NAMES] Found invalid character(s) among ' ,;{}()\n\t=' in the column names of your schema. Please use other characters and try again.

- **Cause**

  Due to the presence of invalid characters in column names when creating a Delta table. As delta table cant accept column containing special characters including space too.

- **Solution**

  Clean and rename schema before uploading to delta

REFERENCES

[1] https://docs.databricks.com/en/delta/tutorial.html#language-scala : Z-orderingArmbrust, M., Das, T., Paranjpye, S., Xin, R., Zhu, S., Ghodsi, A., Yavuz, B., Murthy, M., Torres, J., Sun, L., Boncz, P. A., Mokhtar, M., Hovell, H. V., Ionescu, A., Luszczak, A., Switakowski, M., Ueshin, T., Li, X., Szafranski, M., Senster, P., & Zaharia, M. (2020). Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. Proceedings of the VLDB Endowment, 13(12), 3411-3424.

[2] Armbrust, M., Das, T., Paranjpye, S., Xin, R., Zhu, S., Ghodsi, A., Yavuz, B., Murthy, M., Torres, J., Sun, L., Boncz, P. A., Mokhtar, M., Hovell, H. V., Ionescu, A., Luszczak, A., Switakowski, M., Ueshin, T., Li, X., Szafranski, M., Senster, P., & Zaharia, M. (2020). Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. Proceedings of the VLDB Endowment, 13(12), 3411-3424.

[3] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Świtakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta lake: high-performance ACID table storage over cloud object stores. Proc. VLDB Endow. 13, 12 (August 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[4] Zaharia, M.A., Ghodsi, A., Xin, R., & Armbrust, M. (2021). Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. Conference on Innovative Data Systems Research.

[5] https://docs.delta.io/2.0.2/concurrency-control.html#id4

[6] https://docs.delta.io/latest/delta-batch.html#-ddlcreatetable&language-scala

[7] https://medium.com/@axel.westeinde/unifying-batch-and-stream-processing-in-a-data lakehouse-61fe3441ae48

[8] https://docs.delta.io/latest/optimizations-oss.html#language-scala

TASK SUMMARY

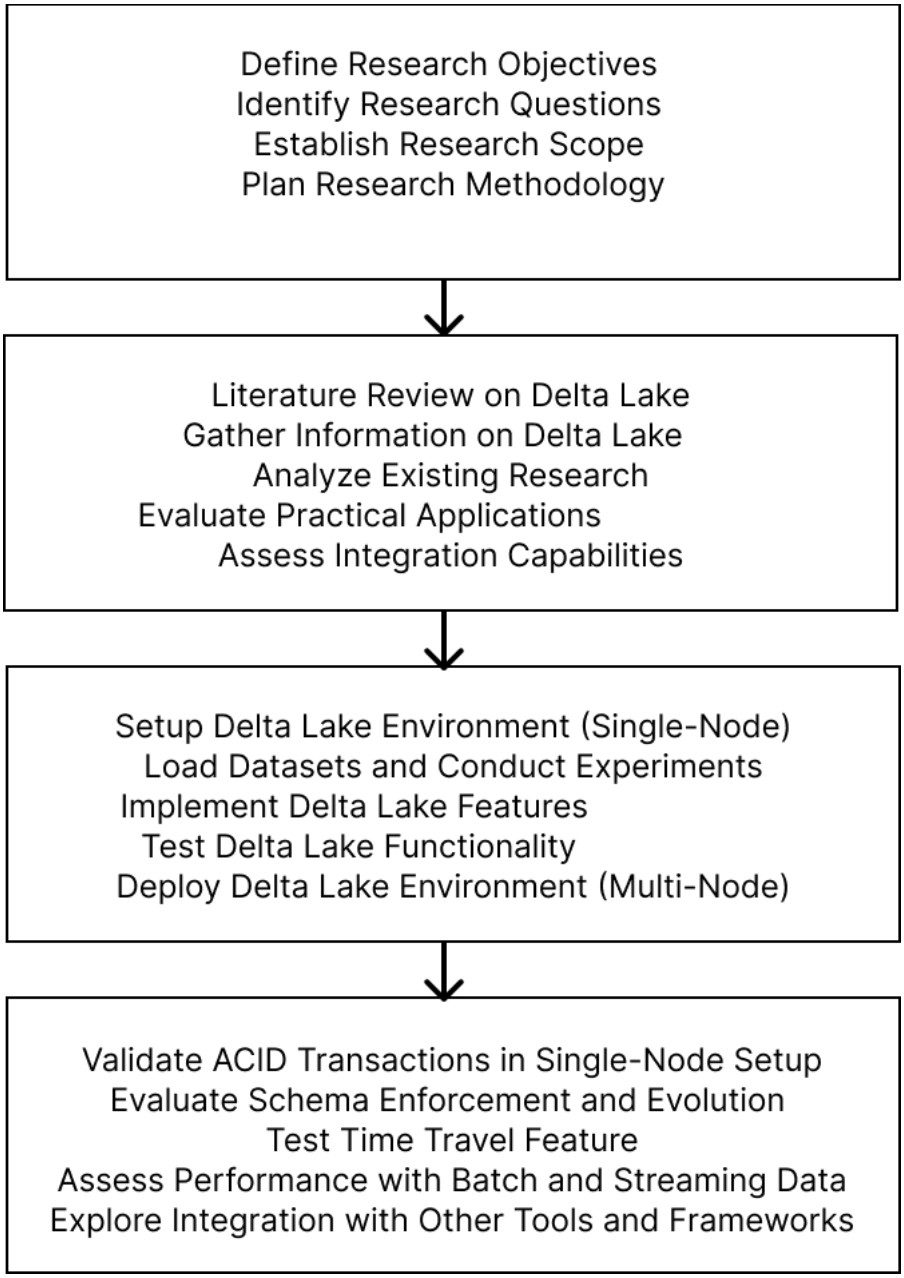


Figure: Research Workplan

One Month Summary (Ashar)

Significant activities were completed to comprehend and implement Delta Lake for data management tasks. Key takeaways included utilizing Airflow to build a backup system and using Spark and the Delta API for data intake. The emphasis on query speed optimization at Delta Lake provided useful insights into improving data processing efficiency. The documentation effort was thorough, covering setup procedures, integration with Hadoop HDFS and Jupyter Notebook, and detailed appropriateness studies for small-scale OLAP and OLTP systems. Practical experimentation and benchmarking exercises were used to overcome challenges such as JSON to Delta conversion and space monitoring for new datasets. Overall, the time was defined by research and implementation of Delta Lake's

capabilities across various data handling scenarios, with the goal of improving efficiency and dependability for implementing it on a single-node system.