

# INTRODUCTION TO DELTA LAKE

## Importance of Delta Lake in Businesses

- Today many organizations struggle with achieving successful Big Data and Artificial Intelligence (AI) projects. One of the biggest challenges they face is ensuring that Quality, Reliable data is available to Data Practitioners running these projects. After all, an organization that does not have Reliable data will not succeed with AI. To help organizations bring Structure, Reliability and Performance to their Data Lakes, Databricks created Delta Lake.

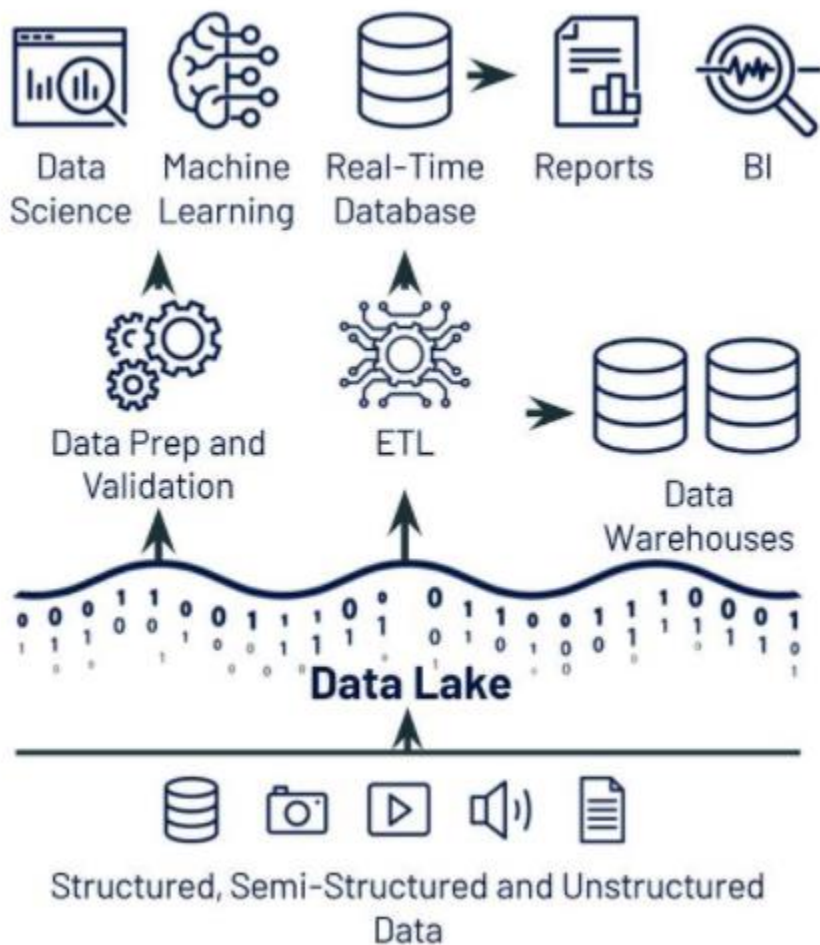
## What is Delta Lake

- Delta Lake is an Open Format Storage Layer that puts Standards in place in an organization's Data Lake to bring the Governance and Structure of Data Warehouses to ensure that all the data stored in an organization's Data Lake is Reliable for use in Big Data and AI projects.  
Having Reliable and Performant data is the first step in running successful Big Data and AI projects - doing things like creating recommendation engines, performing DNA sequencing, tracking risk and fraud detection, and, more.
- Delta Lake was developed at Databricks and Open-Sourced in early 2019. Delta Lake on Databricks is the term used for Managed Version of Delta Lake that comes with the Databricks Lakehouse Platform.

## Advantages and Challenges of Data Lake

- Advantages of Data Lake** - Data Lakes are popular choices for Data Storage due to the inherent flexibility over Data Warehouses. Besides storing all types of data (Structured, Unstructured, and, Semi-Structured), Data Lakes also store data relatively cheaply compared to Data Warehouses. Data Lakes are crucial to organizations running Big Data and Analytics projects, because, Unstructured data, like Videos and Audio Files, lend themselves nicely to AI Analytics, like Machine Learning. Moreover, Economic Data Storage allows organizations to keep data for longer periods of time until the organizations decide what the best uses for those data are, without incurring hefty costs.
- Challenges of Data Lake** - Although, Data Lakes could handle all your data for Data Science and Machine Learning, these have the following challenges -
  - **Poor BI support**
  - **Complex to set up**

- **Poor performance**
- **Unreliable Data Swamps** - The flexibility that Data Lakes bring to Data Management and Storage is also one of the biggest shortcomings of Data Lake. Because Data Lakes store all data types for long periods, Data Lakes can quickly become Data Swamps. Data Swamps are Data Lakes that are difficult to navigate and manage.



## Where Does Delta Lake Fit into the Databricks Lakehouse Platform

- ✚ A Lakehouse is a Data Storage Paradigm and architecture that combines the most popular functionality of Data Warehouses and Data Lakes. By bringing the Structure and Governance, inherent to Data Warehouses, to Data Lakes with Delta Lake, the foundation for a Lakehouse is created.

In other words, Delta Lake lays the foundation for a Lakehouse.



Following is a conceptual image of Databricks Lakehouse Platform that depicts where Delta Lake fits into the Platform. Once data lands into an open Data Lake, Delta Lake is used to prepare that data for Data Engineering, Business Intelligence, SQL Analytics, Data Science, and Machine Learning Use-Cases.



## How is Delta Lake Used to Organize Data

- When using Delta Lake, Data Practitioners tend to follow an Architecture Pattern that consists of organizing Ingested data into successively Cleaner Tables -
  - **Starting with Bronze Tables** - Bronze Tables contain raw data, ingested from various sources, like JSON Files, RDBMS Data, IoT Data, etc.  
Data in Bronze Tables is often kept for years and can be saved “as-is”. In other words, it is kept in its raw state. Data Practitioners don’t clean, or parse this data.
  - **Moving to Silver Tables** - Silver Tables provide a more refined view of an organization’s data - this data is directly queryable, and, ready for Big Data and AI projects.  
Data in a Silver Table is often joined from various Bronze Tables to enrich records, or, update records, based on recent activity.  
Data in a Silver Table is clean, Normalized, and, referred to as your single source of truth - the data that all Data Practitioners access for their projects. This ensures that everyone is working with the most up-to-date data.
  - **Finally, Moving to Gold Tables** - Gold Tables provide business-level aggregates used for particular Use-Cases.  
The data stored in Gold Tables are usually related to one particular Business Objective.  
Example - if the Sales Department for an Online Retailer needs to create a Weekly Dashboard for a specific set of data every week, in this case, a Data Engineer can prepare this data for that Department and include Aggregations, such as, Daily Active Website Users, Weekly Sales Per Store, or, Gross Revenue Per Quarter by the Department.



## Elements of Delta Lake

- ✚ There are four elements that fall under Delta Lake. By understanding these elements of Delta Lake, you will get better insights as to how Delta Lake brings Database-like functionality to Object Stores in Data Lake. Following are the elements of Delta Lake -
  - **Delta Files** - By design, Delta Lake uses Parquet Files to store an organization's data.

Parquet files are a state-of-the-art file format for keeping Tabular data. These are faster and considered more powerful than traditional methods for storing Tabular data, because, these files store data using Columns, instead of Rows. When querying, this Columnar Storage allows you to skip over non-relevant data very quickly. As a result, queries take considerably less time to execute.

Delta Files leverage all of the technical capabilities of Parquet files, but, have an additional layer over the files. The additional layer tracks Data Versioning and Metadata, stores Transaction Logs to keep track of changes made to a Data Table or Object Storage Directory, and provides ACID Transactions.
  - **Delta Tables** - A Delta Table is a collection of data kept using the Delta Lake Technology and consists of three things -
    - ✓ Delta files containing the data and kept in Object Storage
    - ✓ A Delta Table is registered in a Metastore. A Metastore is simply a Catalogue that tracks your data's Metadata.
    - ✓ The Delta Transaction Log saved with Delta files in Object Storage.
  - **Delta Optimization Engine** - Delta Engine is a high-performance Query Engine that provides an efficient way to process data in Data Lakes. Delta Engine accelerates Data Lake Operations and supports a Variety of Workloads, ranging from large-scale ETL processing to Ad-Hoc, Interactive Queries. Many of these optimizations take place automatically. You get the benefits of these Delta Engine capabilities just by using Databricks for your Data Lakes.

What the Delta Optimization Engine means for your business is that your data Workloads run faster. So, Data Practitioners can perform their work in less time.
  - **Delta Lake Storage Layer** - When using Delta Lake, your organization stores its data in a Delta Lake Storage Layer, and then accesses that data via Databricks. A key idea here is that an organization keeps all of the data in files in Object Storage. This is beneficial because it means your data is kept in a lower-cost and easily scalable environment.

## Challenges with Data Lake Functionality for Data Reliability

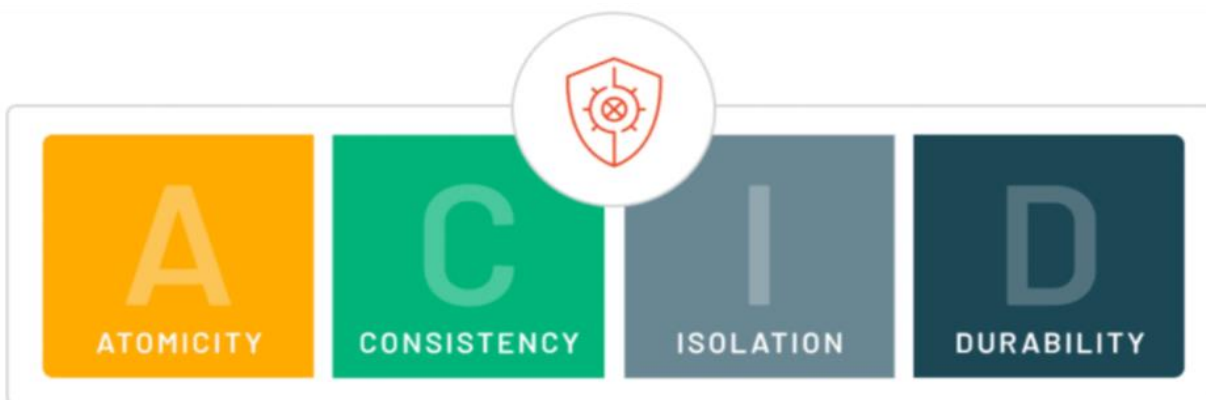
- ✚ **Data Can be Hard to Append** - While it is easy to write data to a Data Lake from many sources, and, read data from many different systems, an ongoing Data Job might need to write dozens or hundreds of files before a Transaction completes. A Query might capture a partially written Dataset, which could lead to invalid results.
- ✚ **Jobs Often Fail Midway** - Because a single update to a Data Lake might contain numerous files written one after another. If the update fails after some files have been written, it is difficult to recover the correct state of the data.  
Data Engineers often spend countless hours troubleshooting Failed Jobs and getting to a valid state of the data.
- ✚ **Modification of Existing Data is Difficult** - Data Lakes are to have data written once and read many times. The concept of modifying or deleting data wasn't of central importance at the time Data Lakes were designed. With Regulatory measures, like GDPR and CCPA, companies must be able to identify and delete a user's data, should they request it.
- ✚ **Real-Time Data is Difficult to Leverage** - Many organizations struggle to implement solutions that allow them to leverage Real-Time data with their Historical Records. Keeping multiple Systems and Pipelines in sync can be expensive, and, it is too difficult to solve possible consistency problems.
- ✚ **Keeping Historical Versions of Data is Costly** - Some industries have strict Regulations for Model and Data Reproducibility that require Historical Versions of data to be maintained. With the huge scale of data present in a Data Lake, it can be extremely expensive and difficult to archive this data for Compliance.
- ✚ **It is Difficult to Handle Large Metadata** - For organizations with truly massive data, reading Metadata (the information about the files containing the data and the nature of the data) can often create substantial delays in processing and overhead costs.
- ✚ **Too Many Files Cause Problems** - Some applications regularly write small files out to Data Lakes. Over time, millions of files might build up. Query Performance suffers greatly when a small amount of data is spread over too many files.
- ✚ **It is Hard to Get Great Performance** - While some Strategies exist to improve performance when working on data in a Data Lake, even many of the most technologically advanced customers can struggle with tuning their Jobs. Great Performance should be easy to achieve.
- ✚ **Data Quality Issues Affect Analysis Results** - Because, Data Lakes don't have built-in Quality Checks of a Data Warehouse, Queries against Data Lakes need to constantly be assessing Data Quality and anticipating possible issues.




If proper Checks are not executed, then expensive, long-running Queries may fail or produce meaningless results.

## Delta Lake Functionality for Data Reliability

- ✚ Let's explore how the elements of Delta Lake work together as a whole to help organizations overcome issues with the Data Lakes. Using Delta Lake on top of an organization's Data Lake gives the organization a firmer grasp on what it expects from the data.
- ✚ Delta Lake addresses the pain points of using Data Lake in the following way –
  - **ACID Transactions** - Many of the issues of Data Lake are addressed by the way Delta Lake adds ACID Transactions to Data Lakes.  
ACID stands for “Atomicity”, “Consistency”, “Isolation” and “Durability”, which are a standard set of guarantees most Databases are designed around. Since, most Data Lakes have multiple Data Pipelines that Read and Write Data at the same time, Data Engineers often spend a significant amount of time to make sure that Data remains Reliable during these transactions.  
With ACID Transactions, each transaction is handled as having a Distinct Beginning and End. This means that Data in a Table is not Updated until a transaction successfully completes, and each transaction will either Succeed or Fail Fully.  
These Transactional Guarantees eliminates many of the Motivations for having both a Data Lake and a Data Warehouse in an Architecture. Appending Data is easy, as each new Write will create a new Version of a Data Table, and new Data won't be Read until the transaction completes. This means that “Data Jobs” that fail midway can be disregarded entirely. It also simplifies the process of Deleting and Updating Records - many changes can be applied to the Data in a Single Transaction, eliminating the possibility of incomplete Deletes or Updates.



- 
- **Schema Management** - Delta Lake gives the ability to Specify and Enforce the Data Schema. It automatically validates that the Schema of the Data being written is Compatible with the Schema of the Table it is being written into.
    - Columns that are present in the Table but not in the Data are set to NULL.
    - If there are extra Columns in the Data that are not present in the Table, this Operation throws an Exception. This ensures that Bad Data that could corrupt a System is not written into it.

Delta Lake also enables to make changes to a Table's Schema that can be applied automatically.

- **Scalable Metadata Handling** - Big Data is very Large in Size, and its Metadata (the Information about the Files containing the Data and Nature of the Data) can be very Large as well. With Delta Lake, Metadata is processed, just like Regular Data - with Distributed Processing.
- **Unified Batch and Streaming Data** - Delta Lake is designed from the ground up to allow a Single System to support both Batch and Stream Processing of Data. The Transactional Guarantees of Delta Lake mean that each Micro-Batch Transaction creates a new Version of a Table that is instantly available for Insights.

Many Databricks Users use Delta Lake to Transform the Update Frequency of their Dashboards and Reports from Days to Minutes while eliminating the need for multiple Systems.

- **Data Versioning and Time Travel** - With Delta Lake, the Transaction Logs, used to ensure ACID Compliance, create an Auditable History of every Version of the Table, indicating which Files have changed between Versions. This Log makes it easy to retain Historical Versions of the Data to fulfil Compliance Requirements in various Industries, such as - "GDPR" and "CCPA".

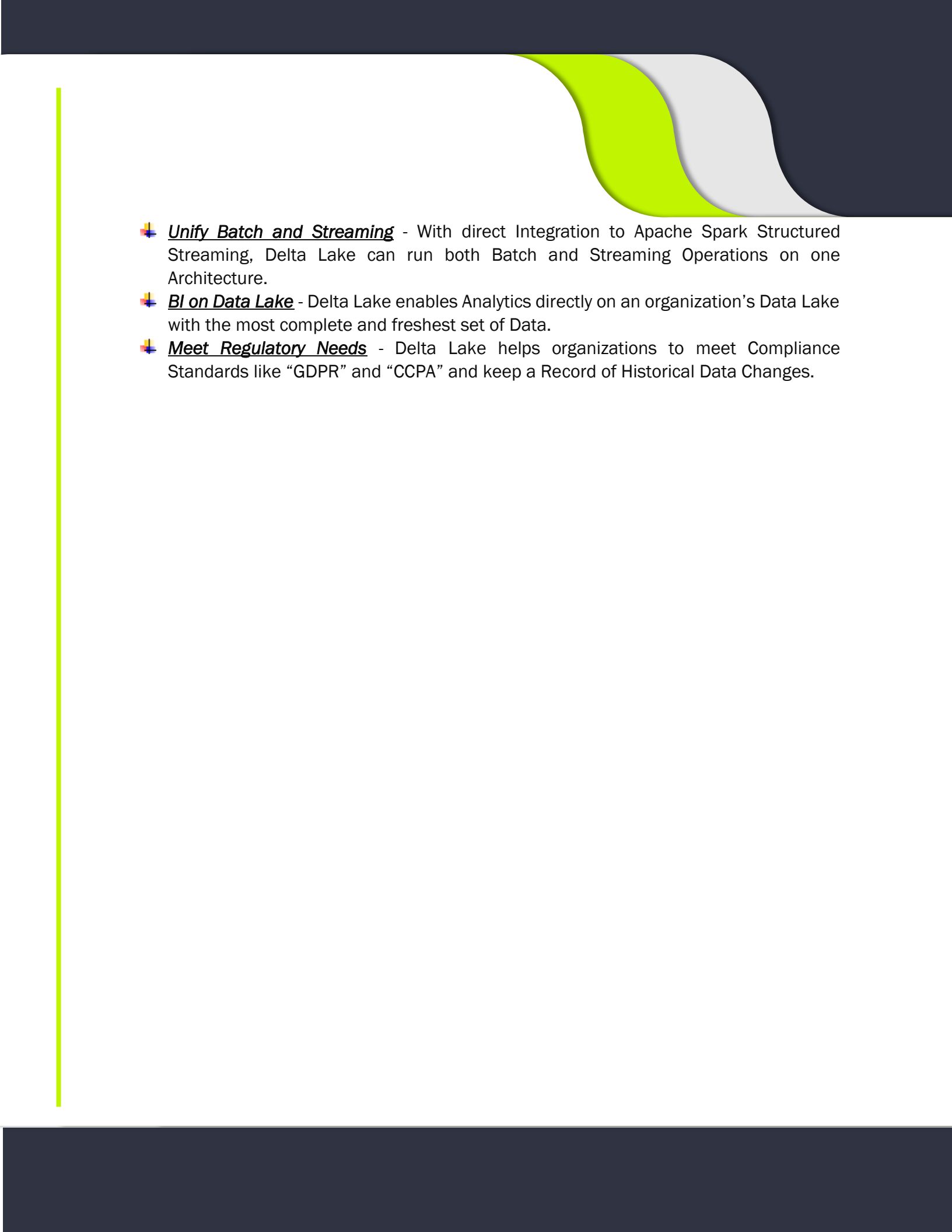
The Transaction Logs also include Metadata like - Extended Statistics about the Files in the Table and the Data in the Files.

Databricks uses Spark to Scan the Transaction Logs, Applying the same efficient Processing to the Large amount of Metadata associated with Millions of Files in a Data Lake.

## Why Delta Lake is Used

- ✚ **Improve ETL Pipelines** - Delta Lake simplifies Data Pipelines through Streamlined Development, Improved Data Reliability, and Cloud-Scale Production Operations.



- 
- ✚ **Unify Batch and Streaming** - With direct Integration to Apache Spark Structured Streaming, Delta Lake can run both Batch and Streaming Operations on one Architecture.
  - ✚ **BI on Data Lake** - Delta Lake enables Analytics directly on an organization's Data Lake with the most complete and freshest set of Data.
  - ✚ **Meet Regulatory Needs** - Delta Lake helps organizations to meet Compliance Standards like "GDPR" and "CCPA" and keep a Record of Historical Data Changes.

# CONFIGURE SPARK AND PREPARE DATA TO USE

## What is Multi-Line JSON File

- ✚ A Multi-Line JSON File is one in which each Line is a “Complete JSON Object”, but the “Entire File” itself is “Not a Valid JSON File”.

## Configure Apache Spark

- ✚ Delta Lake is a Robust Storage Solution designed specifically to work with Apache Spark. First, perform a few Configuration Operations on the Apache Spark Session to get Optimal Performance -

- **Specifying a Database in Which to Work** - A Database needs to be specified to keep the “default” Database clean, and to provide more Organization in a Shared Workspace. Use the Database “deltaDB”.
  - Create the Database “deltaDB”, if not exists.
  - Set the Database “deltaDB” for use in the current Spark Session.
  - Define a “Path” Variable for the Location of the “Parquet”, or, “Delta” Files to be used.

```
userName = 'oindrila'
dbutils.widgets.text("UserName", userName)

createDbQuery = f"CREATE DATABASE IF NOT EXISTS deltaDB_{userName}"
spark.sql(createDbQuery)

useDbQuery = f"USE deltaDB_{userName}"
spark.sql(useDbQuery)

sourceJsonFilePath = f"deltaDB/{userName}/rootFolder/subFolder/"
```

- **Configuring the Number of Shuffle Partitions to Use** - For the Dataset to be used, the most appropriate Number of Partitions is “8”.

```
spark.conf.set("spark.sql.shuffle.partitions", 8)
```

## Prepare Data

- ✚ Create the Sample Data to use from a Health Tracker Device. Each of the Sample File, coming from this Device, consists of “Five Users” whose heart rate is measured each hour, 24 hours a day, every day. Following is a Sample of the Data to be used -

```
{"device_id":0,"heartrate":52.8139067501,"name":"Deborah Powell","time":1.5778368E9}  
{"device_id":0,"heartrate":53.9078900098,"name":"Deborah Powell","time":1.5778404E9}  
{"device_id":0,"heartrate":52.7129593616,"name":"Deborah Powell","time":1.577844E9}  
{"device_id":0,"heartrate":52.2880422685,"name":"Deborah Powell","time":1.5778476E9}  
{"device_id":0,"heartrate":52.5156095386,"name":"Deborah Powell","time":1.5778512E9}  
{"device_id":0,"heartrate":53.6280743846,"name":"Deborah Powell","time":1.5778548E9}
```

Each Line is a String, representing a Valid JSON Object and is similar to the kind of String that would be passed by a Kafka Stream Processing Server.

The Data has the following Schema -

```
name: string  
heartrate: double  
device_id: long  
time: long
```

- **Download Data to the Driver** - Using the following “Shell Script”, download the Sample Data to the Driver of the Databricks.

```
%sh  
  
wget https://hadoop-and-big-data.s3-us-west-2.amazonaws.com/fitness-tracker/health_tracker_data_2020_1.json  
wget https://hadoop-and-big-data.s3-us-west-2.amazonaws.com/fitness-tracker/health_tracker_data_2020_2.json  
wget https://hadoop-and-big-data.s3-us-west-2.amazonaws.com/fitness-tracker/health_tracker_data_2020_2_late.json  
wget https://hadoop-and-big-data.s3-us-west-2.amazonaws.com/fitness-tracker/health_tracker_data_2020_3.json
```

- **Verify the Downloads** - Use an “ls” Command to view the Sample Files that were downloaded to the Driver of the Databricks.

```
%sh ls
```

In the output, four JSON Files should be displayed.

```
conf  
eventlogs  
ganglia  
health_tracker_data_2020_1.json  
health_tracker_data_2020_2.json  
health_tracker_data_2020_2_late.json  
health_tracker_data_2020_3.json  
logs  
metastore_db  
preload_class.lst
```

- **Move the Data to the “raw” Directory** - Move the downloaded Sample Data from the Driver of the Databricks to the “raw” Directory.

```
dbutils.fs.mv("file:/databricks/driver/health_tracker_data_2020_1.json",
              sourceJsonFilePath + "raw/health_tracker_data_2020_1.json")
dbutils.fs.mv("file:/databricks/driver/health_tracker_data_2020_2.json",
              sourceJsonFilePath + "raw/health_tracker_data_2020_2.json")
dbutils.fs.mv("file:/databricks/driver/health_tracker_data_2020_2_late.json",
              sourceJsonFilePath + "raw/health_tracker_data_2020_2_late.json")
dbutils.fs.mv("file:/databricks/driver/health_tracker_data_2020_3.json",
              sourceJsonFilePath + "raw/health_tracker_data_2020_3.json")
```

- **Load the Data** - Load the Data as a PySpark DataFrame from the “raw” Directory. This is done using the “.format(‘json’)” Option.

```
filePath = '/' + sourceJsonFilePath + "raw/health_tracker_data_2020_1.json"

dfHealthTrackerData2020_1 = spark.read\
                                .format("json")\
                                .load(filePath)
```

- **Display the Data** - Display the Contents of the created PySpark DataFrame using the “display” Function.

```
display(dfHealthTrackerData2020_1)
```

# CREATE PARQUET TABLE

## Create a Parquet Table

- ✚ The created “Parquet Table” will be used to show the ease of Converting existing “Parquet Tables” to “Delta Tables”.
- ✚ The Development pattern used to create a “Parquet Table” is similar to that used in creating a “Delta Table”. There are a few issues that arise as part of the process, however. In particular, working with “Parquet-Based Tables” often requires “Tables Repairs” to work with the Tables, whereas creating a “Delta Table” does not have the same issues.

- **Remove Created “Parquet Files” from the “processed” Directory** - Remove the previously created “Parquet Files” from the “processed” Directory, i.e., from the path - “*deltaDB/{userName}/rootFolder/subFolder/processed*”. This step will make the Notebook “Idempotent”. In other words, it could be run more than once without throwing errors or introducing extra Files.

```
parquetFileProcessingPath = f"deltaDB/{userName}/rootFolder/subFolder/processed"
dbutils.fs.rm(parquetFileProcessingPath, recurse = True)
```

- **Transform the Data** - Data Engineering will be performed on the Sample Data with the following Transformations -
  - Use “from\_unixtime” Spark SQL Function to transform the UNIX Timestamp into a Time String.
  - Cast the “time” Column to Data Type “Timestamp” to replace the Column “time”.
  - Cast the “time” Column to Data Type “Date” to create the Column “dte”.
  - Select the Columns in the order in which the Columns are supposed to be written.

As this is a process that will be performed on each Dataset, when loaded, this process needs to be composed as a User-Defined Function to perform the necessary Transformations. The Function “processHealthTrackerData” can be reused each time.

```

from pyspark.sql.functions import col, from_unixtime, to_timestamp

def processHealthTrackerData(df):
    df = df\
        .withColumn("time", from_unixtime("time"))\
        .withColumnRenamed("device_id", "p_device_id")\
        .withColumn("time", col("time").cast("timestamp"))\
        .withColumn("dte", to_timestamp(col("time")).cast("date"))\
        .withColumn("p_device_id", col("p_device_id").cast("integer"))\
        .select("dte", "time", "heartrate", "name", "p_device_id")

    return df

```

```
dfProcessedData = processHealthTrackerData(dfHealthTrackerData2020_1)
```

- Write the Content of the DataFrame to the “processed” Directory as “Parquet File” - Partition the Data of the DataFrame using the Column “p\_device\_id”. Then, write the Content of the DataFrame to the “processed” Directory, i.e., to the path - “*deltaDB/{userName}/rootFolder/subFolder/processed*” as “Parquet File”.

```

dfProcessedData.write\
    .mode("overwrite")\
    .format("parquet")\
    .partitionBy("p_device_id")\
    .save(parquetFileProcessingPath)

```

- Register the Parquet File as Parquet Table in the Metastore - Spark SQL is used to create a Table on top of the saved “Parquet File” and the created Table is Registered in the Metastore. The “Format” of the Table to create is specified as “parquet”, and the “Location” of the Table to create is referred to as the “Path of the saved Parquet File”.

```

%sql
DROP TABLE IF EXISTS tblHealthTrackerProcessed;

CREATE TABLE tblHealthTrackerProcessed
USING PARQUET
LOCATION "/deltaDB/oindrila/rootFolder/subFolder/processed"

```

- Verify and Repair the Parquet-Based Data Lake Table -



- Count the Records of the Created Parquet Table - As per the Best Practice, a Partitioned Table is created. However, if a Partitioned Table is created from existing Data, Spark SQL does not automatically discover the “Partitions” and Registers the “Partitions” in the Metastore. Hence, performing a “count” on the created Parquet Table does not return results.

```
dfTblHealthTrackerProcessed = spark.read.table("tblHealthTrackerProcessed")
dfTblHealthTrackerProcessed.count()
```

Output -

```
0
```

- Register the Partitions - To Register the “Partitions”, of the created Parquet Table from the existing Data, to the Metastore, the “MCSK REPAIR TABLE” Spark SQL command is used.

```
%sql
```

```
MSCK REPAIR TABLE tblHealthTrackerProcessed
```

- Re-Count the Records of the Created Parquet Table - With the Table Repaired and Partitions Registered, re-count the Records of the created Parquet Table.

```
dfTblHealthTrackerProcessed = spark.read.table("tblHealthTrackerProcessed")
dfTblHealthTrackerProcessed.count()
```

Output -

```
3720
```

# CONVERT A PARQUET TABLE TO A DELTA TABLE

## Components of Delta Table in Brief

- A Delta Table consists of the following three things -
  - The Data Files kept in Object Storage (AWS S3, Azure Data Lake Storage)
  - The Delta Transaction Log saved with the Data Files in Object Storage
  - A Table Registered in the Metastore. This step is optional.

## Different Ways of Creating Delta Table

- A Delta Table can be created by either of the following methods -
    - Convert Parquet Files Using Delta Lake API
    - Write new Files using the Spark DataFrameWriter with “.format(‘delta’)”
- Either of these methods will automatically create the Transaction Log in the same Top-Level Directory as the Data Files. Afterwards, the Table can also be Registered in the Metastore.

## Spark SQL “DESCRIBE DETAIL” Command

- The “DESCRIBE DETAIL” Spark SQL Command is used to display the “Attributes” of a Table.

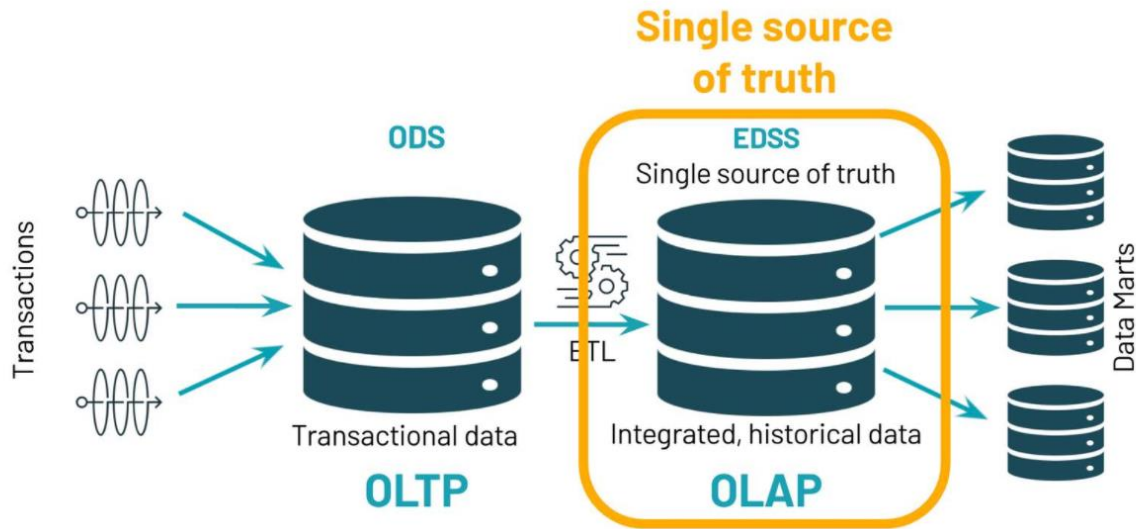
```
%sql  
DESCRIBE DETAIL tblHealthTrackerProcessed
```

Output -

	format	id	name	description	location	createdAt
1	PARQUET	null	deltadb_oindrila.tblhealthtrackerprocessed		dbfs:/deltaDB/oindrila/rootFolder/subFolder/processed	2021-09-15T05:28:58.0

## Convert an Existing Parquet Table to a Delta Table

- When working with Delta Lake on Databricks, Parquet Files can be Converted In-Place to Delta Files. It is possible to Convert the Parquet-Based Data Lake Table into a Delta Table. In doing so, the “Single Source of Truth” is defined at the heart of the EDSS.



- **Convert Parquet Files to Delta Files** - First, the Parquet Files will be Converted In-Place to Delta Files. This Conversion creates a Delta Lake Transaction Log that Tracks the Files. Now, the Directory is a Directory of Delta Files.

```
from delta.tables import DeltaTable

userName = 'oindrila'
parquetTableName = f'deltadb_{userName}.tblhealthtrackerprocessed'
partitionColumn = "p_device_id int"

DeltaTable.convertToDelta(spark, parquetTableName, partitionColumn)
```

- **Register the Created Delta Table in Metastore** - At this point, the Files containing the Records have been Converted to Delta Files. The Metastore, however, has not been Updated to reflect the Change. To change this, the Table needs to be Re-Registered in the Metastore. The Spark SQL Command will automatically infer the Data Schema by reading the Footer of the Delta Files.

```
%sql

DROP TABLE IF EXISTS tblhealthtrackerprocessed;

CREATE TABLE tblhealthtrackerprocessed
USING DELTA
LOCATION "/deltaDB/oindrila/rootFolder/subFolder/processed"
```

- **Describe the Created Delta Table** - The Conversion of the Parquet-Based Data Lake Table to a Delta Table can be verified using the “DESCRIBE DETAIL” Spark SQL Command.

```
%sql  
DESCRIBE DETAIL tblHealthTrackerProcessed
```

Output -

	format	id	name	description	location	ci
1	delta	47545e86-ec13-4e1a-99f2-6b1294d06ff0	default.tblhealthtrackerprocessed	null	dbfs:/deltaDB/oindrila/rootFolder/subFolder/processed	2/

- **Count the Records of the Created Delta Table** - With Delta Lake, the Delta Table is immediately Ready for Use. The Transaction Log stored with the Delta Files contain all Metadata needed for an immediate query.

```
dfTblHealthTrackerProcessed = spark.read.table("tblHealthTrackerProcessed")  
dfTblHealthTrackerProcessed.count()
```

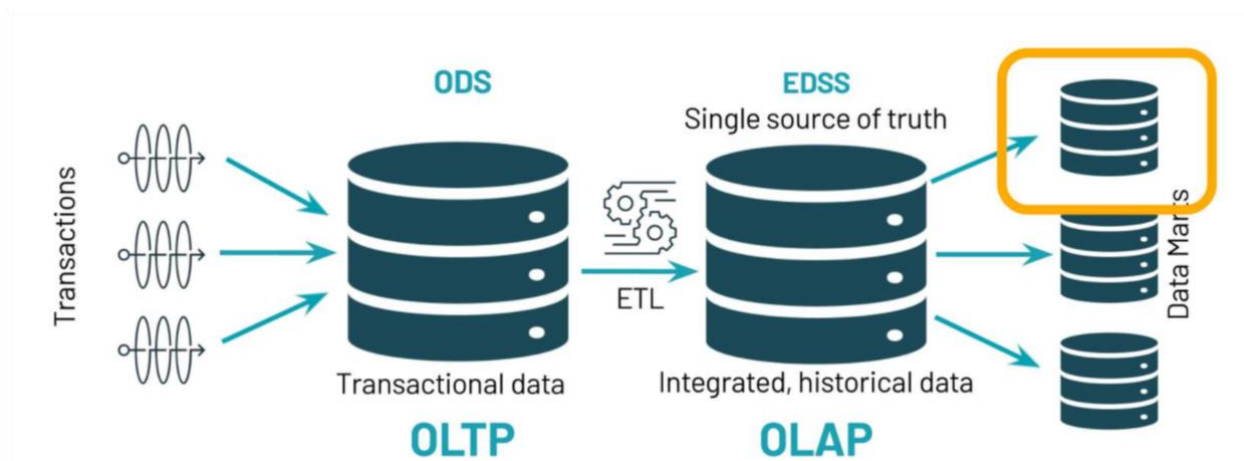
Output -

```
3720
```

# CREATE A NEW DELTA TABLE

## Create a New of Delta Table

- In this case, to create a new Delta Table, an Aggregate Table needs to be created from the Data of the previously created Delta Table. The Aggregate Table will be a Downstream Aggregate Table, or Data Mart.



- Remove Created “Delta Files” from the “analytics” Directory - Remove the previously created “Delta Files” from the “analytics” Directory, i.e., from the path - “`deltaDB/{userName}/rootFolder/subFolder/analytics`”. This step will make the Notebook “Idempotent”. In other words, it could be run more than once without throwing errors or introducing extra Files.

```
userName = 'oindrila'
```

```
deltaFileAggregationPath = f"deltaDB/{userName}/rootFolder/subFolder/analytics"  
dbutils.fs.rm(deltaFileAggregationPath, recurse = True)
```

- Create an Aggregate DataFrame - A Sub-Query is used to define the Aggregated Table that Aggregates the Delta Table “tblHealthTrackerProcessed” by the Column “device\_id” and Computes the Summary Statistics.

```

from pyspark.sql.functions import col, avg, max, stddev

dfTblHealthTrackerProcessed = spark.read.table("tblHealthTrackerProcessed")

dfHealthTrackerGoldUserAnalytics = dfTblHealthTrackerProcessed\
    .groupby("p_device_id")\
    .agg(
        avg(col("heartrate")).alias("avg_heartrate"),
        max(col("heartrate")).alias("max_heartrate"),
        stddev(col("heartrate")).alias("stddev_heartrate")
    )

```

- **Write the Content of the DataFrame to the “analytics” Directory as “Delta File”**  
 - Write the Content of the DataFrame to the “analytics” Directory, i.e., to the path - “**deltaDB/{userName}/rootFolder/subFolder/analytics**” as “Delta File”.

```

userName = 'oindrila'
aggregateDeltaTablePath = '/' + f"deltaDB/{userName}/rootFolder/subFolder/analytics"

dfHealthTrackerGoldUserAnalytics.write\
    .format("delta")\
    .mode("overwrite")\
    .save(aggregateDeltaTablePath)

```

- **Register the Delta File as Delta Table in the Metastore** - Spark SQL is used to create a Table on top of the saved “Delta File” and the created Table is Registered in the Metastore.  
 The “Format” of the Table to create is specified as “delta”, and the “Location” of the Table to create is referred to as the “Path of the saved Delta File”.

```

%sql
DROP TABLE IF EXISTS tblHealthTrackerGoldUserAnalytics;

CREATE TABLE tblHealthTrackerGoldUserAnalytics
USING DELTA
LOCATION "/deltaDB/oindrila/rootFolder/subFolder/analytics"

```

- **Count the Records of the Created Delta Table** - With Delta Lake, the Delta Table is immediately Ready for Use.

```

dfTblHealthTrackerGoldUserAnalytics = spark.read.table("tblHealthTrackerGoldUserAnalytics")
dfTblHealthTrackerGoldUserAnalytics.count()

```

Output -



## Important Note on Creation of New Delta Table

- ✚ When creating Delta Tables, the Delta Files in Object Storage defines the Schema, Partitioning, and Table Properties. For this reason, it is not necessary to specify any of these when Registering the Table with the Metastore.
- ✚ Furthermore, **NO TABLE REPAIR IS REQUIRED**. The Transaction Logs, stored with the Delta Files, contain all the Metadata needed for an immediate query.

# BATCH WRITE TO DELTA TABLE

## Different Ways of Modifying Existing Delta Table

- ✚ A Delta Table can be modified by either of the following methods -
  - Appending Files to an Existing Directory of Delta Files
  - Merging a Set of Updates and Insertions

Within the Context of Data Ingestion Pipeline, this is the addition of new Raw Files to the Single Source of Truth.

## Appending Files to an Existing Delta Table

- ✚ Load the Next Month of Data - To append the next month of Records, first, load the Data from the File “health\_tracker\_data\_2020\_2.json” as a PySpark DataFrame, from the “raw” Directory, using the “.format(‘json’)” Option, as before.

```
userName = 'oindrila'
sourceJsonFilePath = f"deltaDB/{userName}/rootFolder/subFolder/"
filePath = '/' + sourceJsonFilePath + "raw/health_tracker_data_2020_2.json"

dfHealthTrackerData2020_2 = spark.read\
    .format("json")\
    .load(filePath)
```

- ✚ Transform the Data - The same Data Engineering will be performed on the Sample Data for the Next Month with the following Transformations -
  - Use “from\_unixtime” Spark SQL Function to transform the UNIX Timestamp into a Time String.
  - Cast the “time” Column to Data Type “Timestamp” to replace the Column “time”.
  - Cast the “time” Column to Data Type “Date” to create the Column “dte”.
  - Select the Columns in the order in which the Columns are supposed to be written.

The Data Engineering is done using the previously defined User-Defined Function “processHealthTrackerData”.

```
dfProcessedData = processHealthTrackerData(dfHealthTrackerData2020_2)
```

- ✚ Append the Data for New Month to the “tblHealthTrackerProcessed” Delta Table - The Data for the Next Month is appended to the Delta Table “tblHealthTrackerProcessed” by using the “.mode(‘append’)” Option.

It is important to note that, it is not necessary to perform any Action on the Metastore.

```
deltaFileProcessingPath = f"/deltaDB/{userName}/rootFolder/subFolder/processed"

dfProcessedData.write\
    .mode("append")\
    .format("delta")\
    .save(deltaFileProcessingPath)
```

## View the Commit Using Time Travel

🚦 Delta Lake can Query an Earlier Version of a Delta Table using a feature, known as “Time Travel”.

- **View the Table as of Version 0** - This is done by specifying the Option “versionAsOf” as “0”. When it is Timed Travel to “Version 0”, only the Data of the First Month is displayed, i.e., Five Device Measurements, 24 Hours a Day, for 31 Days.

```
spark.read\
    .option("versionAsOf", 0)\
    .format("delta")\
    .load(deltaFileProcessingPath)\
    .count()
```

- **Count the Most Recent Version** - When a Delta Table is Queried without specifying a Version, it shows the Latest Version of the Table and includes the New Records added.

```
%sql
select count(*) from tblHealthTrackerProcessed
```

Output -

7128

At the Current Version, it is expected to see two months of Data, i.e., First Month is displayed, i.e., Five Device Measurements, 24 Hours a Day for (31 + 29) Days, i.e., 7200 Records (The Data was Recorded during the Month of February in a Leap Year, which is why there are 29 Days in a Month). But the Delta Table does not have Correct Count. 72 Records are missing.

## Late Arriving Data

- When the Batch Update was performed using Apache Spark on the Delta Table “tblHealthTrackerProcessed”, it was noticed that some Records were missing in the Delta Table.
- The absence of Records from the last few days of the month shows a phenomenon that may often occur in a Production Data Pipeline, i.e., “Late-Arriving Data”. Delta Lake allows to process Data as it arrives and is prepared to handle the Occurrence of “Late-Arriving Data”.
  - **Count the Number of Records Per Device** - The following Query is used to count the Number of Records Per Device.

```
from pyspark.sql.functions import count

display(
    spark.read\
        .format("delta")\
        .load(deltaFileProcessingPath)\
        .groupby("p_device_id")\
        .agg(count("*"))
)
```

Output -

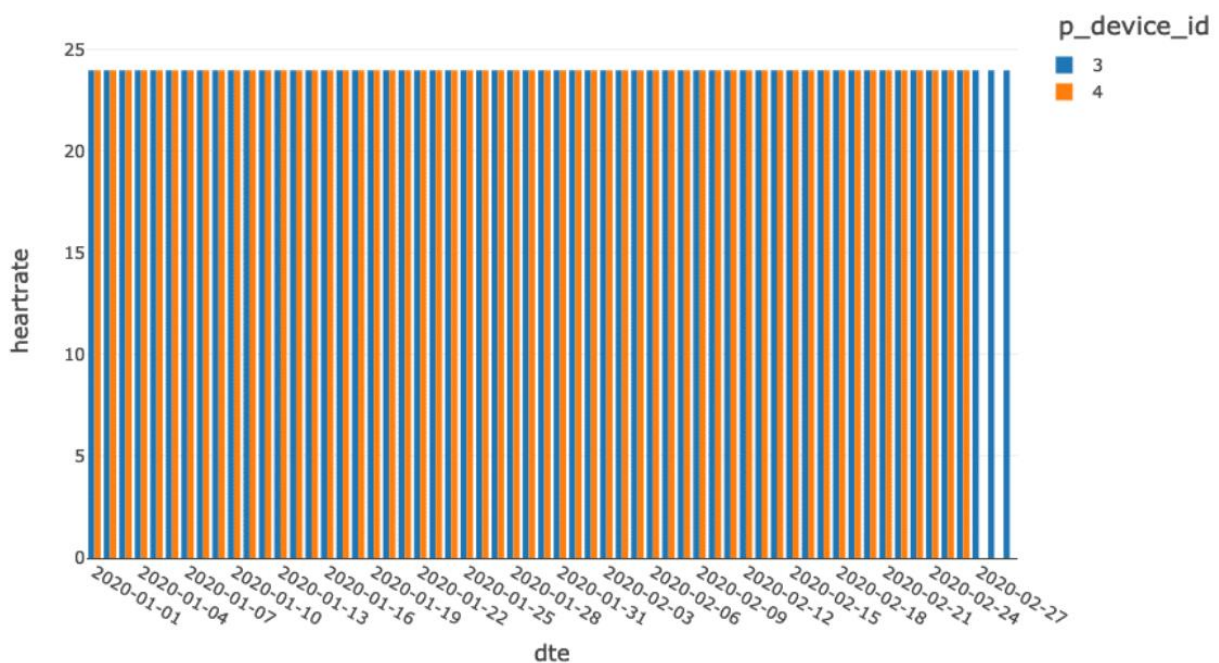
	device_id	count(1)
1	1	1440
2	3	1440
3	4	1368
4	2	1440
5	0	1440

It looks like “Device 4” is missing 72 Records.

- **Plot the Missing Records** - The following Query is used to discover the timing of the missing Records. A Databricks Visualization is used to display the Number of Records per day. It appears that there is no Records for “Device 4” for the last few days of the Month.

```
display(
  spark.read\
    .format("delta")\
    .load(deltaFilePath)
    .where(col("p_device_id").isin([3,4]))
)
```

Output -



## Broken Readings in the Delta Table

Upon the Initial Load of Data into the Delta Table “tblHealthTrackerProcessed”, it is noted that there are Broken Records in the Data. In particular, a note is made of the fact that several negative Readings were present even though it is impossible to record a negative heart rate.

- Create Temporary View for Broken Reading - First, a “Temporary View” is created for the Broken Readings in the Delta Table “tblHealthTrackerProcessed”.

```

from pyspark.sql.functions import to_timestamp, col

dfProcessedData = spark.sql("select * from tblHealthTrackerProcessed")

brokenReadings = dfProcessedData\
    .select(col("heartrate"), col("dte"))\
    .where(col("heartrate") < 0)\
    .groupby("dte")\
    .agg(count("heartrate"))\
    .orderBy("dte")

brokenReadings.createOrReplaceTempView("tempViewBrokenReadings")

```

- **Display the Broken Readings** - Display the Records in the “Temporary View”.

```

%sql

SELECT * FROM tempViewBrokenReadings

```

Output -

	dte ▲	count(heartrate) ▲
1	2020-01-01	1
2	2020-01-02	1
3	2020-01-04	1
4	2020-01-06	1
5	2020-01-07	1
6	2020-01-09	2
7	2020-01-12	3

Showing all 40 rows.

It is seen that most days have at least one Broken Reading and some days have more than one.

- **Sum the Broken Readings** - Next, perform a sum of the Records in the “Temporary View” of the Broken Readings.

```

%sql

SELECT SUM(`count(heartrate)`) FROM tempViewBrokenReadings

```



Output -

60

## Repair Records with an Upsert

- ✚ The word “UPSERT” is a “Portmanteau” of the words “UPDATE” and “INSERT” and this is what it does. An “UPSERT” will Update Records where some Criteria are met and otherwise will Insert the Record.

When Upserting into an existing Delta Table, the Spark SQL is used to perform the “MERGE” from another Registered Table or View. The Transaction Log records the Transaction, and the Metastore immediately reflects the changes.

The “MERGE” appends both -

- New/Inserted Files
- Files containing the Updates to the Delta File Directory

The Transaction Log tells the Delta Reader which File to use for each Record.

- ✚ **Prepare Updates DataFrame** - To Repair the Broken Sensor Readings, i.e., “Less Than Zero”, Interpolation needs to be used using the value, recorded before and after for each Device. The Spark SQL Functions “LAG” and “LEAD” will make this a trivial calculation. These values will be written to a Temporary View called “updatesView”. This View will be used later to “UPSERT” values into the Delta Table “tblHealthTrackerProcessed”.

- **Create a DataFrame Interpolating Broken Values** -

```
from pyspark.sql.window import Window
from pyspark.sql.functions import col, to_timestamp, lag, lead

dteWindow = Window.partitionBy("p_device_id").orderBy("dte")

dfInterpolated = spark.read\
    .table("tblHealthTrackerProcessed")\
    .select(\
        col("dte"),\
        col("time"),\
        col("heartrate"),\
        lag(col("heartrate")).over(dteWindow).alias("prev_amt"),\
        lead(col("heartrate")).over(dteWindow).alias("next_amt"),\
        col("name"),\
        col("p_device_id")\
    )
```

- **Create a DataFrame of Updates** -

```
dfUpdates = dfInterpolated\
    .where(col("heartrate") < 0)\
    .select(\
        col("dte"),\
        col("time"),\
        ((col("prev_amt") + col("next_amt"))/2).alias("heartrate"),\
        col("name"),\
        col("p_device_id")
    )
```

➤ View the Schemas of the “dfUpdates” and Delta Table “tblHealthTrackerProcessed” -

- Schema of DataFrame “dfUpdates” -

```
dfUpdates.printSchema()
```

Output -

```
root
|-- dte: date (nullable = true)
|-- time: timestamp (nullable = true)
|-- heartrate: double (nullable = true)
|-- name: string (nullable = true)
|-- p_device_id: integer (nullable = true)
```

- Schema of Delta Table “tblHealthTrackerProcessed” -

```
dfProcessedData = spark.sql("select * from tblHealthTrackerProcessed")
dfProcessedData.printSchema()
```

Output -

```
root
|-- dte: date (nullable = true)
|-- time: timestamp (nullable = true)
|-- heartrate: double (nullable = true)
|-- name: string (nullable = true)
|-- p_device_id: integer (nullable = true)
```

➤ Verify “dfUpdates” - Perform a “.count()” Operation on the DataFrame “dfUpdates”. It should have the same Number of Records as the SUM, performed on the Temporary View “tempViewBrokenReadings”.

```
dfUpdates.count()
```

Output -

60

➤ **Prepare Inserts DataFrame** - It turns out that the expectation of receiving the Missing Records Late was Correct. These Records have subsequently been made available in the File “health\_tracker\_data\_2020\_2\_late.json”.

- **Load the Late Arriving Data** - Load the Data from the Late-Arriving File “health\_tracker\_data\_2020\_2\_late.json” as a PySpark DataFrame, from the “raw” Directory, using the “.format(‘json’)” Option, as before.

```
userName = 'oindrila'
sourceJsonFilePath = f"deltaDB/{userName}/rootFolder/subFolder/"
filePath = '/' + sourceJsonFilePath + "raw/health_tracker_data_2020_2_late.json"

dfHealthTrackerData2020_2_late = spark.read\
    .format("json")\
    .load(filePath)
```

- **Transform the Data** - In addition to Updating the Broken Records, it is desirable to add the Late-Arriving Data. This can be done by preparing another Temporary View with the appropriate Transformations.

- Use “from\_unixtime” Spark SQL Function to transform the UNIX Timestamp into a Time String.
- Cast the “time” Column to Data Type “Timestamp” to replace the Column “time”.
- Cast the “time” Column to Data Type “Date” to create the Column “dte”.
- Select the Columns in the order in which the Columns are supposed to be written.

The Transformation is done using the previously defined User-Defined Function “processHealthTrackerData”.

```
dfInserts = processHealthTrackerData(dfHealthTrackerData2020_2_late)
```

- **View the Schema of the Inserts DataFrame “dfInserts”** -

```
dfInserts.printSchema()
```

Output -

```

root
|-- dte: date (nullable = true)
|-- time: timestamp (nullable = true)
|-- heartrate: double (nullable = true)
|-- name: string (nullable = true)
|-- p_device_id: integer (nullable = true)

```

#### 🚀 Prepare Upserts DataFrame -

- Create the “Union” DataFrame - Finally, the “dfUpserts” DataFrame is created that consists of all the Records in both the “dfUpdates” and the “dfInserts”. The DataFrame Function “.union()” is used to create the “dfUpserts” DataFrame.

```
dfUpserts = dfUpdates.union(dfInserts)
```

- View the Schema of the Upserts DataFrame -

```
dfUpserts.printSchema()
```

Output -

```

root
|-- dte: date (nullable = true)
|-- time: timestamp (nullable = true)
|-- heartrate: double (nullable = true)
|-- name: string (nullable = true)
|-- p_device_id: integer (nullable = true)

```

- Perform Upsert into the Delta Table “tblHealthTrackerProcessed” - Upsert Operation can be performed on a Delta Table using the “MERGE” Operation. This Operation is similar to the SQL MERGE Command, but, has added support for “Deletes”. The Conditions available with the “MERGE” Operation in Databricks are - “Updates”, “Inserts” and “Deletes”. The Delta Table Command “.merge()” provides full support for an Upsert Operation.

- Perform the Upsert -

```

from delta.tables import *

deltaTablePath = '/deltaDB/oindrila/rootFolder/subFolder/processed'
processedDeltaTable = DeltaTable.forPath(spark, deltaTablePath)

update_match = """
    tblHealthTrackerProcessed.time = upserts.time
    AND
    tblHealthTrackerProcessed.p_device_id = upserts.p_device_id
"""

update = { "heartrate" : "upserts.heartrate" }

insert = {
    "p_device_id" : "upserts.p_device_id",
    "heartrate" : "upserts.heartrate",
    "name" : "upserts.name",
    "time" : "upserts.time",
    "dte" : "upserts.dte"
}

processedDeltaTable.alias("tblHealthTrackerProcessed")\
    .merge(dfUpserts.alias("upserts"), update_match)\
    .whenMatchedUpdate(set = update)\
    .whenNotMatchedInsert(values = insert)\
    .execute()

```

#### 🚦 View the Commit Using Time Travel -

- View the Table as of Version 1 - This is done by specifying the Option "versionAsOf" as "1".
  - When Time Travel is performed to Version 0, only the first month of Data is displayed.
  - When Time Travel is performed to Version 1, the first two months of Data minus the missing 72 Records are displayed.

```

spark.read\
    .option("versionAsOf", 1)\
    .format("delta")\
    .load(deltaTablePath)\
    .count()

```

➤ Count the Most Recent Version -

```
dfProcessedData = spark.sql("select * from tblHealthTrackerProcessed")
dfProcessedData.count()
```

Output -

7200

➤ Describe the History of the Delta Table “tblHealthTrackerProcessed” - The Delta Table Command “.history()” provides provenance information, including the Operation, User and so on, for each Action performed on a Delta Table.

```
from delta.tables import *

processedDeltaTable = DeltaTable.forPath(spark, deltaTablePath)
display(processedDeltaTable.history())
```

Output -

	version	timestamp	userId	userName	operation	operationParameters
1	2	2021-09-20T08:18:43.000+0000	5941607280537835	oindrila.chakraborty2988@gmail.com	MERGE	{“predicate”: “((tblHealthTrackerProcessed.`upserts.`p_device_id`))”, “matchedPredicates” [({“actionType”: “insert”})]}
2	1	2021-09-20T07:43:29.000+0000	5941607280537835	oindrila.chakraborty2988@gmail.com	WRITE	{“mode”: “Append”, “partitionBy”: “[ ]”}
3	0	2021-09-20T07:41:32.000+0000	5941607280537835	oindrila.chakraborty2988@gmail.com	CONVERT	{“numFiles”: “5”, “partitionedBy”: “[\`p_device`”, “deltadb_oindrila`.`tblhealthtrackerprocessed`”}

It is to be noted that, each Operation performed on a Delta Table is given a Version Number. These are the Numbers being used when Time Travel Query is performed on the Delta Table. Example - “SELECT COUNT(\*) FROM tblHealthTrackerProcessed” where VERSION as of 1”.

## Perform a Second Upsert

🔗 An Upsert Operation on the Delta Table “tblHealthTrackerProcessed” was performed simultaneously with -

- Updating Records containing Broken Readings.
- Inserting the Late-Arriving Data.

In doing so, more Broken Readings got added.

🔗 Sum the Broken Readings - Perform a sum of the Records in the “Temporary View” of the Broken Readings once more.

```
%sql

SELECT SUM(`count(heartrate)`) FROM tempViewBrokenReadings
```



Output -

1

It is to be noted that, there are still Broken Readings in the Delta Table. This is because, many of the Records, inserted as part of the Upsert Operation, also contained Broken Readings.

- ✚ **Verify that These are New Broken Readings** - Query the “Temporary View” of the Broken Readings with a WHERE clause to verify that these are indeed new Broken Readings, introduced by inserting the Late-Arriving Data. It is to be noted that, there are no Broken Readings before “2020-02-25”.

```
%sql
```

```
SELECT SUM(`count(heartrate)`) FROM tempViewBrokenReadings WHERE dte < '2020-02-25'
```

Output -

null

- ✚ **Verify Updates** - Perform a “.count()” on the Updates DataFrame “dfUpdates”. It should have the same Number of Records as the SUM performed on the Temporary View “tempViewBrokenReadings”.

```
dfUpdates.count()
```

Output -

1

- ✚ **Perform Upsert into the Delta Table “tblHealthTrackerProcessed”** - Once again, the Upsert Operation is performed on the Delta Table “tblHealthTrackerProcessed” using the Delta Table Command “.merge()”.
- ✚ **Sum the Broken Readings** - Perform the sum of the Records in the “Temporary View” of the Broken Readings one last time. Finally, there are no Broken Readings inside the Delta Table “tblHealthTrackerProcessed”.

```
%sql
```

```
SELECT SUM(`count(heartrate)`) FROM tempViewBrokenReadings
```

Output -

null

# EVOLUTION OF DATA BEING INGESTED

## Evolution of Data Being Ingested into Existing Delta Table

- ✚ It is not uncommon that the Schema of the Data being Ingested into the EDSS will Evolve over time. In this case, the Health Tracker Device has a new Version available, and the Data being Transmitted now contains an additional field that indicates which type of Device is being used.

Each Line, inside each of the Sample File, is a String, representing a Valid JSON Object and is similar to the kind of String that would be passed by a Kafka Stream Processing Server. Following is a Sample of the Data to be used -

```
{"device_id":0,"heartrate":57.6447293596,"name":"Deborah Powell","time":1.5830208E9,"device_type":"version 2"}
{"device_id":0,"heartrate":57.6175546013,"name":"Deborah Powell","time":1.5830244E9,"device_type":"version 2"}
{"device_id":0,"heartrate":57.8486376876,"name":"Deborah Powell","time":1.583028E9,"device_type":"version 2"}
{"device_id":0,"heartrate":57.8821378637,"name":"Deborah Powell","time":1.5830316E9,"device_type":"version 2"}
{"device_id":0,"heartrate":59.0531490807,"name":"Deborah Powell","time":1.5830352E9,"device_type":"version 2"}
```

The Data has the following Schema -

```
name: string
heartrate: double
device_id: long
time: long
device_type: string
```

## Appending Files to an Existing Delta Table Without Schema Evolution

- ✚ Load the Next Month of Data - To append the next month of Records with the new Schema, first, load the Data from the File “health\_tracker\_data\_2020\_3.json” as a PySpark DataFrame, from the “raw” Directory, using the “.format(‘json’)” Option, as before.

```
userName = 'oindrila'
sourceJsonFilePath = f"deltaDB/{userName}/rootFolder/subFolder/"
filePath = '/' + sourceJsonFilePath + "raw/health_tracker_data_2020_3.json"

dfHealthTrackerData2020_3 = spark.read\
    .format("json")\
    .load(filePath)
```

- ✚ Transform the Data - The same Data Engineering will be performed on the Sample Data for the Next Month with the following Transformations -
  - Use “from\_unixtime” Spark SQL Function to transform the UNIX Timestamp into a Time String.

- Cast the “time” Column to Data Type “Timestamp” to replace the Column “time”.
- Cast the “time” Column to Data Type “Date” to create the Column “dte”.
- Select the Columns in the order in which the Columns are supposed to be written.

Now, the User-Defined Function “processHealthTrackerData” needs to be Re-Defined to accommodate the new Schema.

```
from pyspark.sql.functions import col, from_unixtime, to_timestamp

def processHealthTrackerData(df):
    df = df\
        .withColumn("time", from_unixtime("time"))\
        .withColumnRenamed("device_id", "p_device_id")\
        .withColumn("time", col("time").cast("timestamp"))\
        .withColumn("dte", to_timestamp(col("time")).cast("date"))\
        .withColumn("p_device_id", col("p_device_id").cast("integer"))\
        .select("dte", "time", "device_type", "heartrate", "name", "p_device_id")

    return df
```

The Data Engineering is done using the Re-Defined User-Defined Function “processHealthTrackerData”.

```
dfProcessedData = processHealthTrackerData(dfHealthTrackerData2020_3)
```

- ✚ **Append the Data for New Month to the “tblHealthTrackerProcessed” Delta Table** - The Data for the Next Month is appended to the Delta Table “tblHealthTrackerProcessed” by using the “.mode(‘append’)” Option.

It is important to note that, it is not necessary to perform any Action on the Metastore.

```
deltaFileProcessingPath = f"/deltaDB/{userName}/rootFolder/subFolder/processed"

dfProcessedData.write\
    .mode("append")\
    .format("delta")\
    .save(deltaFileProcessingPath)
```

When the above Code is run, an Error is received, because, there is a Mismatch between the Table Schema and the Schema of the Data to be Ingested.

```

⚠️AnalysisException: A schema mismatch detected when writing to the Delta table (Table ID: 29bf0ce2-b7f8-4ecc-8571-9f61564e68a7).
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
-- dtc: date (nullable = true)
-- time: timestamp (nullable = true)
-- heartrate: double (nullable = true)
-- name: string (nullable = true)
-- p_device_id: integer (nullable = true)

Data schema:
root
-- dtc: date (nullable = true)
-- time: timestamp (nullable = true)
-- device_type: string (nullable = true)
-- heartrate: double (nullable = true)
-- name: string (nullable = true)
-- p_device_id: integer (nullable = true)

```

## What is Schema Enforcement

- ✚ Schema Enforcement, also known as Schema Validation, is a Safeguard in Delta Lake that ensures Data Quality by rejecting Writes to a Delta Table that do not match the Delta Table's Schema.
- ✚ Like the Front Desk Manager at a busy Restaurant that only accepts Reservations, the Schema Enforcement checks to see whether each Column in the Data, to be inserted into the Delta Table, is on its List of Expected Column (in other words, whether each one has a "Reservation"), and rejects any Writes with Columns that are not on the List, or, with Data Type Mismatches.

## Appending Files to an Existing Delta Table with Schema Evolution

- ✚ In this case, it is expected that the Delta Table would accept the New Schema and would add the Data to the concerned Delta Table.
- ✚ **Schema Evolution** - Schema Evolution is a Feature that allows Users to easily Change a Delta Table's Current Schema to accommodate Data that is Changing over time. Most commonly, it is used when performing an Append, or Overwrite Operation., to automatically Adapt the Schema to include one or more New Columns
  - **Append the Data for New Month with Schema Evolution to the "tblHealthTrackerProcessed" Delta Table** - The Data for the Next Month is appended to the Delta Table "tblHealthTrackerProcessed" by using the

.mode('append')” Option. In this case, the Option “mergeSchema” also needs to be set to “True”.

```
deltaFileProcessingPath = f"/deltaDB/{userName}/rootFolder/subFolder/processed"

dfProcessedData.write\
    .mode("append")\
    .option("mergeSchema", True)\
    .format("delta")\
    .save(deltaFileProcessingPath)
```

➤ Verify the Commit -

- Count the Most Recent Version - In the Current Version, it is expected to see Three Months of Data, i.e., Five Device Measurements, 24 Hours a Day for (31 + 29 + 31) Days, or 10920 Records.

```
dfProcessedData = spark.sql("select * from tblHealthTrackerProcessed")
dfProcessedData.count()
```

Output -

```
10920
```

# DELETE DATA AND RECOVER LOST DATA

## Delete Data from Existing Delta Table

- ✚ The “DELETE” Spark SQL Command is used to remove All Records from a Delta Table that match the given Predicate. Example - delete all the Records, associated with Device Id “4”, from the Delta Table “tblHealthTrackerProcessed”.

```
from delta.tables import *

deltaTablePath = '/deltaDB/oindrila/rootFolder/subFolder/processed'
processedDeltaTable = DeltaTable.forPath(spark, deltaTablePath)
processedDeltaTable.delete("p_device_id = 4")
```

## Recover Lost Data of Existing Delta Table

- ✚ It is possible to Recover Lost Data by using the Time Travel Capability of Delta Lake.
- **Prepare New Upserts DataFrame** - A New DataFrame needs to be prepared for Upserting using Time Travel to Recover the Missing Data. It is to be noted that, the entire “name” Column is replaced with the NULL value.

```
from pyspark.sql.functions import lit

deltaTablePath = '/deltaDB/oindrila/rootFolder/subFolder/processed'
dfUpserts = spark.read\
    .option("versionAsOf", 4)\
    .format("delta")\
    .load(deltaTablePath)\
    .where("p_device_id = 4")\
    .select("dte", "time", "device_type", "heartrate", lit(None).alias("name"), "p_device_id")
```

- **Perform Upsert into the Delta Table “tblHealthTrackerProcessed”** - Once more, the Upsert Operation needs to be performed on the Delta Table “tblHealthTrackerProcessed” using the Delta Command “.merge()”.

It is necessary to define -

- The Reference to the Delta Table
- The Insert Logic because the Schema has Changed

```

deltaTablePath = '/deltaDB/oindrila/rootFolder/subFolder/processed'
processedDeltaTable = DeltaTable.forPath(spark, deltaTablePath)

update_match = """
    tblHealthTrackerProcessed.time = upserts.time
    AND
    tblHealthTrackerProcessed.p_device_id = upserts.p_device_id
"""

update = { "heartrate" : "upserts.heartrate" }

insert = {
    "dte" : "upserts.dte",
    "time" : "upserts.time",
    "device_type" : "upserts.device_type",
    "heartrate" : "upserts.heartrate",
    "name" : "upserts.name",
    "p_device_id" : "upserts.p_device_id"
}

processedDeltaTable.alias("tblHealthTrackerProcessed")\
    .merge(dfUpserts.alias("upserts"), update_match)\
    .whenMatchedUpdate(set = update)\
    .whenNotMatchedInsert(values = insert)\
    .execute()

```

- **Count the Most Recent Version** - In the Current Version, it is expected to see Three Months of Data, i.e., Five Device Measurements, 24 Hours a Day for (31 + 29 + 31) Days, or 10920 Records.

```

dfProcessedData = spark.sql("select * from tblHealthTrackerProcessed")
dfProcessedData.count()

```

Output -

```
10920
```

- **Query "Device 4" to Demonstrate Compliance** - The Delta Table "tblHealthTrackerProcessed" is Queried to demonstrate that the Records, associated with Device Id "4", have been indeed Removed.

```
display(dfProcessedData.where("p_device_id = 4"))
```

Output -

	dte ▲	time ▲	heartrate ▲	name ▲	p_device_id ▲	device_type ▲
1	2020-03-01	2020-03-01T00:00:00.000+0000	97.8678768636	null	4	version 2
2	2020-03-01	2020-03-01T01:00:00.000+0000	97.586595396	null	4	version 2
3	2020-03-01	2020-03-01T02:00:00.000+0000	97.188151848	null	4	version 2
4	2020-03-01	2020-03-01T03:00:00.000+0000	97.4361573672	null	4	version 2
5	2020-03-01	2020-03-01T04:00:00.000+0000	95.8997954454	null	4	version 2
6	2020-03-01	2020-03-01T05:00:00.000+0000	96.5277339825	null	4	version 2
7	2020-03-01	2020-03-01T06:00:00.000+0000	98.1774838993	null	4	version 2

Truncated results, showing first 1000 rows.



# MAINTAIN COMPLIANCE WITH VACUUM OPERATION

## Maintain Compliance with a Vacuum Operation

Due to the power of the Delta Lake Time Travel Feature, the Delta Table is not yet in Compliance, as the Delta Table could simply be Queried against an Earlier Version to identify the Records, associated with Device Id “4”.

- **Query an Earlier Table Version** - The Delta Table “tblHealthTrackerProcessed” is Queried against an Earlier Version to demonstrate that it is still possible to Retrieve all the Records, associated with Device Id “4”.

```
deltaTablePath = '/deltaDB/oindrila/rootFolder/subFolder/processed'

display(\
  spark.read\
    .option("versionAsOf", 4)\
    .format("delta")\
    .load(deltaTablePath)\
    .where("p_device_id = 4")\
)
```

Output -

	dte	time	heartrate	name	p_device_id	device_type
1	2020-01-01	2020-01-01T00:00:00.000+0000	60.7236962271	James Hou	4	null
2	2020-01-01	2020-01-01T01:00:00.000+0000	59.7518357438	James Hou	4	null
3	2020-01-01	2020-01-01T02:00:00.000+0000	59.7552762926	James Hou	4	null
4	2020-01-01	2020-01-01T03:00:00.000+0000	61.8018342845	James Hou	4	null
5	2020-01-01	2020-01-01T04:00:00.000+0000	60.3112488045	James Hou	4	null
6	2020-01-01	2020-01-01T05:00:00.000+0000	60.0099058887	James Hou	4	null
7	2020-01-01	2020-01-01T06:00:00.000+0000	59.8323375338	James Hou	4	null

Truncated results, showing first 1000 rows.

- **Vacuum Tables to Remove Old Files** - The Spark SQL Command “VACUUM” can be used to solve this problem. The “VACUUM” Command “Recursively Vacuums Directories associated with the Delta Table and Removes Files that are No Longer in the Latest State of the Transaction Log for that Delta Table and that are Older than a Retention Threshold”. The “Default Threshold” is “7 Days”.

```
deltaTablePath = '/deltaDB/oindrila/rootFolder/subFolder/processed'
processedDeltaTable = DeltaTable.forPath(spark, deltaTablePath)
processedDeltaTable.vacuum(0)
```

When the above Code is run, an Error is received due to the Default Threshold. The Default Threshold is in place to prevent Corruption of the Delta Table.

```
IllegalArgumentException: requirement failed: Are you sure you would like to vacuum files with such a low retention period? If you have writers that are currently writing to this table, there is a risk that you may corrupt the state of your Delta table.
```

If you are certain that there are no operations being performed on this table, such as insert/upsert/delete/optimize, then you may turn off this check by setting:  
spark.databricks.delta.retentionDurationCheck.enabled = false

If you are not sure, please use a value not less than "168 hours".

- **Set Delta to Allow the Vacuum Operation** - To demonstrate the “VACUUM” Command, the Retention Period is set to “0 Hours” to be able to Remove the Questionable Files now.

This is typically Not a Best Practice and in fact, there are Safeguards in place to prevent this Operation from being performed. For demonstration purposes, Delta is set to Allow this Operation.

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

- **Vacuum Tables to Remove Old Files** - The Spark SQL Command “VACCUM” can be used to remove the Files that are No Longer in the Latest State of the Transaction Log for the Delta Table “tblHealthTrackerProcessed”.



```
deltaTablePath = '/deltaDB/oindrila/rootFolder/subFolder/processed'  
processedDeltaTable = DeltaTable.forPath(spark, deltaTablePath)  
processedDeltaTable.vacuum(0)
```

- **Attempt to Query an Earlier Table Version** - Now, when it is attempted to perform a Query on an Earlier Version of the Delta Table “tblHealthTrackerProcessed”, an Error is thrown.

```
deltaTablePath = '/deltaDB/oindrila/rootFolder/subFolder/processed'  
  
display(\  
  spark.read\  
    .option("versionAsOf", 4)\  
    .format("delta")\  
    .load(deltaTablePath)\  
    .where("p_device_id = 4")\  
)
```

Output -

```
FileNotFoundException: Error while reading file dbfs:/deltaDB/oindrila/rootFolder/subFolder/processed/p_device_id=4/part-00000-7003d874-fd18-49ed-b775-ede5cc00fe66.c000.snappy.parquet. A file referenced in the transaction log cannot be found. This occurs when data has been manually deleted from the file system rather than using the table 'DELETE' statement. For more information, see https://docs.databricks.com/delta/delta-intro.html#frequently-asked-questions  
Caused by: FileNotFoundException: /deltaDB/oindrila/rootFolder/subFolder/processed/p_device_id=4/part-00000-7003d874-fd18-49ed-b775-ede5cc00fe66.c000.snappy.parquet
```



This Error indicates that it is not possible to Query Data from the Earlier Version because the Files have been Expunged from the System.