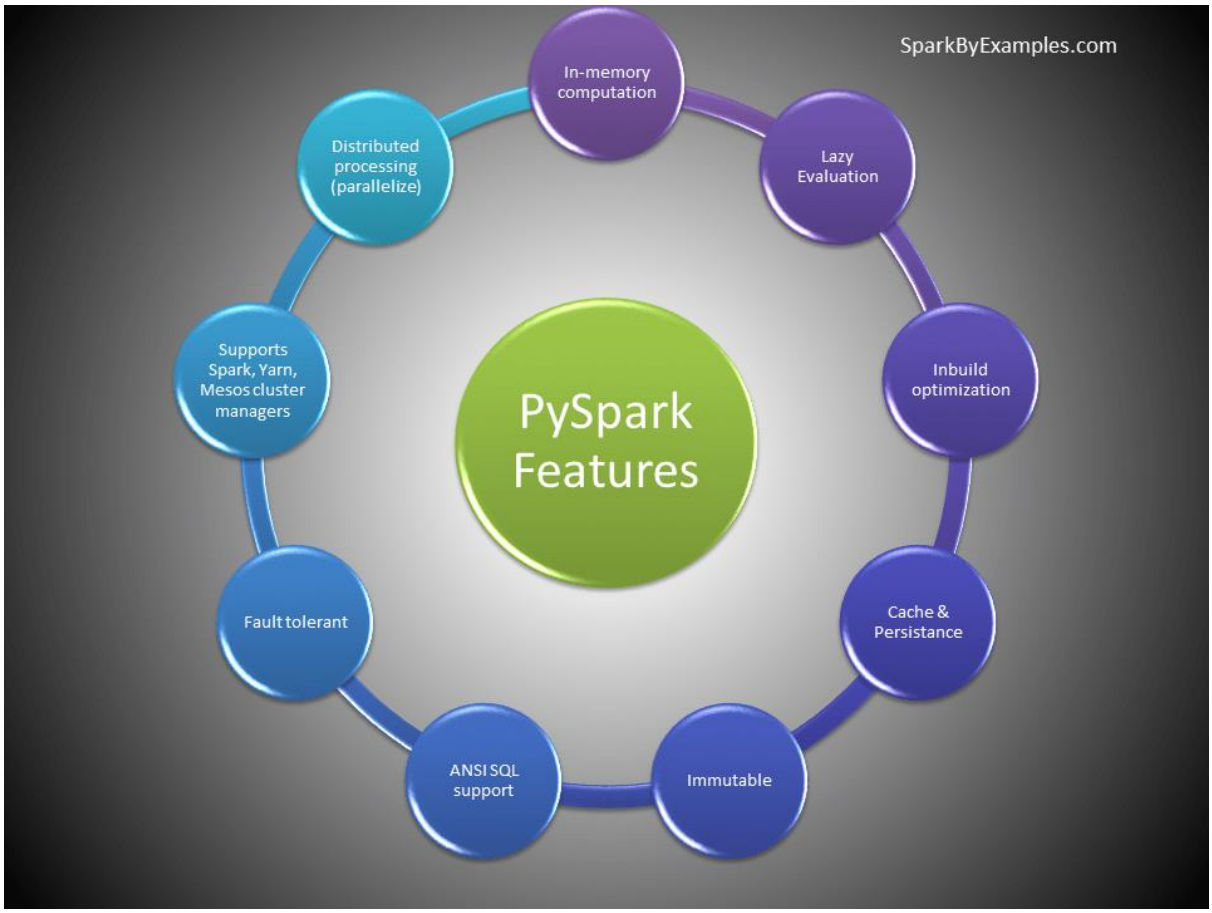
1. What is PySpark?

PySpark is a Spark library written in Python to run Python application using Apache Spark capabilities, using PySpark we can run applications parallelly on the distributed cluster (multiple nodes).

PySpark is a Python API for Apache Spark. Apache Spark is an analytical processing engine for large scale powerful distributed data processing and machine learning applications.

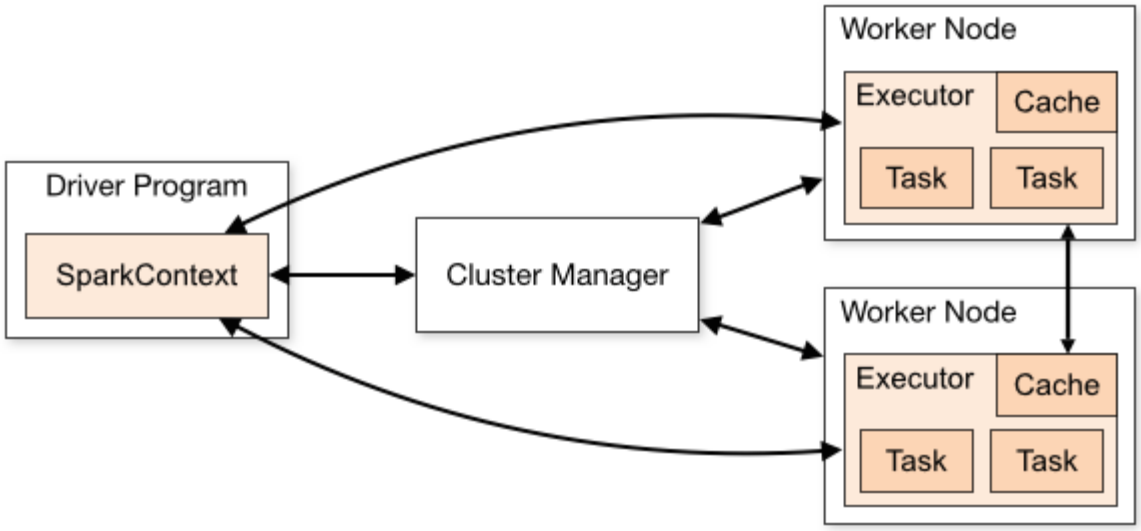
Spark basically written in Scala and later on due to its industry adaptation it's API PySpark released for Python using Py4J. Py4J is a Java library that is integrated within PySpark and allows python to dynamically interface with JVM objects



1.1. Advantages of PySpark

- PySpark is a general-purpose, in-memory, distributed processing engine that allows you to process data efficiently in a distributed fashion.
- Applications running on PySpark are 100x faster than traditional systems.
- You will get great benefits using PySpark for data ingestion pipelines.
- Using PySpark we can process data from Hadoop HDFS, AWS S3, and many file systems.
- PySpark also is used to process real-time data using Streaming and Kafka.
- Using PySpark streaming you can also stream files from the file system and also stream from the socket.
- PySpark natively has machine learning and graph libraries.

1.2. PySpark Architecture



Apache Spark works in a master-slave architecture where the master is called “Driver” and slaves are called “Workers”. When you run a Spark application, Spark Driver creates a context that is an entry point to your application, and all operations (transformations and actions) are executed on worker nodes, and the resources are managed by Cluster Manager.

1.3. Cluster Manager Types

As of writing this Spark with Python (PySpark) tutorial, Spark supports below cluster managers:

- Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos – Mesons is a Cluster manager that can also run Hadoop MapReduce and PySpark applications.
- Hadoop YARN – the resource manager in Hadoop 2. This is mostly used, cluster manager.
- Kubernetes – an open-source system for automating deployment, scaling, and management of containerized applications.

local – which is not really a cluster manager but still I wanted to mention as we use “local” for `master()` in order to run Spark on your laptop/computer

1.4. PySpark Modules & Packages

- PySpark RDD (pyspark.RDD)
- PySpark DataFrame and SQL (pyspark.sql)
- PySpark Streaming (pyspark.streaming)
- PySpark MLib (pyspark.ml, pyspark.mllib)
- PySpark GraphFrames (GraphFrames)
- PySpark Resource (pyspark.resource) It’s new in PySpark 3.0

1.5. Spark Web UI

Apache Spark provides a suite of Web UIs (Jobs, Stages, Tasks, Storage, Environment, Executors, and SQL) to monitor the status of your Spark application, resource consumption of Spark cluster, and Spark configurations. On Spark Web UI, you can see how the operations are executed.

1.6. Spark History Server

Spark History servers, keep a log of all Spark applications you submit by spark-submit, spark-shell. before you start, first you need to set the below config on spark-defaults.conf

spark.eventLog.enabled true
spark.history.fs.logDirectory file:///c:/logs/path

1.7. RDD Creation

SparkSession can be created using a builder() or newSession() methods of the SparkSession. Spark session internally creates a sparkContext variable of SparkContext. You can create multiple SparkSession objects but only one SparkContext per JVM. In case if you want to create another new SparkContext you should stop existing Sparkcontext (using stop()) before creating a new one.

```
# Import SparkSession
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder \
    .master("local[1]") \
    .appName("SparkByExamples.com") \
    .getOrCreate()
```

1.7.1. using parallelize()

SparkContext has several functions to use with RDDs. For example, it's parallelize() method is used to create an RDD from a list.

```
# Create RDD from parallelize
dataList = [("Java", 20000), ("Python", 100000), ("Scala", 3000)]
rdd=spark.sparkContext.parallelize(dataList)
```

1.7.2. using textFile()

RDD can also be created from a text file using textFile() function of the SparkContext.

```
# Create RDD from external Data source
rdd2 = spark.sparkContext.textFile("/path/test.txt")
```

1.7.3. RDD Operations

On PySpark RDD, you can perform two kinds of operations.

RDD transformations – Transformations are lazy operations. When you run a transformation(for example update), instead of updating a current RDD, these operations return another RDD.

RDD actions – operations that trigger computation and return RDD values to the driver.

1.7.4. RDD Transformations

Transformations on Spark RDD returns another RDD and transformations are lazy meaning they don't execute until you call an action on RDD. Some transformations on RDD's are flatMap(), map(), reduceByKey(), filter(), sortByKey() and return new RDD instead of updating the current.

1.7.5. RDD Actions

RDD Action operation returns the values from an RDD to a driver node. In other words, any RDD function that returns non RDD[T] is considered as an action.

Some actions on RDDs are `count()`, `collect()`, `first()`, `max()`, `reduce()` and more.

1.8. PySpark DataFrame

DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python. DataFrames can be constructed from a wide array of sources such as structured data files, tables in Hive, external databases, or existing RDDs.

PySpark DataFrame is mostly similar to Pandas DataFrame with the exception PySpark DataFrames are distributed in the cluster (meaning the data in DataFrame's are stored in different machines in a cluster) and any operations in PySpark executes in parallel on all machines whereas Panda Dataframe stores and operates on a single machine.

1.8.1. DataFrame creation

The simplest way to create a DataFrame is from a Python list of data. DataFrame can also be created from an RDD and by reading files from several sources.

```
data = [('James','','Smith','1991-04-01','M',3000),
  ('Michael','Rose','','2000-05-19','M',4000),
  ('Robert','','Williams','1978-09-05','M',4000),
  ('Maria','Anne','Jones','1967-12-01','F',4000),
  ('Jen','Mary','Brown','1980-02-17','F',-1)
]

columns = ["firstname","middlename","lastname","dob","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
df.printSchema()
```

`df.show()` shows the 20 elements from the DataFrame.

1.8.2. DataFrame operations

Like RDD, DataFrame also has operations like Transformations and Actions.

1.8.3. DataFrame from external data sources

In real-time applications, DataFrames are created from external sources like files from the local system, HDFS, S3 Azure, HBase, MySQL table e.t.c. Below is an example of how to read a CSV file from a local system.

```
df = spark.read.csv("/tmp/resources/zipcodes.csv")
df.printSchema()
```

1.8.4. Supported file formats

DataFrame has a rich set of API which supports reading and writing several file formats

- csv
- text
- Avro
- Parquet

- tsv
- xml and many more

1.9. PySpark SQL Tutorial

PySpark SQL is one of the most used Py**Spark** modules which is used for processing structured columnar data format. Once you have a DataFrame created, you can interact with the data by using SQL syntax.

In other words, Spark SQL brings native RAW SQL queries on Spark meaning you can run traditional ANSI SQL's on Spark Dataframe

In order to use SQL, first, create a temporary table on DataFrame using createOrReplaceTempView() function. Once created, this table can be accessed throughout the SparkSession using sql() and it will be dropped along with your SparkContext termination.

```
df.createOrReplaceTempView("PERSON_DATA")
df2 = spark.sql("SELECT * from PERSON_DATA")
df2.printSchema()
df2.show()
```

1.10. PySpark Streaming Tutorial

PySpark Streaming is a scalable, high-throughput, fault-tolerant streaming processing system that supports both batch and streaming workloads. It is used to process real-time data from sources like file system folder, TCP socket, S3, Kafka, Flume, Twitter, and Amazon Kinesis to name a few. The processed data can be pushed to databases, Kafka, live dashboards e.t.c

2. Spark Web UI – Understanding Spark Execution

Apache Spark provides a suite of Web UI/User Interfaces (Jobs, Stages, Tasks, Storage, Environment, Executors, and SQL) to monitor the status of your Spark/PySpark application, resource consumption of Spark cluster, and Spark configurations.

Your application code is the set of instructions that instructs the driver to do a Spark Job and let the driver decide how to achieve it with the help of executors.

Instructions to the driver are called Transformations and action will trigger the execution.

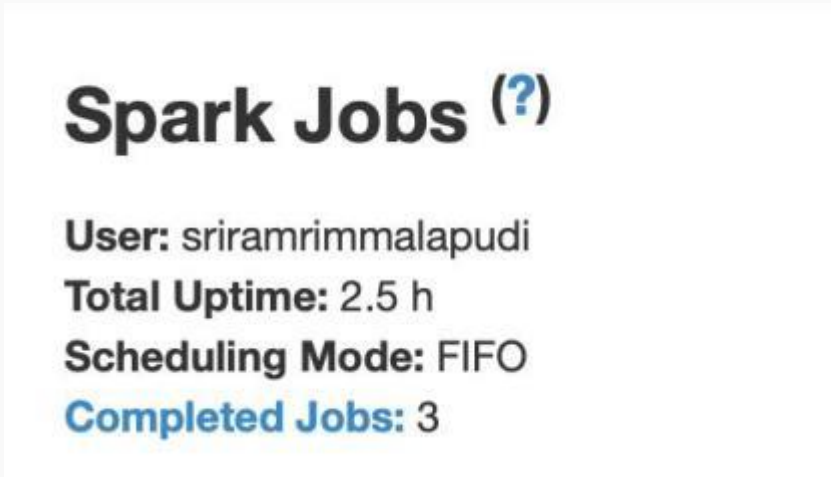
```
//Transformation
val rawDF = spark.read //job(0) : Read
              .option("inferSchema", "true") //job(1) : InferSchema
              .option("header", "true")
              .csv( path = "data/survey.csv")
//Action
rawDF.count()// job(2): Get Count
```

Spark UI is separated into below tabs.

- Spark Jobs
- Stages
- Tasks
- Storage
- Environment
- Executors
- SQL

2.1. Spark Jobs Tab

The details that I want you to be aware of under the jobs section are **Scheduling mode**, the **number of Spark Jobs**, the **number of stages** it has, and **Description** in your spark job.



2.1.1. Scheduling Mode

We have three Scheduling modes.

- Standalone mode
- YARN mode
- Mesos

2.1.2. Number of Spark Jobs:

Always keep in mind, the number of Spark jobs is equal to the number of actions in the application and each Spark job should have at least one Stage.
In our above application, we have performed 3 Spark jobs (0,1,2)

- Job 0. read the CSV file.
- Job 1. Inferschema from the file.
- Job 2. Count Check

2.1.3. Number of Stages

Each Wide Transformation results in a separate Number of Stages. In our case, Spark job0 and Spark job1 have individual single stages but when it comes to Spark job 3 we can see two stages that are because of the partition of data. Data is partitioned into two files by default.

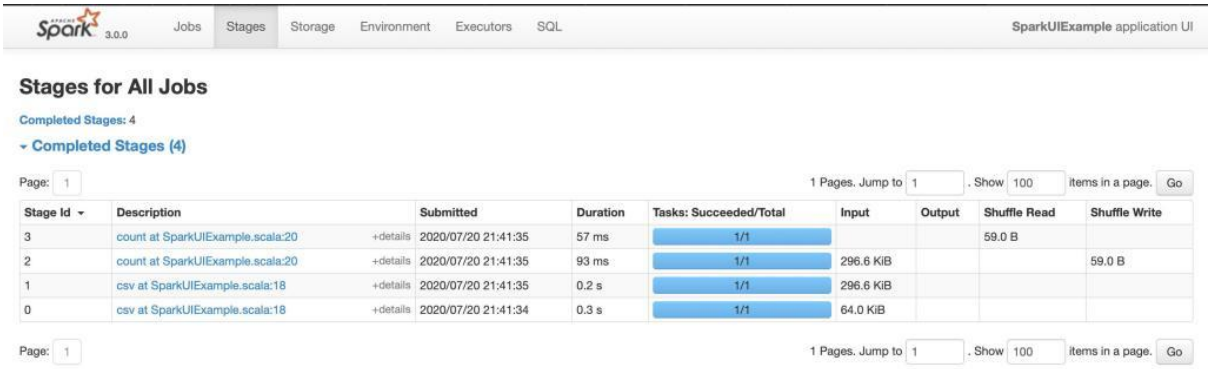
2.1.4. Description

Description links the complete details of the associated SparkJob like Spark Job Status, DAG Visualization, Completed Stages

2.2. Stages Tab

We can navigate into Stage Tab in two ways.

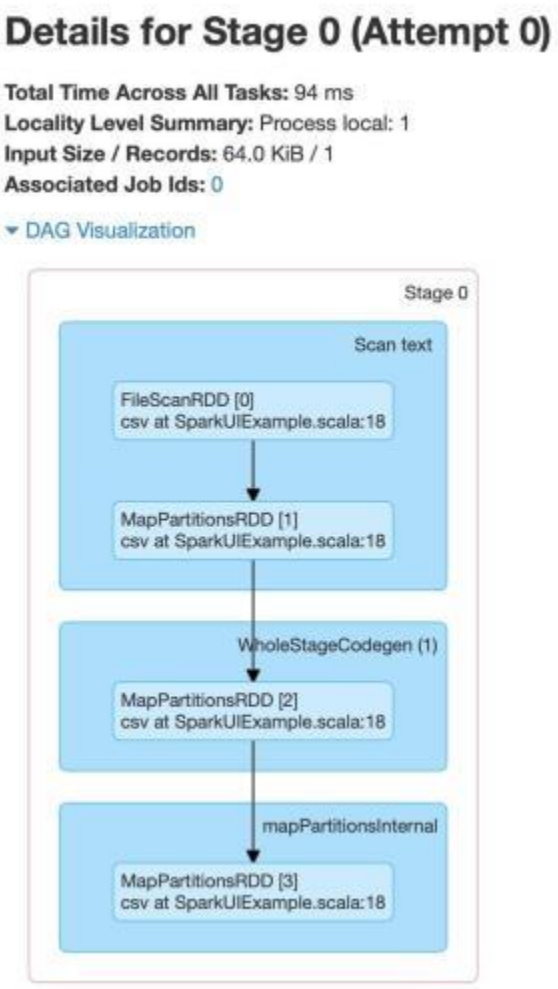
- Select the Description of the respective Spark job (Shows stages only for the Spark job opted)
- On the top of Spark Job tab select Stages option (Shows all stages in Application)



In our application, we have a total of **4 *Stages***.

The Stage tab displays a summary page that shows the current state of all stages of all Spark jobs in the spark application

The number of tasks you could see in each stage is the number of partitions that spark is going to work on and each task inside a stage is the same work that will be done by spark but on a different partition of data.



2.2.1. Stage detail

Details of stage showcase Directed Acyclic Graph (DAG) of this stage, where vertices represent the RDDs or DataFrame and edges represent an operation to be applied.

let us analyze operations in Stages
Operations in Stage0 are
1.FileScanRDD
2.MapPartitionsRDD

FileScanRDD

FileScan represents reading the data from a file.
It is given FilePartitions that are custom RDD partitions with PartitionedFiles (file blocks)
In our scenario, the *CSV file is read*

MapPartitionsRDD

MapPartitionsRDD will be created when you use map Partition transformation

SQLExecutionRDD

SQLExecutionRDD is Spark property that is used to track multiple Spark jobs that should all together constitute a single structured query execution.

Operation in Stage(2) and Stage(3) are
1.FileScanRDD
2.MapPartitionsRDD
3.WholeStageCodegen
4.Exchange

Wholestagecodegen

A physical query optimizer in Spark SQL that fuses multiple physical operators

Exchange

Exchange is performed because of the COUNT method.
As data is divided into partitions and shared among executors, to get count there should be adding of the count of from individual partition.

Represents the shuffle i.e data movement across the cluster(Executors).
It is the most expensive operation and if number of partitions is more exchange of data between executors will also be more.

2.3. Tasks

Tasks are located at the bottom space in the respective stage.


Tasks (1)

Show 20 entries Search:

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Errors
0	3	0	SUCCESS	NODE_LOCAL	driver	192.168.55.101		2020-07-20 16:11:35	49.0 ms		59 B / 1	

Key things to look task page are:
1. Input Size – Input for the Stage
2. Shuffle Write-Output is the stage written.

2.4. Storage

The Storage tab displays the persisted RDDs and DataFrames, if any, in the application. The summary page shows the storage levels, sizes and partitions of all RDDs, and the details page shows the sizes and using executors for all partitions in an RDD or DataFrame.

2.5. Environment Tab

This environment page has five parts. It is a useful place to check whether your properties have been set correctly.

1. **Runtime Information**: simply contains the runtime properties like versions of Java and Scala.
2. **Spark Properties**: lists the application properties like 'spark.app.name' and 'spark.driver.memory'.
3. **Hadoop Properties**: displays properties relative to Hadoop and YARN. **Note**: Properties like 'spark.hadoop' are shown not in this part but in 'Spark Properties'.
4. **System Properties**: shows more details about the JVM.
5. **Classpath Entries**: lists the classes loaded from different sources, which is very useful to resolve class conflicts.

2.6. Executors Tab

The Executors tab displays summary information about the executors that were created for the application, including memory and disk usage and task and shuffle information. The Storage Memory column shows the amount of memory used and reserved for caching data.

2.7. SQL Tab

If the application executes Spark SQL queries then the SQL tab displays information, such as the duration, Spark jobs, and physical and logical plans for the queries.

In our application, we performed read and count operation on files and DataFrame. So both read and count are listed SQL Tab

3. PySpark – What is SparkSession?

Since Spark 2.0 SparkSession has become an entry point to PySpark to work with RDD, and DataFrame. Prior to 2.0, SparkContext used to be an entry point.

What is SparkSession

SparkSession was introduced in version 2.0, It is an entry point to underlying PySpark functionality in order to programmatically create PySpark RDD, DataFrame. It's object *spark* is default available in pyspark-shell and it can be created programmatically using SparkSession.

3.1. SparkSession

With Spark 2.0 a new class SparkSession (`pyspark.sql import SparkSession`) has been introduced. SparkSession is a combined class for all different contexts we used to have prior to 2.0 release (SQLContext and HiveContext e.t.c). Since 2.0 SparkSession can be used in replace with SQLContext, HiveContext, and other contexts defined prior to 2.0.

As mentioned in the beginning SparkSession is an entry point to PySpark and creating a SparkSession instance would be the first statement you would write to program with RDD, DataFrame, and Dataset. SparkSession will be created using `SparkSession.builder` builder patterns.

Though SparkContext used to be an entry point prior to 2.0, It is not completely replaced with SparkSession, many features of SparkContext are still available and used in Spark 2.0 and later. You should also know that SparkSession internally creates SparkConfig and SparkContext with the configuration provided with SparkSession.

SparkSession also includes all the APIs available in different contexts –

- SparkContext,
- SQLContext,
- StreamingContext,
- HiveContext.

How many SparkSessions can you create in a PySpark application?

You can create as many `SparkSession` as you want in a PySpark application using either `SparkSession.builder()` or `SparkSession.newSession()`. Many Spark session objects are required when you wanted to keep PySpark tables (relational entities) logically separated.

```
# Create SparkSession from builder
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[1]") \
                    .appName('SparkByExamples.com') \
                    .getOrCreate()
```

`master()` – If you are running it on the cluster you need to use your master name as an argument to master(). usually, it would be either `yarn` or `mesos` depends on your cluster setup.

Use local[x] when running in Standalone mode. x should be an integer value and should be greater than 0; this represents how many partitions it should create when using RDD, DataFrame, and Dataset. Ideally, x value should be the number of CPU cores you have.

appName() – Used to set your application name.

getOrCreate() – This returns a SparkSession object if already exists, and creates a new one if not exist.

Note: SparkSession object spark is by default available in the PySpark shell.

```
# Create new SparkSession
spark2 = SparkSession.newSession
print(spark2)
# Get Existing SparkSession
spark3 = SparkSession.builder.getOrCreate
print(spark3)
# Usage of config()
spark = SparkSession.builder \
      .master("local[1]") \
      .appName("SparkByExamples.com") \
      .config("spark.some.config.option", "config-value") \
      .getOrCreate()
# Enabling Hive to use in Spark
spark = SparkSession.builder \
      .master("local[1]") \
      .appName("SparkByExamples.com") \
      .config("spark.sql.warehouse.dir", "<path>/spark-warehouse") \
      .enableHiveSupport() \
      .getOrCreate()
# Set Config
spark.conf.set("spark.executor.memory", "5g")

# Get a Spark Config
partions = spark.conf.get("spark.sql.shuffle.partitions")
print(partions)
# Create DataFrame
df = spark.createDataFrame(
    [("Scala", 25000), ("Spark", 35000), ("PHP", 21000)])
df.show()
# Spark SQL
df.createOrReplaceTempView("sample_table")
df2 = spark.sql("SELECT _1,_2 FROM sample_table")
df2.show()
# Create Hive table & query it.
spark.table("sample_table").write.saveAsTable("sample_hive_table")
df3 = spark.sql("SELECT _1,_2 FROM sample_hive_table")
df3.show()
```

3.2. SparkSession Commonly Used Methods

version() – Returns the Spark version where your application is running, probably the Spark version your cluster is configured with.

createDataFrame() – This creates a DataFrame from a collection and an RDD

getActiveSession() – returns an active Spark session.

read() – Returns an instance of DataFrameReader class, this is used to read records from csv, parquet, avro, and more file formats into DataFrame.

readStream() – Returns an instance of DataStreamReader class, this is used to read streaming data. that can be used to read streaming data into DataFrame.

sparkContext() – Returns a SparkContext.

sql() – Returns a DataFrame after executing the SQL mentioned.

sqlContext() – Returns SQLContext.

stop() – Stop the current SparkContext.

table() – Returns a DataFrame of a table or view.

udf() – Creates a PySpark UDF to use it on DataFrame, Dataset, and SQL.

4. PySpark SparkContext Explained

pyspark.SparkContext is an entry point to the PySpark functionality that is used to communicate with the cluster and to create an RDD, accumulator, and broadcast variables. In this article, you will learn how to create PySpark SparkContext with examples. Note that you can create only one SparkContext per JVM, in order to create another first you need to stop the existing one using stop() method.

The Spark driver program creates and uses SparkContext to connect to the cluster manager to submit PySpark jobs, and know what resource manager (YARN, Mesos, or Standalone) to communicate to. It is the heart of the PySpark application.

4.1. Create SparkContext in PySpark

At any given time only one SparkContext instance should be active per JVM. In case you want to create another you should stop existing SparkContext using stop() before creating a new one.

```
# Create SparkSession from builder
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[1]") \
                    .appName('SparkByExamples.com') \
                    .getOrCreate()
print(spark.sparkContext)
print("Spark App Name : "+ spark.sparkContext.appName)
# SparkContext stop() method
spark.sparkContext.stop()
```

As explained above you can have only one SparkContext per JVM. If you wanted to create another, you need to shutdown it first by using stop() method and create a new SparkContext.

When you try to create multiple SparkContext you will get the below error.

ValueError: Cannot run multiple SparkContexts at once;

4.2. Creating SparkContext prior to PySpark 2.0

You can create SparkContext by programmatically using its constructor, and pass parameters like master and appName at least as these are mandatory params. The below example creates context with a master as local and app name as Spark_Example_App.

```
# Create SparkContext
```

```
from pyspark import SparkContext
sc = SparkContext("local", "Spark_Example_App")
print(sc.appName)
```

You can also create it using SparkContext.getOrCreate(). It actually returns an existing active SparkContext otherwise creates one with a specified master and app name.

```
# Create Spark Context
from pyspark import SparkConf, SparkContext
conf = SparkConf()
conf.setMaster("local").setAppName("Spark Example App")
sc = SparkContext.getOrCreate(conf)
print(sc.appName)
# Create RDD
rdd = spark.sparkContext.range(1, 5)
print(rdd.collect())
```

4.3. SparkContext Commonly Used Variables

applicationId – Returns a unique ID of a PySpark application.

version – Version of PySpark cluster where your job is running.

uiWebUrl – Provides the Spark Web UI url that started by SparkContext.

4.4. SparkContext Commonly Used Methods

accumulator(value[, accum_param]) – It creates an pyspark accumulator variable with initial specified value. Only a driver can access accumulator variables.

broadcast(value) – read-only PySpark broadcast variable. This will be broadcast to the entire cluster. You can broadcast a variable to a PySpark cluster only once.

emptyRDD() – Creates an empty RDD

getOrCreate() – Creates or returns a SparkContext

hadoopFile() – Returns an RDD of a Hadoop file

newAPIHadoopFile() – Creates an RDD for a Hadoop file with a new API InputFormat.

sequenceFile() – Get an RDD for a Hadoop SequenceFile with given key and value types.

setLogLevel() – Change log level to debug, info, warn, fatal, and error

textFile() – Reads a text file from HDFS, local or any Hadoop supported file systems and returns an RDD

union() – Union two RDDs

wholeTextFiles() – Reads a text file in the folder from HDFS, local or any Hadoop supported file systems and returns an RDD of Tuple2. The first element of the tuple consists file name and the second element consists context of the text file.

SparkContext is an entry point to the PySpark execution engine which communicates with the cluster. Using this you can create a RDD, Accumulators and Broadcast variables.

5. PySpark RDD Tutorial | Learn with Examples

5.1. What is RDD (Resilient Distributed Dataset)?

RDD (Resilient Distributed Dataset) is a fundamental building block of PySpark which is fault-tolerant, immutable distributed collections of objects. Immutable meaning once you create an RDD you cannot change it. Each record in RDD is divided into logical partitions, which can be computed on different nodes of the cluster.

In other words, RDDs are a collection of objects similar to list in Python, with the difference being RDD is computed on several processes scattered across multiple physical servers also called nodes in a cluster while a Python collection lives and process in just one process.

Additionally, RDDs provide data abstraction of partitioning and distribution of the data designed to run computations in parallel on several nodes, while doing transformations on RDD we don't have to worry about the parallelism as PySpark by default provides.

5.2. PySpark RDD Benefits

In-Memory Processing

PySpark loads the data from disk and process in memory and keeps the data in memory, this is the main difference between PySpark and Mapreduce (I/O intensive). In between the transformations, we can also cache/persists the RDD in memory to reuse the previous computations.

Immutability

PySpark RDD's are immutable in nature meaning, once RDDs are created you cannot modify. When we apply transformations on RDD, PySpark creates a new RDD and maintains the RDD Lineage.

Fault Tolerance

PySpark operates on fault-tolerant data stores on HDFS, S3 e.t.c hence any RDD operation fails, it automatically reloads the data from other partitions. Also, When PySpark applications running on a cluster, PySpark task failures are automatically recovered for a certain number of times (as per the configuration) and finish the application seamlessly.

Lazy Evolution

PySpark does not evaluate the RDD transformations as they appear/encountered by Driver instead it keeps the all transformations as it encounters(DAG) and evaluates the all transformation when it sees the first RDD action.

Partitioning

When you create RDD from a data, It by default partitions the elements in a RDD. By default it partitions to the number of cores available.

5.3. PySpark RDD Limitations

PySpark RDDs are not much suitable for applications that make updates to the state store such as storage systems for a web application. For these applications, it is more efficient to use systems that perform traditional update

logging and data checkpointing, such as databases. The goal of RDD is to provide an efficient programming model for batch analytics and leave these asynchronous applications.

5.4. Creating RDD

RDD's are created primarily in two different ways,

- parallelizing an existing collection and
- referencing a dataset in an external storage system (HDFS, S3 and many more).

In realtime application, you will pass master from spark-submit instead of hardcoding on Spark application.

```
from pyspark.sql import SparkSession
spark:SparkSession = SparkSession.builder()
      .master("local[1]")
      .appName("SparkByExamples.com")
      .getOrCreate()
```

master() – If you are running it on the cluster you need to use your master name as an argument to master(). usually, it would be either yarn (Yet Another Resource Negotiator) or mesos depends on your cluster setup.

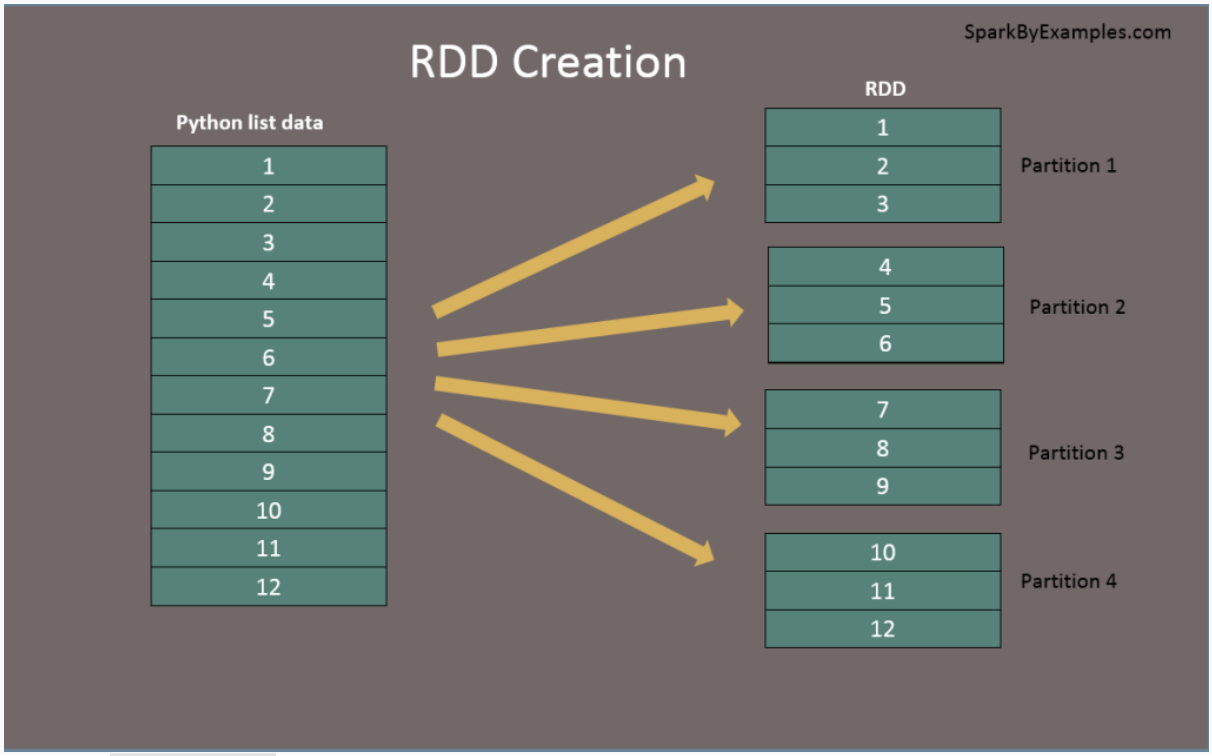
Use local[x] when running in Standalone mode. x should be an integer value and should be greater than 0; this represents how many partitions it should create when using RDD, DataFrame, and Dataset. Ideally, x value should be the number of CPU cores you have.

appName() – Used to set your application name.

getOrCreate() – This returns a SparkSession object if already exists, and creates a new one if not exist.

Note: Creating SparkSession object, internally creates one SparkContext per JVM.

Create RDD using sparkContext.parallelize()



By using parallelize() function of SparkContext (sparkContext.parallelize()) you can create an RDD. This function loads the existing collection from your driver program into parallelizing RDD. This is a basic method to create RDD and is used when you already have data in memory that is either loaded from a file or from a database. and it required all data to be present on the driver program prior to creating RDD.

```
#Create RDD from parallelize
```

```
data = [1,2,3,4,5,6,7,8,9,10,11,12]
rdd=spark.sparkContext.parallelize(data)
```

Create RDD using sparkContext.textFile()

Using textFile() method we can read a text (.txt) file into RDD.

```
#Create RDD from external Data source
rdd2 = spark.sparkContext.textFile("/path/textFile.txt")
```

Create RDD using sparkContext.wholeTextFiles()

wholeTextFiles() function returns a PairRDD with the key being the file path and value being file content.

```
#Reads entire file into a RDD as single record.
rdd3 = spark.sparkContext.wholeTextFiles("/path/textFile.txt")
```

Create empty RDD using sparkContext.emptyRDD

Using emptyRDD() method on sparkContext we can create an RDD with no data. This method creates an empty RDD with no partition.

```
# Creates empty RDD with no partition
rdd = spark.sparkContext.emptyRDD
# rddString = spark.sparkContext.emptyRDD[String]
```

Creating empty RDD with partition

Sometimes we may need to write an empty RDD to files by partition, In this case, you should create an empty RDD with partition.

```
#Create empty RDD with partition
rdd2 = spark.sparkContext.parallelize([],10) #This creates 10 partitions
```

5.5. RDD Parallelize

When we use parallelize() or textFile() or wholeTextFiles() methods of SparkContxt to initiate RDD, it automatically splits the data into partitions based on resource availability. when you run it on a laptop it would create partitions as the same number of cores available on your system.

```
print("initial partition count:"+str(rdd.getNumPartitions()))
#Outputs: initial partition count:2
```

5.6. Repartition and Coalesce

Sometimes we may need to repartition the RDD, PySpark provides two ways to repartition; first using repartition() method which shuffles data from all nodes also called full shuffle and second coalesce() method which shuffle data from minimum nodes, for examples if you have data in 4 partitions and doing coalesce(2) moves data from just 2 nodes.

```
reparRdd = rdd.repartition(4)
print("re-partition count:"+str(reparRdd.getNumPartitions()))
#Outputs: "re-partition count:4
```

Note: repartition() or coalesce() methods also returns a new RDD.

5.7. PySpark RDD Operations

RDD transformations – Transformations are lazy operations, instead of updating an RDD, these operations return another RDD.
RDD actions – operations that trigger computation and return RDD values.

5.7.1. RDD Transformations with example

Transformations on PySpark RDD returns another RDD and transformations are lazy meaning they don't execute until you call an action on RDD. Some transformations on RDD's are flatMap(), map(), reduceByKey(), filter(), sortByKey() and return new RDD instead of updating the current.

flatMap – flatMap() transformation flattens the RDD after applying the function and returns a new RDD. On the below example, first, it splits each record by space in an RDD and finally flattens it. Resulting RDD consists of a single word on each record.

```
rdd2 = rdd.flatMap(lambda x: x.split(" "))
```

map – map() transformation is used the apply any complex operations like adding a column, updating a column e.t.c, the output of map transformations would always have the same number of records as input.

In our word count example, we are adding a new column with value 1 for each word, the result of the RDD is PairRDDFunctions which contains key-value pairs, word of type String as Key and 1 of type Int as value.

```
rdd3 = rdd2.map(lambda x: (x,1))
```

reduceByKey – reduceByKey() merges the values for each key with the function specified. In our example, it reduces the word string by applying the sum function on value. The result of our RDD contains unique words and their count.

```
rdd5 = rdd4.reduceByKey(lambda a,b: a+b)
```

sortByKey – sortByKey() transformation is used to sort RDD elements on key. In our example, first, we convert RDD[(String,Int]) to RDD[(Int, String]) using map transformation and apply sortByKey which ideally does sort on an integer value. And finally, foreach with println statements returns all words in RDD and their count as key-value pair

```
rdd6 = rdd5.map(lambda x: (x[1],x[0])).sortByKey()
#Print rdd6 result to console
print(rdd6.collect())
```

filter – filter() transformation is used to filter the records in an RDD. In our example we are filtering all words starts with "a".

```
rdd4 = rdd3.filter(lambda x : 'an' in x[1])
print(rdd4.collect())
```

5.7.2. RDD Actions with example

RDD Action operations return the values from an RDD to a driver program. In other words, any RDD function that returns non-RDD is considered as an action.

count() – Returns the number of records in an RDD

```
# Action - count
print("Count : "+str(rdd6.count()))
```

first() – Returns the first record.

```
# Action - first
firstRec = rdd6.first()
print("First Record : "+str(firstRec[0]) + ","+ firstRec[1])
```

max() – Returns max record.

```
# Action - max
datMax = rdd6.max()
```

```
print("Max Record : "+str(datMax[0]) + ","+ datMax[1])
```
reduce() – Reduces the records to single, we can use this to count or sum.

```
# Action - reduce
totalWordCount = rdd6.reduce(lambda a,b: (a[0]+b[0],a[1]))
print("dataReduce Record : "+str(totalWordCount[0]))
```
take() – Returns the record specified as an argument.

```
# Action - take
data3 = rdd6.take(3)
for f in data3:
    print("data3 Key:"+ str(f[0]) +", Value:"+f[1])
```
collect() – Returns all data from RDD as an array. Be careful when you use this action when you are working with huge RDD with millions and billions of data as you may run out of memory on the driver.

```
# Action - collect
data = rdd6.collect()
for f in data:
    print("Key:"+ str(f[0]) +", Value:"+f[1])
```
saveAsTextFile() – Using saveAsTestFile action, we can write the RDD to a text file.

```
rdd6.saveAsTextFile("/tmp/wordCount")
```

5.8. Types of RDD

PairRDDFunctions or PairRDD – Pair RDD is a key-value pair This is mostly used RDD type,

ShuffledRDD –

DoubleRDD –

SequenceFileRDD –

HadoopRDD –

ParallelCollectionRDD –

5.9. Shuffle Operations

Shuffling is a mechanism PySpark uses to redistribute the data across different executors and even across machines. PySpark shuffling triggers when we perform certain transformation operations like gropByKey(), reduceByKey(), join() on RDDS

PySpark Shuffle is an expensive operation since it involves the following

- Disk I/O
- Involves data serialization and deserialization
- Network I/O

When creating an RDD, PySpark doesn't necessarily store the data for all keys in a partition since at the time of creation there is no way we can set the key for data set.

Hence, when we run the reduceByKey() operation to aggregate the data on keys, PySpark does the following. needs to first run tasks to collect all the data from all partitions and

For example, when we perform reduceByKey() operation, PySpark does the following

- PySpark first runs m*ap* tasks on all partitions which groups all values for a single key.
- The results of the map tasks are kept in memory.
- When results do not fit in memory, PySpark stores the data into a disk.
- PySpark shuffles the mapped data across partitions, some times it also stores the shuffled data into a disk for reuse when it needs to recalculate.

- Run the garbage collection
- Finally runs reduce tasks on each partition based on key.

PySpark RDD triggers shuffle and repartition for several operations
like repartition() and coalesce(), groupByKey(), reduceByKey(), cogroup() and join() but not `countByKey()` .

5.10. PySpark RDD Persistence Tutorial

PySpark *Cache* and P*ersist* are optimization techniques to improve the performance of the RDD jobs that are iterative and interactive

Using cache() and persist() methods, PySpark provides an optimization mechanism to store the intermediate computation of an RDD so they can be reused in subsequent actions.

When you persist or cache an RDD, each worker node stores it's partitioned data in memory or disk and reuses them in other actions on that RDD. And Spark's persisted data on nodes are fault-tolerant meaning if any partition is lost, it will automatically be recomputed using the original transformations that created it.

5.10.1. RDD Cache

PySpark RDD cache() method by default saves RDD computation to storage level `MEMORY_ONLY` meaning it will store the data in the JVM heap as unserialized objects.

PySpark cache() method in RDD class internally calls persist() method which in turn uses sparkSession.sharedState.cacheManager.cacheQuery to cache the result set of RDD. Let's look at an example.

```
cachedRdd = rdd.cache()
```

5.10.2. RDD Persist

PySpark persist() method is used to store the RDD to one of the storage levels MEMORY_ONLY,MEMORY_AND_DISK, MEMORY_ONLY_SER, MEMORY_AND_DISK_SER, DISK_ONLY, MEMORY_ONLY_2,MEMORY_AND_DISK_2 and more.

PySpark persist has two signature first signature doesn't take any argument which by default saves it to MEMORY_ONLY storage level and the second signature which takes StorageLevel as an argument to store it to different storage levels.

```
import pyspark
dfPersist = rdd.persist(pyspark.StorageLevel.MEMORY_ONLY)
dfPersist.show(false)
```

5.10.3. RDD Unpersist

PySpark automatically monitors every persist() and cache() calls you make and it checks usage on each node and drops persisted data if not used or by using least-recently-used (LRU) algorithm. You can also manually remove using unpersist() method. unpersist() marks the RDD as non-persistent, and remove all blocks for it from memory and disk.

5.11. PySpark Shared Variables

When PySpark executes transformation using map() or reduce() operations, It executes the transformations on a remote node by using the variables that are shipped with the tasks and these variables are not sent back to PySpark Driver hence there is no capability to reuse and sharing the variables across tasks. PySpark shared variables solve this problem using the below two techniques. PySpark provides two types of shared variables.

- Broadcast variables (read-only shared variable)
- Accumulator variables (updatable shared variables)

5.12. Creating RDD from DataFrame and vice-versa

```
# Converts RDD to DataFrame
dfFromRDD1 = rdd.toDF()
# Converts RDD to DataFrame with column names
dfFromRDD2 = rdd.toDF("col1","col2")
# using createDataFrame() - Convert DataFrame to RDD
df = spark.createDataFrame(rdd).toDF("col1","col2")
# Convert DataFrame to RDD
rdd = df.rdd
```

5.13. PySpark parallelize() – Create RDD from a list data

PySpark parallelize() is a function in SparkContext and is used to create an RDD from a list collection.

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
sparkContext=spark.sparkContext
rdd=sparkContext.parallelize([1,2,3,4,5])
rddCollect = rdd.collect()
print("Number of Partitions: "+str(rdd.getNumPartitions()))
print("Action: First element: "+str(rdd.first()))
print(rddCollect)
```

5.14. PySpark Repartition() vs Coalesce()

repartition() is used to increase or decrease the RDD/DataFrame partitions whereas the PySpark coalesce() is used to only decrease the number of partitions in an efficient way.

One important point to note is, PySpark repartition() and coalesce() are **very expensive operations** as they **shuffle the data across many partitions** hence try to minimize using these as much as possible.

5.14.1. PySpark RDD Repartition() vs Coalesce()

In RDD, you can create parallelism at the time of the creation of an RDD using parallelize(), textFile() and wholeTextFiles().

```
rdd = spark.sparkContext.parallelize((0,20))
print("From local[5]"+str(rdd.getNumPartitions()))

rdd1 = spark.sparkContext.parallelize((0,25), 6)
print("parallelize : "+str(rdd1.getNumPartitions()))

rddFromFile = spark.sparkContext.textFile("src/main/resources/test.txt",10)
print("TextFile : "+str(rddFromFile.getNumPartitions()))
rdd1.saveAsTextFile("/tmp/partition")
//Writes 6 part files, one for each partition
Partition 1 : 0 1 2
Partition 2 : 3 4 5
Partition 3 : 6 7 8 9
Partition 4 : 10 11 12
```

```
Partition 5 : 13 14 15
Partition 6 : 16 17 18 19
```

5.14.2. RDD repartition()

Spark RDD repartition() method is used to increase or decrease the partitions. The below example decreases the partitions from 10 to 4 by moving data from all partitions.

```
rdd2 = rdd1.repartition(4)
print("Repartition size : "+str(rdd2.getNumPartitions()))
rdd2.saveAsTextFile("/tmp/re-partition")
Partition 1 : 1 6 10 15 19
Partition 2 : 2 3 7 11 16
Partition 3 : 4 8 12 13 17
Partition 4 : 0 5 9 14 18
```

5.14.3. RDD coalesce()

Spark RDD `coalesce()` is used only to reduce the number of partitions. This is optimized or improved version of `repartition()` where the movement of the data across the partitions is lower using coalesce.

```
rdd3 = rdd1.coalesce(4)
print("Repartition size : "+str(rdd3.getNumPartitions()))
rdd3.saveAsTextFile("/tmp/coalesce")
Partition 1 : 0 1 2
Partition 2 : 3 4 5 6 7 8 9
Partition 4 : 10 11 12
Partition 5 : 13 14 15 16 17 18 19
```

5.14.4. PySpark DataFrame repartition() vs coalesce()

Like RDD, you can't specify the partition/parallelism while creating DataFrame. DataFrame by default internally uses the methods specified in Section 1 to determine the default partition and splits the data for parallelism.

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com') \
        .master("local[5]").getOrCreate()

df=spark.range(0,20)
print(df.rdd.getNumPartitions())

df.write.mode("overwrite").csv("c:/tmp/partition.csv")
```

The above example creates 5 partitions as specified in `master("local[5]")` and the data is distributed across all these 5 partitions.

```
Partition 1 : 0 1 2 3
Partition 2 : 4 5 6 7
Partition 3 : 8 9 10 11
Partition 4 : 12 13 14 15
Partition 5 : 16 17 18 19
```

5.14.5. DataFrame repartition()

Similar to RDD, the PySpark DataFrame repartition() method is used to increase or decrease the partitions. The below example increases the partitions from 5 to 6 by moving data from all partitions.

```
df2 = df.repartition(6)
print(df2.rdd.getNumPartitions())
```

Just increasing 1 partition results data movements from all partitions.

```
Partition 1 : 14 1 5
Partition 2 : 4 16 15
Partition 3 : 8 3 18
Partition 4 : 12 2 19
Partition 5 : 6 17 7 0
Partition 6 : 9 10 11 13
```

And, even decreasing the partitions also results in moving data from all partitions. hence when you wanted to decrease the partition recommendation is to use coalesce()

5.14.6. DataFrame coalesce()

Spark DataFrame coalesce() is used only to decrease the number of partitions. This is an optimized or improved version of repartition() where the movement of the data across the partitions is fewer using coalesce.

```
df3 = df.coalesce(2)
print(df3.rdd.getNumPartitions())
Partition 1 : 0 1 2 3 8 9 10 11
Partition 2 : 4 5 6 7 12 13 14 15 16 17 18 19
```

Since we are reducing 5 to 2 partitions, the data movement happens only from 3 partitions and it moves to remain 2 partitions.

5.15. Default Shuffle Partition

Calling groupBy(), union(), join() and similar functions on DataFrame results in shuffling data between multiple executors and even machines and finally repartitions data into **200** partitions by default. PySpark default defines shuffling partition to 200 using spark.sql.shuffle.partitions configuration.

6. PySpark –DataFrame with Examples

You can manually create a PySpark DataFrame using toDF() and createDataFrame() methods, both these function takes different signatures in order to create DataFrame from existing RDD, list, and DataFrame.

You can also create PySpark DataFrame from data sources like TXT, CSV, JSON, ORV, Avro, Parquet, XML formats by reading from HDFS, S3, DBFS, Azure Blob file systems e.t.c.

Finally, PySpark DataFrame also can be created by reading data from RDBMS Databases and NoSQL databases.

SPARKSESSION	RDD	DATAFRAME
createDataFrame(rdd)	toDF()	toDF(*cols)
createDataFrame(dataList)	toDF(*cols)	
createDataFrame(rowData,columns)		
createDataFrame(dataList,schema)		

columns = ["language","users_count"]

data = [("Java", "20000"), ("Python", "100000"), ("Scala", "3000")]

6.1. Create DataFrame from RDD

One easy way to manually create PySpark DataFrame is from an existing RDD. first, let's create a Spark RDD from a collection List by calling parallelize() function from SparkContext . We would need this rdd object for all our examples below.

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

rdd = spark.sparkContext.parallelize(data)

6.1.1. Using toDF() function

PySpark RDD's toDF() method is used to create a DataFrame from the existing RDD. Since RDD doesn't have columns, the DataFrame is created with default column names "_1" and "_2" as we have two columns.

dfFromRDD1 = rdd.toDF()

If you wanted to provide column names to the DataFrame use toDF() method with column names as arguments as shown below.

columns = ["language","users_count"]

dfFromRDD1 = rdd.toDF(columns)

dfFromRDD1.printSchema()

6.1.2. Using createDataFrame() from SparkSession

Using createDataFrame() from SparkSession is another way to create manually and it takes rdd object as an argument. and chain with toDF() to specify name to the columns.

dfFromRDD2 = spark.createDataFrame(rdd).toDF(*columns)

6.2. Create DataFrame from List Collection

we will see how to create PySpark DataFrame from a list. These examples would be similar to what we have seen in the above section with RDD, but we use the list data object instead of "rdd" object to create DataFrame.

6.2.1. Using createDataFrame() from SparkSession

Calling createDataFrame() from SparkSession is another way to create PySpark DataFrame manually, it takes a list object as an argument. and chain with toDF() to specify names to the columns.

dfFromData2 = spark.createDataFrame(data).toDF(*columns)

6.2.2. Using createDataFrame() with the Row type

createDataFrame() has another signature in PySpark which takes the collection of Row type and schema for column names as arguments. To use this first we need to convert our “data” object from the list to list of Row.

```
rowData = map(lambda x: Row(*x), data)

dfFromData3 = spark.createDataFrame(rowData,columns)
```

6.3. Create DataFrame with schema

```
from pyspark.sql.types import StructType,StructField, StringType, IntegerType

data2 = [("James","","Smith","36636","M",3000),

    ("Michael","Rose","","40288","M",4000),

    ("Robert","","Williams","42114","M",4000),

    ("Maria","Anne","Jones","39192","F",4000),

    ("Jen","Mary","Brown","","F",-1)

  ]


schema = StructType([ \

    StructField("firstname",StringType(),True), \

    StructField("middlename",StringType(),True), \

    StructField("lastname",StringType(),True), \

    StructField("id", StringType(), True), \

    StructField("gender", StringType(), True), \

    StructField("salary", IntegerType(), True) \

  ])


df = spark.createDataFrame(data=data2,schema=schema)

df.printSchema()

df.show(truncate=False)
```

6.4. Create DataFrame from Data sources

In real-time mostly you create DataFrame from data source files like CSV, Text, JSON, XML e.t.c.

PySpark by default supports many data formats out of the box without importing any libraries and to create DataFrame you need to use the appropriate method available in DataFrameReader class.

6.4.1. Creating DataFrame from CSV

Use csv() method of the DataFrameReader object to create a DataFrame from CSV file. you can also provide options like what delimiter to use, whether you have quoted data, date formats, infer schema, and many more. Please refer PySpark Read CSV into DataFrame

df2 = spark.read.csv("/src/resources/file.csv")

6.4.2. Creating from text (TXT) file

df2 = spark.read.text("/src/resources/file.txt")

6.4.3. Creating from JSON file

df2 = spark.read.json("/src/resources/file.json")

6.5. PySpark – Create an Empty DataFrame & RDD

While working with files, sometimes we may not receive a file for processing, however, we still need to create a DataFrame manually with the same schema we expect. If we don't create with the same schema, our operations/transformations (like union's) on DataFrame fail as we refer to the columns that may not present.

To handle situations similar to these, we always need to create a DataFrame with the same schema, which means the same column names and datatypes regardless of the file exists or empty file processing.

6.5.1. Create Empty RDD in PySpark

Create an empty RDD by using `emptyRDD()` of SparkContext for example `spark.sparkContext.emptyRDD()`.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

#Creates Empty RDD
emptyRDD = spark.sparkContext.emptyRDD()
print(emptyRDD)

#Diplays
#EmptyRDD[188] at emptyRDD
```

Alternatively you can also get empty RDD by using `spark.sparkContext.parallelize([])`.

```
#Creates Empty RDD using parallelize
rdd2= spark.sparkContext.parallelize([])
print(rdd2)

#EmptyRDD[205] at emptyRDD at NativeMethodAccessorImpl.java:0
#ParallelCollectionRDD[206] at readRDDFromFile at PythonRDD.scala:262
```

Note: If you try to perform operations on empty RDD you going to get `ValueError("RDD is empty")`.

6.5.2. Create Empty DataFrame with Schema (StructType)

In order to create an empty PySpark DataFrame manually with schema (column names & data types) first, Create a schema using StructType and StructField .

```
#Create Schema
from pyspark.sql.types import StructType,StructField, StringType
schema = StructType([
  StructField('firstname', StringType(), True),
  StructField('middlename', StringType(), True),
  StructField('lastname', StringType(), True)
  ])
```

Now use the empty RDD created above and pass it to `createDataFrame()` of SparkSession along with the schema for column names & data types.

```
#Create empty DataFrame from empty RDD
df = spark.createDataFrame(emptyRDD,schema)
df.printSchema()
```

This yields below schema of the empty DataFrame.

6.5.3. Convert Empty RDD to DataFrame

You can also create empty DataFrame by converting empty RDD to DataFrame using `toDF()`.

```
#Convert empty RDD to Dataframe
df1 = emptyRDD.toDF(schema)
df1.printSchema()
```

6.5.4. Create Empty DataFrame with Schema.

So far I have covered creating an empty DataFrame from RDD, but here will create it manually with schema and without RDD.

```
#Create empty DataFrame directly.
df2 = spark.createDataFrame([], schema)
df2.printSchema()
```

6.5.5. Create Empty DataFrame without Schema (no columns)

To create empty DataFrame with out schema (no columns) just create a empty schema and use it while creating PySpark DataFrame.

```
#Create empty DatFrame with no schema (no columns)
df3 = spark.createDataFrame([], StructType([]))
df3.printSchema()

#print below empty schema
#root
```

6.6. PySpark DataFrame show() Syntax & Example

```
show(self, n=20, truncate=True, vertical=False)
```

6.7. Defining Nested StructType object struct

```
structureData = [
    (("James","","Smith"),"36636","M",3100),
    (("Michael","Rose",""),"40288","M",4300),
    (("Robert","","Williams"),"42114","M",1400),
    (("Maria","Anne","Jones"),"39192","F",5500),
    (("Jen","Mary","Brown"),"","F",-1)
  ]
structureSchema = StructType([
        StructField('name', StructType([
             StructField('firstname', StringType(), True),
             StructField('middlename', StringType(), True),
             StructField('lastname', StringType(), True)
            ])),
         StructField('id', StringType(), True),
         StructField('gender', StringType(), True),
         StructField('salary', IntegerType(), True)
         ])

df2 = spark.createDataFrame(data=structureData,schema=structureSchema)
df2.printSchema()
df2.show(truncate=False)
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)
```

6.7.1. Adding & Changing struct of the DataFrame

```
from pyspark.sql.functions import col,struct,when
updatedDF = df2.withColumn("OtherInfo",
    struct(col("id").alias("identifier"),
    col("gender").alias("gender"),
    col("salary").alias("salary"),
    when(col("salary").cast(IntegerType()) < 2000,"Low")
      .when(col("salary").cast(IntegerType()) < 4000,"Medium")
      .otherwise("High").alias("Salary_Grade")
  )).drop("id","gender","salary")

updatedDF.printSchema()
updatedDF.show(truncate=False)
```

Here, it copies “gender“, “salary” and “id” to the new struct “otherInfo” and add’s a new column “Salary_Grade“.

```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- OtherInfo: struct (nullable = false)
 |    |-- identifier: string (nullable = true)
 |    |-- gender: string (nullable = true)
 |    |-- salary: integer (nullable = true)
 |    |-- Salary_Grade: string (nullable = false)
```

6.8. Using SQL ArrayType and MapType

```
arrayStructureSchema = StructType([
    StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
        ])),
        StructField('hobbies', ArrayType(StringType()), True),
        StructField('properties', MapType(StringType(),StringType()), True)
```

```
    ])
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- hobbies: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- properties: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

6.9. Checking if a Column Exists in a DataFrame

```
print(df.schema.fieldNames.contains("firstname"))
print(df.schema.contains(StructField("firstname",StringType,true)))
```

6.10. PySpark Row using on DataFrame and RDD

In PySpark Row class is available by importing `pyspark.sql.Row` which is represented as a record/row in DataFrame, one can create a Row object by using named arguments, or create a custom Row like class. In this article I will explain how to use Row class on RDD, DataFrame and its functions.

Before we start using it on RDD & DataFrame, let's understand some basics of Row class.

Key Points of Row Class:

- Earlier to Spark 3.0, when used Row class with named arguments, the fields are sorted by name.
- Since 3.0, Rows created from named arguments are not sorted alphabetically instead they will be ordered in the position entered.
- To enable sorting by names, set the environment variable `PYSPARK_ROW_FIELD_SORTING_ENABLED` to `true`.
- Row class provides a way to create a struct-type column as well.

6.10.1. *Create a Row Object*

Row class extends the tuple hence it takes variable number of arguments, Row() is used to create the row object. Once the row object created, we can retrieve the data from Row using index similar to tuple.

```
from pyspark.sql import Row
row=Row("James",40)
print(row[0] +","+str(row[1]))
```

This outputs `James,40`. Alternatively you can also write with named arguments. Benefits with the named argument is you can access with field name `row.name`. Below example print "Alice".

```
row=Row(name="Alice", age=11)
```

```
print(row.name)
```

6.10.2. Create Custom Class from Row

We can also create a Row like class, for example “Person” and use it similar to Row object. This would be helpful when you wanted to create real time object and refer it’s properties. On below example, we have created a Person class and used similar to Row.

```
Person = Row("name", "age")
p1=Person("James", 40)
p2=Person("Alice", 35)
print(p1.name +","+p2.name)
```

This outputs `James,Alice`

6.10.3. Using Row class on PySpark RDD

We can use Row class on PySpark RDD. When you use Row to create an RDD, after collecting the data you will get the result back in Row.

```
from pyspark.sql import SparkSession, Row
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [Row(name="James,,Smith",lang=["Java","Scala","C++"],state="CA"),
    Row(name="Michael,Rose,",lang=["Spark","Java","C++"],state="NJ"),
    Row(name="Robert,,Williams",lang=["CSharp","VB"],state="NV")]
rdd=spark.sparkContext.parallelize(data)
print(rdd.collect())
```

This yields below output.

```
[Row(name='James,,Smith', lang=['Java', 'Scala', 'C++'], state='CA'), Row(name='Michael,Rose,',
lang=['Spark', 'Java', 'C++'], state='NJ'), Row(name='Robert,,Williams', lang=['CSharp', 'VB'],
state='NV')]
```

Now, let’s collect the data and access the data using its properties.

```
collData=rdd.collect()
for row in collData:
    print(row.name + "," +str(row.lang))
```

This yields below output.

```
James,,Smith,['Java', 'Scala', 'C++']
Michael,Rose,,['Spark', 'Java', 'C++']
Robert,,Williams,['CSharp', 'VB']
```

Alternatively, you can also do by creating a Row like class “Person”

```
Person=Row("name","lang","state")
data = [Person("James,,Smith",["Java","Scala","C++"],"CA"),
    Person("Michael,Rose,",["Spark","Java","C++"],"NJ"),
    Person("Robert,,Williams",["CSharp","VB"],"NV")]
```

6.10.4. Using Row class on PySpark DataFrame

Similarly, Row class also can be used with PySpark DataFrame, By default data in DataFrame represent as Row. To demonstrate, I will use the same data that was created for RDD.

Note that Row on DataFrame is not allowed to omit a named argument to represent that the value is None or missing. This should be explicitly set to None in this case.

```
df=spark.createDataFrame(data)
df.printSchema()
df.show()
```

This yields below output. Note that DataFrame able to take the column names from Row object.

```
root
 |-- name: string (nullable = true)
 |-- lang: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- state: string (nullable = true)
```

You can also change the column names by using `toDF()` function

```
columns = ["name","languagesAtSchool","currentState"]
df=spark.createDataFrame(data).toDF(*columns)
df.printSchema()
```

This yields below output, note the column name "languagesAtSchool" from the previous example.

```
root
 |-- name: string (nullable = true)
 |-- languagesAtSchool: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- currentState: string (nullable = true)
```

6.10.5. Create Nested Struct Using Row Class

The below example provides a way to create a struct type using the Row class. Alternatively, you can also create struct type using By Providing Schema using PySpark StructType & StructFields

```
#Create DataFrame with struct using Row class
from pyspark.sql import Row
data=[Row(name="James",prop=Row(hair="black",eye="blue")),
      Row(name="Ann",prop=Row(hair="grey",eye="black"))]
df=spark.createDataFrame(data)
df.printSchema()
```

Yields below schema

```
root
 |-- name: string (nullable = true)
 |-- prop: struct (nullable = true)
 |    |-- hair: string (nullable = true)
 |    |-- eye: string (nullable = true)
```

6.10.6. Complete Example of PySpark Row usage on RDD & DataFrame

Below is complete example for reference.

```
from pyspark.sql import SparkSession, Row

row=Row("James",40)
print(row[0] +","+str(row[1]))
row2=Row(name="Alice", age=11)
print(row2.name)

Person = Row("name", "age")
p1=Person("James", 40)
p2=Person("Alice", 35)
print(p1.name +","+p2.name)

#PySpark Example
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [Row(name="James,,Smith",lang=["Java","Scala","C++"],state="CA"),
    Row(name="Michael,Rose,",lang=["Spark","Java","C++"],state="NJ"),
    Row(name="Robert,,Williams",lang=["CSharp","VB"],state="NV")]

#RDD Example 1
rdd=spark.sparkContext.parallelize(data)
collData=rdd.collect()
print(collData)
for row in collData:
    print(row.name + "," +str(row.lang))

# RDD Example 2
Person=Row("name","lang","state")
data = [Person("James,,Smith",["Java","Scala","C++"],"CA"),
    Person("Michael,Rose,",["Spark","Java","C++"],"NJ"),
    Person("Robert,,Williams",["CSharp","VB"],"NV")]
rdd=spark.sparkContext.parallelize(data)
collData=rdd.collect()
print(collData)
for person in collData:
    print(person.name + "," +str(person.lang))

#DataFrame Example 1
columns = ["name","languagesAtSchool","currentState"]
df=spark.createDataFrame(data)
df.printSchema()
df.show()

collData=df.collect()
print(collData)
for row in collData:
```

```
    print(row.name + "," +str(row.lang))

#DataFrame Example 2
columns = ["name","languagesAtSchool","currentState"]
df=spark.createDataFrame(data).toDF(*columns)
df.printSchema()
```

7. PySpark Column Class | Operators & Functions

`pyspark.sql.Column` class provides several functions to work with DataFrame to manipulate the Column values, evaluate the boolean expression to filter rows, retrieve a value or part of a value from a DataFrame column, and to work with list, map & struct columns.

Key Points:

- PySpark Column class represents a single Column in a DataFrame.
- It provides functions that are most used to manipulate DataFrame Columns & Rows.
- Some of these Column functions evaluate a Boolean expression that can be used with filter() transformation to filter the DataFrame Rows.
- Provides functions to get a value from a list column by index, map value by key & index, and finally struct nested column.
- PySpark also provides additional functions pyspark.sql.functions that take Column object and return a Column type.
-

Note: Most of the pyspark.sql.functions return Column type hence it is very important to know the operation you can perform with Column type.

7.1. Create Column Class Object

One of the simplest ways to create a Column class object is by using PySpark lit() SQL function, this takes a literal value and returns a Column object.

```
from pyspark.sql.functions import lit
colObj = lit("sparkbyexamples.com")
```

You can also access the Column from DataFrame by multiple ways.

```
data=[("James",23),("Ann",40)]
df=spark.createDataFrame(data).toDF("name.fname","gender")
df.printSchema()
#root
# |-- name.fname: string (nullable = true)
# |-- gender: long (nullable = true)

# Using DataFrame object (df)
df.select(df.gender).show()
df.select(df["gender"]).show()
#Accessing column name with dot (with backticks)
df.select(df["`name.fname`"]).show()

#Using SQL col() function
from pyspark.sql.functions import col
df.select(col("gender")).show()
```

```
#Accessing column name with dot (with backticks)
df.select(col("`name.fname`")).show()
```

Below example demonstrates accessing struct type columns. Here I have use PySpark Row class to create a struct type. Alternatively you can also create it by using PySpark StructType & StructField classes

```
#Create DataFrame with struct using Row class
from pyspark.sql import Row
data=[Row(name="James",prop=Row(hair="black",eye="blue")),
      Row(name="Ann",prop=Row(hair="grey",eye="black"))]
df=spark.createDataFrame(data)
df.printSchema()
#root
# |-- name: string (nullable = true)
# |-- prop: struct (nullable = true)
# |    |-- hair: string (nullable = true)
# |    |-- eye: string (nullable = true)

#Access struct column
df.select(df.prop.hair).show()
df.select(df["prop.hair"]).show()
df.select(col("prop.hair")).show()

#Access all columns from struct
df.select(col("prop.*")).show()
```

7.2. PySpark Column Operators

PySpark column also provides a way to do arithmetic operations on columns using operators.

```
data=[(100,2,1),(200,3,4),(300,4,4)]
df=spark.createDataFrame(data).toDF("col1","col2","col3")

#Arthmetic operations
df.select(df.col1 + df.col2).show()
df.select(df.col1 - df.col2).show()
df.select(df.col1 * df.col2).show()
df.select(df.col1 / df.col2).show()
df.select(df.col1 % df.col2).show()

df.select(df.col2 > df.col3).show()
df.select(df.col2 < df.col3).show()
df.select(df.col2 == df.col3).show()
```

7.3. PySpark Column Functions

Let's see some of the most used Column Functions, on below table, I have grouped related functions together to make it easy, click on the link for examples.

COLUMN FUNCTION	FUNCTION DESCRIPTION
alias(*alias, **kwargs) name(*alias, **kwargs)	Provides alias to the column or expressions name() returns same as alias().
asc() asc_nulls_first() asc_nulls_last()	Returns ascending order of the column. asc_nulls_first() Returns null values first then non-null values. asc_nulls_last() – Returns null values after non-null values.
astype(dataType) cast(dataType)	Used to cast the data type to another type. astype() returns same as cast().
between(lowerBound, upperBound)	Checks if the columns values are between lower and upper bound. Returns boolean value.
bitwiseAND(other) bitwiseOR(other) bitwiseXOR(other)	Compute bitwise AND, OR & XOR of this expression with another expression respectively.
contains(other)	Check if String contains in another string.
desc() desc_nulls_first() desc_nulls_last()	Returns descending order of the column. desc_nulls_first() -null values appear before non-null values. desc_nulls_last() – null values appear after non-null values.
startswith(other) endswith(other)	String starts with. Returns boolean expression String ends with. Returns boolean expression
eqNullSafe(other)	Equality test that is safe for null values.
getField(name)	Returns a field by name in a StructField and by key in Map.
getItem(key)	Returns a values from Map/Key at the provided position.
isNotNull() isNull()	isNotNull() – Returns True if the current expression is NOT null. isNull() – Returns True if the current expression is null.
isin(*cols)	A boolean expression that is evaluated to true if the value of this expression is contained by the evaluated values of the arguments.
like(other) rlike(other)	Similar to SQL like expression. Similar to SQL RLIKE expression (LIKE with Regex).
over(window)	Used with window column
substr(startPos, length)	Return a Column which is a substring of the column.

COLUMN FUNCTION	FUNCTION DESCRIPTION
when(condition, value) otherwise(value)	Similar to SQL CASE WHEN, Executes a list of conditions and returns one of multiple possible result expressions.
dropFields(*fieldNames)	Used to drops fields in StructType by name.
withField(fieldName, col)	An expression that adds/replaces a field in StructType by name.

7.4. PySpark Column Functions Examples

Let's create a simple DataFrame to work with PySpark SQL Column examples. For most of the examples below, I will be referring DataFrame object name (df.) to get the column.

```
data=[("James","Bond","100",None),
      ("Ann","Varsa","200",'F'),
      ("Tom Cruise","XXX","400",''),
      ("Tom Brand",None,"400",'M')]
columns=["fname","lname","id","gender"]
df=spark.createDataFrame(data,columns)
```

7.4.1. alias() – Set's name to Column

On below example `df.fname` refers to Column object and `alias()` is a function of the Column to give alternate name. Here, `fname` column has been changed to `first_name` & `lname` to `last_name`.

On second example I have use PySpark expr() function to concatenate columns and named column as `fullName`.

```
#alias
from pyspark.sql.functions import expr
df.select(df.fname.alias("first_name"), \
          df.lname.alias("last_name")
    ).show()

#Another example
df.select(expr(" fname ||','|| lname").alias("fullName") \
    ).show()
```

7.4.2. asc() & desc() – Sort the DataFrame columns by Ascending or Descending order.

```
#asc, desc to sort ascending and descending order repsectively.
df.sort(df.fname.asc()).show()
df.sort(df.fname.desc()).show()
```

7.4.3. cast() & astype() – Used to convert the data Type.

```
#cast
df.select(df.fname,df.id.cast("int")).printSchema()
```

7.4.4. between() – Returns a Boolean expression when a column values in between lower and upper bound.

```
#between
df.filter(df.id.between(100,300)).show()
```

7.4.5. contains() – Checks if a DataFrame column value contains a a value specified in this function.

```
#contains
df.filter(df.fname.contains("Cruise")).show()
```

7.4.6. startswith() & endswith() – Checks if the value of the DataFrame Column starts and ends with a String respectively.

```
#startswith, endswith()
df.filter(df.fname.startswith("T")).show()
df.filter(df.fname.endswith("Cruise")).show()
```

7.4.7. isNull & isNotNull() – Checks if the DataFrame column has NULL or non NULL values.

Refer to

```
#isNull & isNotNull
df.filter(df.lname.isNull()).show()
df.filter(df.lname.isNotNull()).show()
```

7.4.8. like() & rlike() – Similar to SQL LIKE expression

```
#like , rlike
df.select(df.fname,df.lname,df.id) \
  .filter(df.fname.like("%om"))
```

7.4.9. substr() – Returns a Column after getting sub string from the Column

```
df.select(df.fname.substr(1,2).alias("substr")).show()
```

7.4.10. when() & otherwise() – It is similar to SQL Case When, executes sequence of expressions until it matches the condition and returns a value when match.

```
#when & otherwise
from pyspark.sql.functions import when
df.select(df.fname,df.lname,when(df.gender=="M","Male") \
                .when(df.gender=="F","Female") \
                .when(df.gender==None ,"") \
                .otherwise(df.gender).alias("new_gender") \
    ).show()
```

7.4.11. isin() – Check if value presents in a List.

```
#isin
li=["100","200"]
df.select(df.fname,df.lname,df.id) \
  .filter(df.id.isin(li)) \
  .show()
```

7.4.12. getField() – To get the value by key from MapType column and by stuct child name from StructType column

Rest of the below functions operates on List, Map & Struct data structures hence to demonstrate these I will use another DataFrame with list, map and struct columns. For more explanation how to use Arrays refer to PySpark ArrayType Column on DataFrame Examples & for map refer to PySpark MapType Examples

```
#Create DataFrame with struct, array & map
from pyspark.sql.types import StructType,StructField,StringType,ArrayType,MapType
data=[(("James","Bond"),["Java","C#"],{'hair':'black','eye':'brown'}),
        (("Ann","Varsa"),[".NET","Python"],{'hair':'brown','eye':'black'}),
        (("Tom Cruise",""),["Python","Scala"],{'hair':'red','eye':'grey'}),
        (("Tom Brand",None),["Perl","Ruby"],{'hair':'black','eye':'blue'})]

schema = StructType([
         StructField('name', StructType([
             StructField('fname', StringType(), True),
             StructField('lname', StringType(), True)])),
         StructField('languages', ArrayType(StringType()),True),
         StructField('properties', MapType(StringType(),StringType()),True)
      ])
df=spark.createDataFrame(data,schema)
df.printSchema()

#Display's to console
root
 |-- name: struct (nullable = true)
 |    |-- fname: string (nullable = true)
```

```
 |    |-- lname: string (nullable = true)
 |-- languages: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- properties: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

getField Example

```
#getField from MapType
df.select(df.properties.getField("hair")).show()

#getField from Struct
df.select(df.name.getField("fname")).show()
```

7.4.13. getItem() – To get the value by index from MapType or ArrayTupe & ny key for MapType column.

```
#getItem() used with ArrayType
df.select(df.languages.getItem(1)).show()

#getItem() used with MapType
df.select(df.properties.getItem("hair")).show()
```

8. PySpark Select Columns From DataFrame

```
df.select("firstname","lastname").show()
df.select(df.firstname,df.lastname).show()
df.select(df["firstname"],df["lastname"]).show()

#By using col() function
from pyspark.sql.functions import col
df.select(col("firstname"),col("lastname")).show()

#Select columns by regular expression
df.select(df.colRegex("`^.*name*`")).show()

# Select All columns from List
df.select(*columns).show()

# Select All columns
df.select([col for col in df.columns]).show()
df.select("*").show()

#Selects first 3 columns and top 3 rows
df.select(df.columns[:3]).show(3)

#Selects columns 2 to 4  and top 3 rows
df.select(df.columns[2:4]).show(3)

#Select Nested Struct Columns from PySpark
df2.select("name").show(truncate=False)
df2.select("name.firstname","name.lastname").show(truncate=False)
df2.select("name.*").show(truncate=False)
```

9. PySpark withColumn() & withColumnRenamed() Usage with Examples

PySpark `withColumn()` is a transformation function of DataFrame which is used to change the value, convert the datatype of an existing column, create a new column, and many more

```
# Change DataType using PySpark withColumn()
df.withColumn("salary",col("salary").cast("Integer")).show()

# Update The Value of an Existing Column
df.withColumn("salary",col("salary")*100).show()

# Create a Column from an Existing
df.withColumn("CopiedColumn",col("salary")* -1).show()

# Add a New Column using withColumn()
df.withColumn("Country", lit("USA")).show()
df.withColumn("Country", lit("USA")) \
  .withColumn("anotherColumn",lit("anotherValue")) \
  .show()

# Rename Column Name
df.withColumnRenamed("gender","sex") \
  .show(truncate=False)

# Drop Column From PySpark DataFrame
df.drop("salary") \
  .show()

# To rename multiple columns
df2 = df.withColumnRenamed("dob","DateOfBirth") \
     .withColumnRenamed("salary","salary_amount")

# To rename nested elements
df.select(col("name.firstname").alias("fname"), \
  col("name.middlename").alias("mname"), \
  col("name.lastname").alias("lname"), \
  col("dob"),col("gender"),col("salary")) \
  .printSchema()

# To rename a nested column in Dataframe
df.select(col("name").cast(schema2), \
      col("dob"), col("gender"),col("salary")) \
    .printSchema()

# To rename nested columns
df4 = df.withColumn("fname",col("name.firstname")) \
       .withColumn("mname",col("name.middlename")) \
```

```
        .withColumn("lname",col("name.lastname")) \
        .drop("name")

# To change all columns in a PySpark DataFrame
newColumns = ["newCol1","newCol2","newCol3","newCol4"]
df.toDF(*newColumns).printSchema()
```

10. PySpark Where Filter Function | Multiple Conditions

PySpark `filter()` function is used to filter the rows from RDD/DataFrame based on the given condition or SQL expression, you can also use `where()` clause instead of the filter() if you are coming from an SQL background, both these functions operate exactly the same.

```
df.filter(df.state == "OH") \
    .show(truncate=False)

df.filter(col("state") == "OH") \
    .show(truncate=False)

df.filter("gender == 'M'") \
    .show(truncate=False)

df.filter( (df.state  == "OH") & (df.gender  == "M") ) \
    .show(truncate=False)

df.filter(array_contains(df.languages,"Java")) \
    .show(truncate=False)

df.filter(df.name.lastname == "Williams") \
    .show(truncate=False)
```

11. PySpark – Distinct to Drop Duplicate Rows

PySpark `distinct()` function is used to drop/remove the duplicate rows (all columns) from DataFrame and `dropDuplicates()` is used to drop rows based on selected (one or multiple) columns. In this article, you will learn how to use distinct() and dropDuplicates() functions with PySpark example.

```
#Distinct
distinctDF = df.distinct()
print("Distinct count: "+str(distinctDF.count()))
distinctDF.show(truncate=False)

#Drop duplicates
df2 = df.dropDuplicates()
print("Distinct count: "+str(df2.count()))
df2.show(truncate=False)
```

```
#Drop duplicates on selected columns
dropDisDF = df.dropDuplicates(["department","salary"])
print("Distinct count of department salary : "+str(dropDisDF.count()))
dropDisDF.show(truncate=False)
}
```

12. PySpark orderBy() and sort() explained

You can use either `sort()` or `orderBy()` function of PySpark DataFrame to sort DataFrame by ascending or descending order based on single or multiple columns, you can also do sorting using PySpark SQL sorting functions,

```
df.sort("department","state").show(truncate=False)
df.sort(col("department"),col("state")).show(truncate=False)

df.orderBy("department","state").show(truncate=False)
df.orderBy(col("department"),col("state")).show(truncate=False)

df.sort(df.department.asc(),df.state.asc()).show(truncate=False)
df.sort(col("department").asc(),col("state").asc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").asc()).show(truncate=False)

df.sort(df.department.asc(),df.state.desc()).show(truncate=False)
df.sort(col("department").asc(),col("state").desc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").desc()).show(truncate=False)

df.createOrReplaceTempView("EMP")
spark.sql("select employee_name,department,state,salary,age,bonus from EMP ORDER BY department asc").show(truncate=False)
```

13. PySpark UDF (User Defined Function)

PySpark UDF (a.k.a User Defined Function) is the most useful feature of Spark SQL & DataFrame that is used to extend the PySpark build in capabilities. In this article, I will explain what is UDF? why do we need it and how to create and use it on DataFrame `select()`, withColumn() and SQL using PySpark (Spark with Python) examples.

Note: UDF's are the most expensive operations hence use them only you have no choice and when essential.

```
def convertCase(str):
    resStr=""
    arr = str.split(" ")
    for x in arr:
        resStr= resStr + x[0:1].upper() + x[1:len(x)] + " "
    return resStr
""" Converting function to UDF """
convertUDF = udf(lambda z: convertCase(z),StringType())
""" Converting function to UDF
StringType() is by default hence not required """
convertUDF = udf(lambda z: convertCase(z))
```

```
def upperCase(str):
    return str.upper()

upperCaseUDF = udf(lambda z:upperCase(z),StringType())

df.withColumn("Cureated Name", upperCaseUDF(col("Name"))) \
  .show(truncate=False)

""" Using UDF on SQL """Registering PySpark UDF & use it on SQL
spark.udf.register("convertUDF", convertCase,StringType())
df.createOrReplaceTempView("NAME_TABLE")
spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE") \
     .show(truncate=False)

# Creating UDF using annotation
@udf(returnType=StringType())
def upperCase(str):
    return str.upper()

df.withColumn("Cureated Name", upperCase(col("Name"))) \
.show(truncate=False)
```

14. PySpark fillna() & fill() – Replace NULL/None Values

In PySpark, DataFrame.`fillna()` or `DataFrameNaFunctions.fill()` is used to replace NULL/None values on all or selected multiple DataFrame columns with either **zero(0), empty string, space, or any constant literal** values.

While working on PySpark DataFrame we often need to replace null values since certain operations on null value return error hence, we need to graciously handle nulls as the first step before processing. Also, while writing to a file, it's always best practice to replace null values, not doing this result nulls on the output file.

PySpark provides DataFrame.fillna() and DataFrameNaFunctions.fill() to replace NULL/None values. These two are aliases of each other and returns the same results.

```
#Replace 0 for null for all integer columns
df.na.fill(value=0).show()

#Replace 0 for null on only population column
df.na.fill(value=0,subset=["population"]).show()

df.fillna(value="").show()
df.na.fill(value="").show()

df.fillna("unknown",["city"]) \
    .fillna("",["type"]).show()

df.fillna({"city": "unknown", "type": ""}) \
```

```
    .show()

df.na.fill("unknown",["city"]) \
    .na.fill("",["type"]).show()

df.na.fill({"city": "unknown", "type": ""}) \
    .show()
```

15. PySpark Aggregate Functions with Examples

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import approx_count_distinct,collect_list
from pyspark.sql.functions import collect_set,sum,avg,max,countDistinct,count
from pyspark.sql.functions import first, last, kurtosis, min, mean, skewness
from pyspark.sql.functions import stddev, stddev_samp, stddev_pop, sumDistinct
from pyspark.sql.functions import variance,var_samp,  var_pop

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

simpleData = [("James", "Sales", 3000),
    ("Michael", "Sales", 4600),
    ("Robert", "Sales", 4100),
    ("Maria", "Finance", 3000),
    ("James", "Sales", 3000),
    ("Scott", "Finance", 3300),
    ("Jen", "Finance", 3900),
    ("Jeff", "Marketing", 3000),
    ("Kumar", "Marketing", 2000),
    ("Saif", "Sales", 4100)
  ]
schema = ["employee_name", "department", "salary"]

df = spark.createDataFrame(data=simpleData, schema = schema)
df.printSchema()
df.show(truncate=False)

print("approx_count_distinct: " + \
      str(df.select(approx_count_distinct("salary")).collect()[0][0]))

print("avg: " + str(df.select(avg("salary")).collect()[0][0]))

df.select(collect_list("salary")).show(truncate=False)

df.select(collect_set("salary")).show(truncate=False)

df2 = df.select(countDistinct("department", "salary"))
df2.show(truncate=False)
print("Distinct Count of Department & Salary: "+str(df2.collect()[0][0]))

print("count: "+str(df.select(count("salary")).collect()[0]))
```

```
df.select(first("salary")).show(truncate=False)
df.select(last("salary")).show(truncate=False)
df.select(kurtosis("salary")).show(truncate=False)
df.select(max("salary")).show(truncate=False)
df.select(min("salary")).show(truncate=False)
df.select(mean("salary")).show(truncate=False)
df.select(skewness("salary")).show(truncate=False)
df.select(stddev("salary"), stddev_samp("salary"), \
    stddev_pop("salary")).show(truncate=False)
df.select(sum("salary")).show(truncate=False)
df.select(sumDistinct("salary")).show(truncate=False)
df.select(variance("salary"),var_samp("salary"),var_pop("salary")) \
  .show(truncate=False)
```

16. PySpark SQL Date and Timestamp Functions

PYSPARK DATE FUNCTION	DATE FUNCTION DESCRIPTION
current_date()	Returns the current date as a date column.
date_format(dateExpr,format)	Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument.
to_date()	Converts the column into `DateType` by casting rules to `DateType`.
to_date(column, fmt)	Converts the column into a `DateType` with a specified format
add_months(Column, numMonths)	Returns the date that is `numMonths` after `startDate`.
date_add(column, days) date_sub(column, days)	Returns the date that is `days` days after `start`
datediff(end, start)	Returns the number of days from `start` to `end`.
months_between(end, start)	Returns number of months between dates `start` and `end`. A whole number is returned if both inputs have the same day of month or both are the last day of their respective months. Otherwise, the difference is calculated assuming 31 days per month.

PYSPARK DATE FUNCTION	DATE FUNCTION DESCRIPTION
months_between(end, start, roundOff)	Returns number of months between dates `end` and `start`. If `roundOff` is set to true, the result is rounded off to 8 digits; it is not rounded otherwise.
next_day(column, dayOfWeek)	Returns the first date which is later than the value of the `date` column that is on the specified day of the week. For example, `next_day('2015-07-27', "Sunday")` returns 2015-08-02 because that is the first Sunday after 2015-07-27.
trunc(column, format)	Returns date truncated to the unit specified by the format. For example, `trunc("2018-11-19 12:01:19", "year")` returns 2018-01-01 format: 'year', 'yyyy', 'yy' to truncate by year, 'month', 'mon', 'mm' to truncate by month
date_trunc(format, timestamp)	Returns timestamp truncated to the unit specified by the format. For example, `date_trunc("year", "2018-11-19 12:01:19")` returns 2018-01-01 00:00:00 format: 'year', 'yyyy', 'yy' to truncate by year, 'month', 'mon', 'mm' to truncate by month, 'day', 'dd' to truncate by day, Other options are: 'second', 'minute', 'hour', 'week', 'month', 'quarter'
year(column)	Extracts the year as an integer from a given date/timestamp/string
quarter(column)	Extracts the quarter as an integer from a given date/timestamp/string.
month(column)	Extracts the month as an integer from a given date/timestamp/string
dayofweek(column)	Extracts the day of the week as an integer from a given date/timestamp/string. Ranges from 1 for a Sunday through to 7 for a Saturday
dayofmonth(column)	Extracts the day of the month as an integer from a given date/timestamp/string.
dayofyear(column)	Extracts the day of the year as an integer from a given date/timestamp/string.

PYSPARK DATE FUNCTION	DATE FUNCTION DESCRIPTION
weekofyear(column)	Extracts the week number as an integer from a given date/timestamp/string. A week is considered to start on a Monday and week 1 is the first week with more than 3 days, as defined by ISO 8601
last_day(column)	Returns the last day of the month which the given date belongs to. For example, input "2015-07-27" returns "2015-07-31" since July 31 is the last day of the month in July 2015.
from_unixtime(column)	Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the yyyy-MM-dd HH:mm:ss format.
from_unixtime(column, f)	Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the given format.
unix_timestamp()	Returns the current Unix timestamp (in seconds) as a long
unix_timestamp(column)	Converts time string in format yyyy-MM-dd HH:mm:ss to Unix timestamp (in seconds), using the default timezone and the default locale.
unix_timestamp(column, p)	Converts time string with given pattern to Unix timestamp (in seconds).

PYSPARK TIMESTAMP FUNCTION SIGNATURE	TIMESTAMP FUNCTION DESCRIPTION
current_timestamp()	Returns the current timestamp as a timestamp column
hour(column)	Extracts the hours as an integer from a given date/timestamp/string.
minute(column)	Extracts the minutes as an integer from a given date/timestamp/string.
second(column)	Extracts the seconds as an integer from a given date/timestamp/string.

PYSPARK TIMESTAMP FUNCTION SIGNATURE	TIMESTAMP FUNCTION DESCRIPTION
to_timestamp(column)	Converts to a timestamp by casting rules to `TimestampType`.
to_timestamp(column, fmt)	Converts time string with the given pattern to timestamp.

17. PySpark Read CSV file into DataFrame

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType,StructField, StringType, IntegerType
from pyspark.sql.types import ArrayType, DoubleType, BooleanType
from pyspark.sql.functions import col,array_contains

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

df = spark.read.csv("/tmp/resources/zipcodes.csv")

df.printSchema()

df2 = spark.read.option("header",True) \
      .csv("/tmp/resources/zipcodes.csv")
df2.printSchema()

df3 = spark.read.options(header='True', delimiter=',') \
   .csv("/tmp/resources/zipcodes.csv")
df3.printSchema()

schema = StructType() \
       .add("RecordNumber",IntegerType(),True) \
       .add("Zipcode",IntegerType(),True) \
       .add("ZipCodeType",StringType(),True)

df_with_schema = spark.read.format("csv") \
       .option("header", True) \
       .schema(schema) \
       .load(/tmp/resources/zipcodes.csv")
df_with_schema.printSchema()

df2.write.option("header",True) \
 .csv("/tmp/spark_output/zipcodes123")
```

18. PySpark Read and Write Parquet File

What is Parquet File?

Apache Parquet file is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model, or programming language.

While querying columnar storage, it skips the nonrelevant data very quickly, making faster query execution. As a result aggregation queries consume less time compared to row-oriented databases.

It is able to support advanced nested data structures.

Parquet supports efficient compression options and encoding schemes.

Pyspark SQL provides support for both reading and writing Parquet files that automatically capture the schema of the original data, It also reduces data storage by 75% on average. Pyspark by default supports Parquet in its library hence we don't need to add any dependency libraries.

```
df=spark.createDataFrame(data,columns)
df.write.mode("overwrite").parquet("/tmp/output/people.parquet")
parDF1=spark.read.parquet("/tmp/output/people.parquet")
parDF1.createOrReplaceTempView("parquetTable")
parDF1.printSchema()
parDF1.show(truncate=False)

parkSQL = spark.sql("select * from ParquetTable where salary >= 4000 ")
parkSQL.show(truncate=False)

spark.sql("CREATE TEMPORARY VIEW PERSON USING parquet OPTIONS (path
\"/tmp/output/people.parquet\")")
spark.sql("SELECT * FROM PERSON").show()

df.write.partitionBy("gender","salary").mode("overwrite").parquet("/tmp/output/people2.parquet")

parDF2=spark.read.parquet("/tmp/output/people2.parquet/gender=M")
parDF2.show(truncate=False)

spark.sql("CREATE TEMPORARY VIEW PERSON2 USING parquet OPTIONS (path
\"/tmp/output/people2.parquet/gender=F\")")
spark.sql("SELECT * FROM PERSON2"  ).show()
```

19. PySpark Read JSON file into DataFrame

```
# Read JSON file into dataframe
df = spark.read.json("resources/zipcodes.json")
df.printSchema()
df.show()

# Read multiline json file
multiline_df = spark.read.option("multiline","true") \
      .json("resources/multiline-zipcode.json")
multiline_df.show()

#Read multiple files
df2 = spark.read.json(
```

```
      ['resources/zipcode2.json','resources/zipcode1.json'])
df2.show()

#Read All JSON files from a directory
df3 = spark.read.json("resources/*.json")
df3.show()

# Define custom schema
schema = StructType([
       StructField("RecordNumber",IntegerType(),True),
       StructField("Zipcode",IntegerType(),True),
       StructField("ZipCodeType",StringType(),True),
   ])

df_with_schema = spark.read.schema(schema) \
         .json("resources/zipcodes.json")
df_with_schema.printSchema()
df_with_schema.show()

# Create a table from Parquet File
spark.sql("CREATE OR REPLACE TEMPORARY VIEW zipcode3 USING json OPTIONS"  +
        " (path 'resources/zipcodes.json')")
spark.sql("select * from zipcode3").show()

# PySpark write Parquet File
df2.write.mode('Overwrite').json("/tmp/spark_output/zipcodes.json")
```

20. SQL Questions

Explain the various types of Joins?
Explain Equi join?
Explain the Right Outer Join?
How do you alter the name of a column?
how can you build a stored procedure?
Distinguish between MongoDB and MySQL?
Compare the 'Having' and 'Where' clauses in detail?
What are the differences between COALESCE() and ISNULL()?
What is the difference between “Stored Procedure” and “Function”?
What is the difference between the “DELETE” and “TRUNCATE” commands?
What is difference between “Clustered Index” and “Non Clustered Index”?
What is the difference between “Primary Key” and “Unique Key”?
What is the difference between a “Local Temporary Table” and “Global Temporary Table”?
What is the difference between primary key and unique constraints?
What are the differences between DDL, DML and DCL in SQL?
What is a view in SQL? How to create view?
What is a Trigger?
What is the difference between Trigger and Stored Procedure?
What are indexes?
What are Primary Keys and Foreign Keys?
What are wildcards used in database for Pattern Matching?

What is Union, minus and Interact commands?
What is RDBMS?
What is OLTP?
What is Aggregate Functions?
What is the difference between UNION and UNION ALL?
What is a foreign key, and what is it used for?

Scenario Based Questions--

- SQL Query to find second highest salary of Employee?
- SQL Query to find Max Salary from each department?
- Write SQL Query to display current date?
- Write an SQL Query to check whether date passed to Query is date of given format or not?
- Write a SQL Query to print the name of distinct employee whose DOB is between 01/01/1960 to 31/12/1975?
- Write an SQL Query to find employee whose Salary is equal or greater than 10000?
- Write an SQL Query to find name of employee whose name Start with 'M'?
- Find the 3rd MAX salary in the emp table?
- Suppose there is annual salary information provided by emp table.
- How to fetch monthly salary of each and every employee?
- Display the list of employees who have joined the company before 30th June 90 or after 31st dec 90?

21. Spark Questions

What is spark? Explain Architecture
Explain where did you use spark in your project?
What all optimization techniques have you used in spark?
Explain transformations and actions have you used?
What happens when you use shuffle in spark?
Difference between ReduceByKey Vs GroupByKey?
Explain the issues you resolved when you working with spark?
Compare Spark vs Hadoop MapReduce?
Difference between Narrow & wide transformations?
What is partition and how spark Partitions the data?
What is RDD?
what is broadcast variable?
Difference between Sparkcontext Vs Sparksession?
Explain about transformations and actions in the spark?
what is Executor memory in spark?
What is lineage graph?
What is DAG?
Explain libraries that Spark Ecosystem supports?
What is a DStream?
What is Catalyst optimizer and explain it?
Why parquet file format is best for spark?

Difference between dataframe Vs Dataset Vs RDD?
Explain features of Apache Spark?
Explain Lazy evaluation and why is it need?
Explain Pair RDD?
What is Spark Core?
What is the difference between persist() and cache()?
What are the various levels of persistence in Apache Spark?
Does Apache Spark provide check pointing?
How can you achieve high availability in Apache Spark?
Explain Executor Memory in a Spark?
What are the disadvantages of using Apache Spark?
What is the default level of parallelism in apache spark?
Compare map() and flatMap() in Spark?
Difference between repartition Vs coalesce?
Explain Spark Streaming?
Explain accumulators?
What is the use of broadcast join?

Apache Spark:
➞ It is general purpose
➞ in memory
➞ compute engine

❗ Compute Engine:
what does hadoop provides?
hadoop provides 3 things:
hdfs = storage
MapReduce = computation
YARN = Resource manager.

-- Spark is an replacement/alternative of Mapreduce.
-- it is not good to compare spark with Hadoop, but we can compare spark with mapreduce.
-- spark is a plug an play compute engine which requires 2 things.
1. storage - local storage, Hdfs, Amazon s3
2. Resource manager - Yarn, Mesos, Kubernetes
-- spark is not bounded for particular storage or resource manager.

❗ In Memory:
-- for each MapReduce job HDFS required 2-disc access i.e. onetime for reading and one time for writing.
-- but in Spark only one Io's disc is required which is initial read and final write.
-- spark is said to be 10 to 100 times faster than MapReduce.

General Purpose:
❗ in hadoop we use Pig for cleaning.
❗ hive for querying.
❗ for machine learning mahout
❗ sqoop for database streaming data.
❗ in mapreduce we only bound to use map and reduce.
❗ but in spark everything is possible. like ❗ whatever discussed above.
❗ to achieve we just need to learn one style of code.

- these are the reasons Spark is most preferred choice.
- spark also provides filter too.
- And this is called as General purpose compute.
- The basic unit which holds the data in spark is called as RDD (Resilient Distributed Dataset).

- In spark there are 2 kinds of operations.

1. Transformations.
2. Actions.

22. PySpark Questions

- What is PySpark Architecture?
- What's the difference between an RDD, a DataFrame & DataSet?
- How can you create a DataFrame a) using existing RDD, and b) from a CSV file?
- Explain the use of StructType and StructField classes in PySpark with examples?
- What are the different ways to handle row duplication in a PySpark DataFrame?
- Explain PySpark UDF with the help of an example?
- Discuss the map() transformation in PySpark DataFrame
- what do you mean by 'joins' in PySpark DataFrame? What are the different types of joins?
- What is PySpark ArrayType?
- What is PySpark Partition?
- What is meant by PySpark MapType? How can you create a MapType using StructType?
- How can PySpark DataFrame be converted to Pandas DataFrame?
- What is the function of PySpark's pivot() method?
- In PySpark, how do you generate broadcast variables?
- When to use Client and Cluster modes used for deployment?
- How can data transfers be kept to a minimum while using PySpark?
- What are Sparse Vectors? What distinguishes them from dense vectors?
- What API does PySpark utilize to implement graphs?
- What is meant by Piping in PySpark?
- What are the various levels of persistence that exist in PySpark?
- List some of the benefits of using PySpark?
- Why do we use PySpark SparkFiles?
- Does PySpark provide a machine learning API?
- What are the types of PySpark's shared variables and why are they useful?
- What PySpark DAGScheduler?

23. PySpark Collect() – Retrieve data from DataFrame

PySpark RDD/DataFrame `collect()` is an action operation that is used to retrieve all the elements of the dataset (from all nodes) to the driver node. We should use the collect() on smaller dataset usually after filter(), group() e.t.c. Retrieving larger datasets results in `OutOfMemory` error.

When to avoid Collect()

Usually, collect() is used to retrieve the action output when you have very small result set and calling `collect()` on an RDD/DataFrame with a bigger result set causes out of memory as it returns the entire dataset (from all workers) to the driver hence we should avoid calling collect() on a larger dataset.

collect () vs select ()

`select()` is a transformation that returns a new DataFrame and holds the columns that are selected whereas collect() is an action that returns the entire data set in an Array to the driver.

24. PySpark Groupby Explained with Example

Similar to SQL `GROUP BY` clause, PySpark `groupBy()` function is used to collect the identical data into groups on DataFrame and perform aggregate functions on the grouped data. In this article, I will explain several `groupBy()` examples using PySpark (Spark with Python).

When we perform `groupBy()` on PySpark Dataframe, it returns `GroupedData` object which contains below aggregate functions.

`count()` – Returns the count of rows for each group.

`mean()` – Returns the mean of values for each group.

`max()` – Returns the maximum of values for each group.

`min()` – Returns the minimum of values for each group.

`sum()` – Returns the total for values for each group.

`avg()` – Returns the average for values for each group.

`agg()` – Using agg() function, we can calculate more than one aggregate at a time.

Using filter on aggregate data

Similar to SQL "HAVING" clause, On PySpark DataFrame we can use either where() or filter() function to filter the rows of aggregated data.

```
schema = ["employee_name","department","state","salary","age","bonus"]
df.groupBy("department").sum("salary").show(truncate=False)

df.groupBy("department").count().show(truncate=False)


df.groupBy("department","state") \
    .sum("salary","bonus") \
   .show(truncate=False)

df.groupBy("department") \
    .agg(sum("salary").alias("sum_salary"), \
```

```
        avg("salary").alias("avg_salary"), \
        sum("bonus").alias("sum_bonus"), \
        max("bonus").alias("max_bonus") \
     ) \
    .show(truncate=False)

df.groupBy("department") \
    .agg(sum("salary").alias("sum_salary"), \
      avg("salary").alias("avg_salary"), \
      sum("bonus").alias("sum_bonus"), \
      max("bonus").alias("max_bonus")) \
    .where(col("sum_bonus") >= 50000) \
    .show(truncate=False)
```

25. PySpark Join Types | Join Two DataFrames

PySpark Join is used to combine two DataFrames and by chaining these you can join multiple DataFrames; it supports all basic join type operations available in traditional SQL like `INNER`, `LEFT OUTER`, `RIGHT OUTER`, `LEFT ANTI`, `LEFT SEMI`, `CROSS`, `SELF` JOIN. PySpark Joins are wider transformations that involve data shuffling across the network.

PySpark SQL Joins comes with more optimization by default (thanks to DataFrames) however still there would be some performance issues to consider while using.

`join()` operation takes parameters as below and returns DataFrame.

param other: Right side of the join

param on: a string for the join column name

param how: default `inner`. Must be one of `inner`, `cross`, `outer`,`full`, `full_outer`, `left`, `left_outer`, `right`, `right_outer`,`left_semi`, and `left_anti`.

Join String	Equivalent SQL Join
inner	INNER JOIN
outer, full, fullouter, full_outer	FULL OUTER JOIN
left, leftouter, left_outer	LEFT JOIN
right, rightouter, right_outer	RIGHT JOIN

```
empColumns = ["emp_id","name","superior_emp_id","year_joined", \
       "emp_dept_id","gender","salary"]
deptColumns = ["dept_name","dept_id"]
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"inner") \
     .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"outer") \
    .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"full") \
    .show(truncate=False)
```

```
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"fullouter") \
    .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"left") \
    .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"leftouter") \
   .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"right") \
   .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"rightouter") \
   .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"leftsemi") \
   .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"leftanti") \
   .show(truncate=False)

empDF.alias("emp1").join(empDF.alias("emp2"), \
    col("emp1.superior_emp_id") == col("emp2.emp_id"),"inner") \
    .select(col("emp1.emp_id"),col("emp1.name"), \
      col("emp2.emp_id").alias("superior_emp_id"), \
      col("emp2.name").alias("superior_emp_name")) \
   .show(truncate=False)

empDF.createOrReplaceTempView("EMP")
deptDF.createOrReplaceTempView("DEPT")

joinDF = spark.sql("select * from EMP e, DEPT d where e.emp_dept_id == d.dept_id") \
  .show(truncate=False)

joinDF2 = spark.sql("select * from EMP e INNER JOIN DEPT d ON e.emp_dept_id == d.dept_id")
\
  .show(truncate=False)
```

26. PySpark Union and UnionAll Explained

PySpark union() and unionAll() transformations are used to merge two or more DataFrame's of the same schema or structure. In this PySpark article, I will explain both union transformations with PySpark examples.

Dataframe union() – `union()` method of the DataFrame is used to merge two DataFrame's of the same structure/schema. If schemas are not the same it returns an error.

DataFrame unionAll() – `unionAll()` is deprecated since Spark "2.0.0" version and replaced with union().

Note: In other SQL languages, Union eliminates the duplicates but UnionAll merges two datasets including duplicate records. But, in PySpark both behave the same and recommend using DataFrame duplicate() function to remove duplicate rows.

```
columns= ["employee_name","department","state","salary","age","bonus"]
columns2= ["employee_name","department","state","salary","age","bonus"]
unionDF = df.union(df2)
```

```
unionDF.show(truncate=False)
disDF = df.union(df2).distinct()
disDF.show(truncate=False)
```

27. PySpark map() Transformation

PySpark map (`map()`) is an RDD transformation that is used to apply the transformation function (lambda) on every element of RDD/DataFrame and returns a new RDD. In this article, you will learn the syntax and usage of the RDD map() transformation with an example and how to use it with DataFrame.

RDD `map()` transformation is used to apply any complex operations like adding a column, updating a column, transforming the data e.t.c, the output of map transformations would always have the same number of records as input.

Note1: DataFrame doesn't have map() transformation to use with DataFrame hence you need to DataFrame to RDD first.

Note2: If you have a heavy initialization use PySpark mapPartitions() transformation instead of map(), as with mapPartitions() heavy initialization executes only once for each partition instead of every record.

```
rdd=spark.sparkContext.parallelize(data)

rdd2=rdd.map(lambda x: (x,1))
for element in rdd2.collect():
    print(element)
rdd2=df.rdd.map(lambda x:
    (x[0]+","+x[1],x[2],x[3]*2)
    )
df2=rdd2.toDF(["name","gender","new_salary"]   )
df2.show()

#Referring Column Names
rdd2=df.rdd.map(lambda x:
    (x["firstname"]+","+x["lastname"],x["gender"],x["salary"]*2)
    )

#Referring Column Names
rdd2=df.rdd.map(lambda x:
    (x.firstname+","+x.lastname,x.gender,x.salary*2)
    )

def func1(x):
    firstName=x.firstname
rdd2=df.rdd.map(lambda x: func1(x))
```

28. PySpark partitionBy() – Write to Disk Example

PySpark `partitionBy()` is a function of `pyspark.sql.DataFrameWriter` class which is used to partition the large dataset (DataFrame) into smaller files based on one or multiple columns while writing to disk, let's see how to use this with Python examples.

Partitioning the data on the file system is a way to improve the performance of the query when dealing with a large dataset in the Data lake. A Data Lake is a centralized repository of structured, semi-structured, unstructured, and binary data that allows you to store a large amount of data as-is in its original raw format.

1. What is PySpark Partition?

PySpark partition is a way to split a large dataset into smaller datasets based on one or more partition keys. When you create a DataFrame from a file/table, based on certain parameters PySpark creates the DataFrame with a certain number of partitions in memory. This is one of the main advantages of PySpark DataFrame over Pandas DataFrame. Transformations on partitioned data run faster as they execute transformations parallelly for each partition.

PySpark supports partition in two ways; partition in memory (DataFrame) and partition on the disk (File system).

Partition in memory: You can partition or repartition the DataFrame by calling repartition() or coalesce() transformations.

Partition on disk: While writing the PySpark DataFrame back to disk, you can choose how to partition the data based on columns using partitionBy() of pyspark.sql.DataFrameWriter. This is similar to Hives partitions scheme.

2. Partition Advantages

As you are aware PySpark is designed to process large datasets with 100x faster than the tradition processing, this wouldn't have been possible with out partition. Below are some of the advantages using PySpark partitions on memory or on disk.

Fast accessed to the data
Provides the ability to perform an operation on a smaller dataset
Partition at rest (disk) is a feature of many databases and data processing frameworks and it is key to make jobs work at scale.

How to Choose a Partition Column When Writing to File system?

When creating partitions you have to be very cautious with the number of partitions you would create, as having too many partitions creates too many sub-directories on HDFS which brings unnecessarily and overhead to NameNode (if you are using Hadoop) since it must keep all metadata for the file system in memory.

Let's assume you have a US census table that contains zip code, city, state, and other columns. Creating a partition on the state, splits the table into around 50 partitions, when searching for a zipcode within a state (state='CA' and zipCode ='92704') results in faster as it needs to scan only in a **state=CA** partition directory.

Partition on zipcode may not be a good option as you might end up with too many partitions.

Another good example of partition is on the Date column. Ideally, you should partition on Year/Month but not on a date.

```
df.write.option("header",True) \
        .partitionBy("state") \
        .mode("overwrite") \
        .csv("/tmp/zipcodes-state")
#partitionBy() multiple columns
df.write.option("header",True) \
        .partitionBy("state","city") \
        .mode("overwrite") \
        .csv("/tmp/zipcodes-state")
```

Using repartition() and partitionBy() together

For each partition column, if you wanted to further divide into several partitions, use `repartition()` and `partitionBy()` together as explained in the below example.

repartition() creates specified number of partitions in memory. The `partitionBy()` will write files to disk for each memory partition and partition column.

```
#Use repartition() and partitionBy() together
dfRepart.repartition(2)
        .write.option("header",True) \
        .partitionBy("state") \
        .mode("overwrite") \
        .csv("c:/tmp/zipcodes-state-more")
```

Read a Specific Partition

Reads are much faster on partitioned data. This code snippet retrieves the data from a specific partition `"state=AL and city=SPRINGVILLE"`. Here, It just reads the data from that specific folder instead of scanning a whole file (when not partitioned).

```
dfSinglePart=spark.read.option("header",True) \
            .csv("c:/tmp/zipcodes-state/state=AL/city=SPRINGVILLE")
```

How to Choose a Partition Column When Writing to File system?

When creating partitions you have to be very cautious with the number of partitions you would create, as having too many partitions creates too many sub-directories on HDFS which brings unnecessarily and overhead to NameNode (if you are using Hadoop) since it must keep all metadata for the file system in memory.

Let's assume you have a US census table that contains zip code, city, state, and other columns. Creating a partition on the state, splits the table into around 50 partitions, when searching for a zipcode within a state (state='CA' and zipCode ='92704') results in faster as it needs to scan only in a state=CA partition directory.

Partition on zipcode may not be a good option as you might end up with too many partitions.

Another good example of partition is on the Date column. Ideally, you should partition on Year/Month but not on a date.

29. Hive Table Types

29.1. Internal or Managed Table

By default, Hive creates an Internal table also known as the Managed table, In the managed table, Hive owns the data/files on the table meaning any data you insert or load files to the table are managed by the Hive process when you drop the table the underlying data or files are also get deleted.

29.2. External Table

Using `EXTERNAL` option you can create an external table, Hive doesn't manage the external table, when you drop an external table, only table metadata from Metastore will be removed but the underlying files will not be removed and still they can be accessed via HDFS commands, Pig, Spark or any other Hadoop compatible tools.

Let's see this in action by dropping the table **emp.employee_external** using `DROP TABLE emp.employee_external` command and check if the file still exists by running above `hdfs -ls` command.

29.3. Temporary Table

A temporary table is created using `TEMPORARY` option, these tables exist only within the current session, upon exiting the session the temporary tables will be removed and cannot be accessed in another session.

There are few limitations to the temporary table

- Cannot Create Partitioned Table
- Indexes are not supported

29.4. Transactional Table

Hive 4.0 supports another type of table called Transactional tables., Transactional Tables have support ACID operations like Insert, Update and Delete operations.

30. What is a cluster?

A Databricks cluster is a set of computation resources that performs the heavy lifting of all of the data workloads you run in Databricks. These workloads can be run as commands in notebooks, commands run from BI tools that are connected to Databricks, or automated jobs that you've scheduled. Clusters perform the processing of these workloads and then return results or save them out to data stores.

A cluster consists of multiple nodes (individual machines) that operate on your workloads in parallel. There is one driver node for every cluster, which is the one that delegates tasks and oversees the execution of your specific workload. There are also many worker nodes for every cluster that perform the processing. If a worker node in a Databricks cluster is lost for any reason, the driver can reallocate remaining work to the remaining nodes.

At the left side are two columns indicating if the cluster has been pinned and the status of the cluster:

- Pinned
- Starting , Terminating

Standard cluster

- Running
- Terminated

High concurrency cluster

- Running
- Terminated

Access Denied

- Running
- Terminated

Table ACLs enabled

- Running
- Terminated

30.1. Cluster mode

Databricks supports three cluster modes: Standard, High Concurrency, and Single Node. Most regular users use Standard or Single Node clusters.

- Standard clusters are ideal for processing large amounts of data with Apache Spark.
- Single Node clusters are intended for jobs that use small amounts of data or non-distributed workloads such as single-node machine learning libraries.
- High Concurrency clusters are ideal for groups of users who need to share resources or run ad-hoc jobs. Administrators usually create High Concurrency clusters. Databricks recommends enabling autoscaling for High Concurrency clusters.

Autoscaling allows clusters to resize automatically based on workloads.

Pools reduce cluster start and scale-up times by maintaining a set of available, ready-to-use instances.

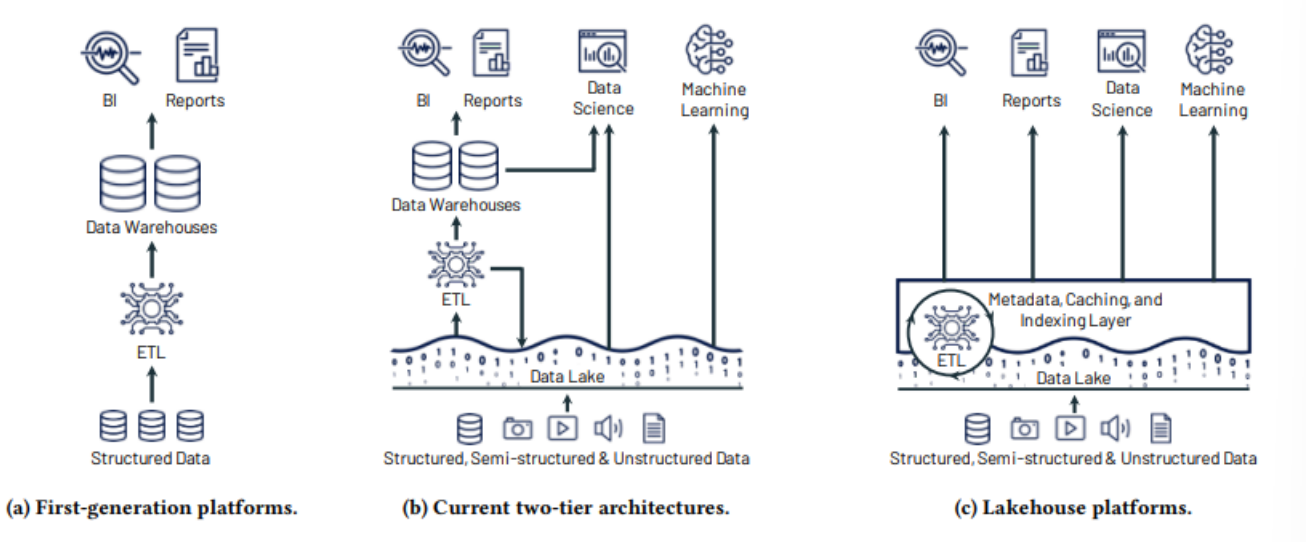
Databricks cluster policies allow administrators to enforce controls over the creation and configuration of clusters.

31. Delta Lake

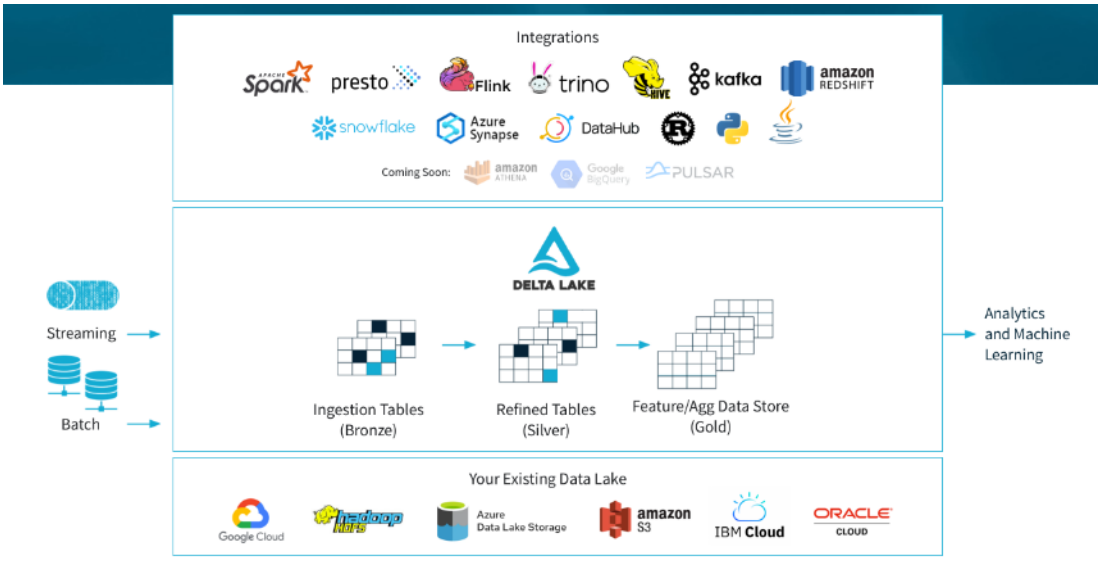
Delta Lake is an open source project that enables building a Lakehouse architecture on top of data lakes. Delta Lake provides ACID transactions, scalable metadata handling, and unifies streaming and batch data processing on top of existing data lakes, such as S3, ADLS, GCS, and HDFS.

Specifically, Delta Lake offers:

- ACID transactions on Spark: Serializable isolation levels ensure that readers never see inconsistent data.
- Scalable metadata handling: Leverages Spark distributed processing power to handle all the metadata for petabyte-scale tables with billions of files at ease.
- Streaming and batch unification: A table in Delta Lake is a batch table as well as a streaming source and sink. Streaming data ingest, batch historic backfill, interactive queries all just work out of the box.
- Schema enforcement: Automatically handles schema variations to prevent insertion of bad records during ingestion.
- Time travel: Data versioning enables rollbacks, full historical audit trails, and reproducible machine learning experiments.
- Upserts and deletes: Supports merge, update and delete operations to enable complex use cases like change-data-capture, slowly-changing-dimension (SCD) operations, streaming upserts, and so on.



(a) First-generation platforms. (b) Current two-tier architectures. (c) Lakehouse platforms.



31.1. Create a table

To create a Delta table, write a DataFrame out in the `delta` format. You can use existing Spark SQL code and change the format from `parquet`, `csv`, `json`, and so on, to `delta`.

```
data = spark.range(0, 5)
data.write.format("delta").save("/tmp/delta-table")
```

31.2. Read data

You read data in your Delta table by specifying the path to the files: "`/tmp/delta-table`":

```
df = spark.read.format("delta").load("/tmp/delta-table")
df.show()
```

31.3. Update table data

Delta Lake supports several operations to modify tables using standard DataFrame APIs. This example runs a batch job to overwrite the data in the table:

```
data = spark.range(5, 10)
data.write.format("delta").mode("overwrite").save("/tmp/delta-table")
deltaTable = DeltaTable.forPath(spark, "/tmp/delta-table")

# Update every even value by adding 100 to it
deltaTable.update(
  condition = expr("id % 2 == 0"),
  set = { "id": expr("id + 100") })

# Delete every even value
deltaTable.delete(condition = expr("id % 2 == 0"))

# Upsert (merge) new data
newData = spark.range(0, 20)

deltaTable.alias("oldData") \
  .merge(
    newData.alias("newData"),
    "oldData.id = newData.id") \
  .whenMatchedUpdate(set = { "id": col("newData.id") }) \
  .whenNotMatchedInsert(values = { "id": col("newData.id") }) \
  .execute()

deltaTable.toDF().show()
```

31.4. Write a stream of data to a table

You can also write to a Delta table using Structured Streaming. The Delta Lake transaction log guarantees exactly-once processing, even when there are other streams or batch queries running concurrently against the table. By default, streams run in append mode, which adds new records to the table:

```
streamingDf = spark.readStream.format("rate").load()
stream = streamingDf.selectExpr("value as
id").writeStream.format("delta").option("checkpointLocation",
"/tmp/checkpoint").start("/tmp/delta-table")
```

You can stop the stream by running `stream.stop()` in the same terminal that started the stream.

Delta Lake supports most of the options provided by Apache Spark DataFrame read and write APIs for performing batch reads and writes on tables.

31.5. Create a table

Delta Lake supports creating two types of tables—tables defined in the metastore and tables defined by path.

To work with metastore-defined tables, you must enable integration with Apache Spark DataSourceV2 and Catalog APIs by setting configurations when you create a new `SparkSession`. See Configure SparkSession.

You can create tables in the following ways.

- **SQL DDL commands**: You can use standard SQL DDL commands supported in Apache Spark (for example, `CREATE TABLE` and `REPLACE TABLE`) to create Delta tables.

```
CREATE TABLE IF NOT EXISTS default.people10m (
  id INT,
  firstName STRING,
  birthDate TIMESTAMP,
  ssn STRING,
  salary INT
) USING DELTA

CREATE OR REPLACE TABLE default.people10m (
  id INT,
  firstName STRING,
  birthDate TIMESTAMP,
  ssn STRING,
  salary INT
) USING DELTA
```

SQL also supports creating a table at a path, without creating an entry in the Hive metastore.

```
-- Create or replace table with path
CREATE OR REPLACE TABLE delta.`/tmp/delta/people10m` (
  id INT,
  firstName STRING,
  birthDate TIMESTAMP,
  ssn STRING,
  salary INT
) USING DELTA
```

- **`DataFrameWriter` API**: If you want to simultaneously create a table and insert data into it from Spark DataFrames or Datasets, you can use the Spark `DataFrameWriter` (Scala or Java and Python).

```
# Create table in the metastore using DataFrame's schema and write data to it
df.write.format("delta").saveAsTable("default.people10m")

# Create or replace partitioned table with path using DataFrame's schema and write/overwrite data to it
df.write.format("delta").mode("overwrite").save("/tmp/delta/people10m")
```

You can also create Delta tables using the Spark `DataFrameWriterV2` API.

- **`DeltaTableBuilder` API**: You can also use the `DeltaTableBuilder` API in Delta Lake to create tables. Compared to the DataFrameWriter APIs, this API makes it easier to specify additional information like column comments, table properties, and generated columns.

This feature is new and is in Preview.

```
# Create table in the metastore
DeltaTable.createIfNotExists(spark) \
  .tableName("default.people10m") \
  .addColumn("id", "INT") \
  .addColumn("firstName", "STRING") \
  .addColumn("middleName", "STRING") \
  .addColumn("lastName", "STRING", comment = "surname") \
  .addColumn("gender", "STRING") \
  .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("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()
```

31.5.1. Partition data

You can partition data to speed up queries or DML that have predicates involving the partition columns. To partition data when you create a Delta table, specify a partition by columns. The following example partitions by gender.

```
-- Create table in the metastore
CREATE TABLE default.people10m (
  id INT,
  firstName STRING,
  birthDate TIMESTAMP,
  ssn STRING,
  salary INT
)
USING DELTA
PARTITIONED BY (gender)
```

To determine whether a table contains a specific partition, use the statement

`SELECT COUNT(*) > 0 FROM <table-name> WHERE <partition-column> = <value>`. If the partition exists, `true` is returned. For example:

```
SELECT COUNT(*) > 0 AS `Partition exists` FROM default.people10m WHERE gender = "M"
```

31.5.2. Control data location

For tables defined in the metastore, you can optionally specify the `LOCATION` as a path. Tables created with a specified `LOCATION` are considered unmanaged by the metastore. Unlike a managed table, where no path is specified, an unmanaged table's files are not deleted when you `DROP` the table.

When you run `CREATE TABLE` with a `LOCATION` that *already* contains data stored using Delta Lake, Delta Lake does the following:

- If you specify *only the table name and location*, for example:

```
CREATE TABLE default.people10m
USING DELTA
LOCATION '/tmp/delta/people10m'
```

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

- If you specify *any configuration* (schema, partitioning, or table properties), Delta Lake verifies that the specification exactly matches the configuration of the existing data.

Important: If the specified configuration does not *exactly* match the configuration of the data, Delta Lake throws an exception that describes the discrepancy.

Note: The metastore is not the source of truth about the latest information of a Delta table. In fact, the table definition in the metastore may not contain all the metadata like schema and properties. It contains the location of the table, and the table's transaction log at the location is the source of truth. If you query the metastore from a system that is not aware of this Delta-specific customization, you may see incomplete or stale table information.

31.5.3. Use generated columns

Note: This feature is new and is in Preview.

Delta Lake supports generated columns which are a special type of columns whose values are automatically generated based on a user-specified function over other columns in the Delta table. When you write to a table with generated columns and you do not explicitly provide values for them, Delta Lake automatically computes the values. For example, you can automatically generate a date column (for partitioning the table by date) from the timestamp column; any writes into the table need only specify the data for the timestamp column. However, if you explicitly provide values for them, the values must satisfy the constraint `(<value> <=> <generation expression>) IS TRUE` or the write will fail with an error.

Important

Tables created with generated columns have a higher table writer protocol version than the default. See Table protocol versioning to understand table protocol versioning and what it means to have a higher version of a table protocol version.

The following example shows how to create a table with generated columns:

```
DeltaTable.create(spark) \
  .tableName("default.people10m") \
  .addColumn("id", "INT") \
  .addColumn("firstName", "STRING") \
  .addColumn("middleName", "STRING") \
  .addColumn("lastName", "STRING", comment = "surname") \
  .addColumn("gender", "STRING") \
  .addColumn("birthDate", "TIMESTAMP") \
  .addColumn("dateOfBirth", DateType(), generatedAlwaysAs="CAST(birthDate AS DATE)") \
  .addColumn("ssn", "STRING") \
  .addColumn("salary", "INT") \
```

```
  .partitionedBy("gender") \
  .execute()
```
Generated columns are stored as if they were normal columns. That is, they occupy storage.

The following restrictions apply to generated columns:

- A generation expression can use any SQL functions in Spark that always return the same result when given the same argument values, except the following types of functions:
    - User-defined functions.
    - Aggregate functions.
    - Window functions.
    - Functions returning multiple rows.
- For Delta Lake 1.1.0 and above, `MERGE` operations support generated columns when you set `spark.databricks.delta.schema.autoMerge.enabled` to true.

Delta Lake may be able to generate partition filters for a query whenever a partition column is defined by one of the following expressions:

- `CAST(col AS DATE)` and the type of `col` is `TIMESTAMP`.
- `YEAR(col)` and the type of `col` is `TIMESTAMP`.
- Two partition columns defined by `YEAR(col), MONTH(col)` and the type of `col` is `TIMESTAMP`.
- Three partition columns defined by `YEAR(col), MONTH(col), DAY(col)` and the type of `col` is `TIMESTAMP`.
- Four partition columns defined by `YEAR(col), MONTH(col), DAY(col), HOUR(col)` and the type of `col` is `TIMESTAMP`.
- `SUBSTRING(col, pos, len)` and the type of `col` is `STRING`
- `DATE_FORMAT(col, format)` and the type of `col` is `TIMESTAMP`.

If a partition column is defined by one of the preceding expressions, and a query filters data using the underlying base column of a generation expression, Delta Lake looks at the relationship between the base column and the generated column, and populates partition filters based on the generated partition column if possible. For example, given the following table:

```
DeltaTable.create(spark) \
  .tableName("default.events") \
  .addColumn("eventId", "BIGINT") \
  .addColumn("data", "STRING") \
  .addColumn("eventType", "STRING") \
  .addColumn("eventTime", "TIMESTAMP") \
  .addColumn("eventDate", "DATE", generatedAlwaysAs="CAST(eventTime AS DATE)") \
  .partitionedBy("eventType", "eventDate") \
  .execute()
```

If you then run the following query:

```
spark.sql('SELECT * FROM default.events WHERE eventTime >= "2020-10-01 00:00:00" <= "2020-10-01 12:00:00"')
```

Delta Lake automatically generates a partition filter so that the preceding query only reads the data in partition `date=2020-10-01` even if a partition filter is not specified.

As another example, given the following table:

```
DeltaTable.create(spark) \
  .tableName("default.events") \
  .addColumn("eventId", "BIGINT") \
  .addColumn("data", "STRING") \
  .addColumn("eventType", "STRING") \
  .addColumn("eventTime", "TIMESTAMP") \
  .addColumn("year", "INT", generatedAlwaysAs="YEAR(eventTime)") \
  .addColumn("month", "INT", generatedAlwaysAs="MONTH(eventTime)") \
  .addColumn("day", "INT", generatedAlwaysAs="DAY(eventTime)") \
  .partitionedBy("eventType", "year", "month", "day") \
  .execute()
```

If you then run the following query:

```
spark.sql('SELECT * FROM default.events WHERE eventTime >= "2020-10-01 00:00:00" <= "2020-10-01 12:00:00"')
```

Delta Lake automatically generates a partition filter so that the preceding query only reads the data in partition `year=2020/month=10/day=01` even if a partition filter is not specified.

You can use an EXPLAIN clause and check the provided plan to see whether Delta Lake automatically generates any partition filters.

31.5.4. Use special characters in column names

By default, special characters such as spaces and any of the characters `,;{}()\n\t=` are not supported in table column names. To include these special characters in a table's column name, enable column mapping.

31.6. Read a table

You can load a Delta table as a DataFrame by specifying a table name or a path:

```
SELECT * FROM default.people10m   -- query table in the metastore

SELECT * FROM delta.`/tmp/delta/people10m`  -- query table by path
```

The DataFrame returned automatically reads the most recent snapshot of the table for any query; you never need to run `REFRESH TABLE`. Delta Lake automatically uses partitioning and statistics to read the minimum amount of data when there are applicable predicates in the query.

31.7. Query an older snapshot of a table (time travel)

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

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

31.7.1. Syntax

This section shows how to query an older version of a Delta table.

DataFrameReader options

DataFrameReader options allow you to create a DataFrame from a Delta table that is fixed to a specific version of the table.

```
df1 = spark.read.format("delta").option("timestampAsOf",
timestamp_string).load("/tmp/delta/people10m")
df2 = spark.read.format("delta").option("versionAsOf", version).load("/tmp/delta/people10m")
```

For `timestamp_string`, only date or timestamp strings are accepted. For example, `"2019-01-01"` and `"2019-01-01T00:00:00.000Z"`.

A common pattern is to use the latest state of the Delta table throughout the execution of a job to update downstream applications.

Because Delta tables auto update, a DataFrame loaded from a Delta table may return different results across invocations if the underlying data is updated. By using time travel, you can fix the data returned by the DataFrame across invocations:

```
history = spark.sql("DESCRIBE HISTORY delta.`/tmp/delta/people10m`")
latest_version = history.selectExpr("max(version)").collect()
df = spark.read.format("delta").option("versionAsOf",
latest_version[0][0]).load("/tmp/delta/people10m")
```

31.7.2. Examples

- Fix accidental deletes to a table for the user `111`:

```
yesterday = spark.sql("SELECT CAST(date_sub(current_date(), 1) AS STRING)").collect()[0][0]
df = spark.read.format("delta").option("timestampAsOf", yesterday).load("/tmp/delta/events")
df.where("userId = 111").write.format("delta").mode("append").save("/tmp/delta/events")
```

- Fix accidental incorrect updates to a table:

```
yesterday = spark.sql("SELECT CAST(date_sub(current_date(), 1) AS STRING)").collect()[0][0]
df = spark.read.format("delta").option("timestampAsOf", yesterday).load("/tmp/delta/events")
df.createOrReplaceTempView("my_table_yesterday")
spark.sql('''
MERGE INTO delta.`/tmp/delta/events` target
  USING my_table_yesterday source
  ON source.userId = target.userId
  WHEN MATCHED THEN UPDATE SET *
''')
```

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

```
last_week = spark.sql("SELECT CAST(date_sub(current_date(), 7) AS STRING)").collect()[0][0]
df = spark.read.format("delta").option("timestampAsOf", last_week).load("/tmp/delta/events")
last_week_count = df.select("userId").distinct().count()
count =
spark.read.format("delta").load("/tmp/delta/events").select("userId").distinct().count()
new_customers_count = count - last_week_count
```

31.7.3. Data retention

To time travel to a previous version, you must retain *both* the log and the data files for that version.

The data files backing a Delta table are *never* deleted automatically; data files are deleted only when you run VACUUM. `VACUUM` *does not* delete Delta log files; log files are automatically cleaned up after checkpoints are written.

By default you can time travel to a Delta table up to 30 days old unless you have:

- Run `VACUUM` on your Delta table.
- Changed the data or log file retention periods using the following table properties:
  - `delta.logRetentionDuration = "interval <interval>"`: controls how long the history for a table is kept. The default is `interval 30 days`.
  - Each time a checkpoint is written, Delta automatically cleans up log entries older than the retention interval. If you set this config to a large enough value, many log entries are retained. This should not impact performance as operations against the log are constant time. Operations on history are parallel but will become more expensive as the log size increases.
  - `delta.deletedFileRetentionDuration = "interval <interval>"`: controls how long ago a file must have been deleted *before being a candidate for* `VACUUM`. The default is `interval 7 days`.
  - To access 30 days of historical data even if you run `VACUUM` on the Delta table, set `delta.deletedFileRetentionDuration = "interval 30 days"`. This setting may cause your storage costs to go up.

Note

Due to log entry cleanup, instances can arise where you cannot time travel to a version that is less than the retention interval. Delta Lake requires all consecutive log entries since the previous checkpoint to time travel to a particular version. For example, with a table initially consisting of log entries for versions [0, 19] and a checkpoint at verison 10, if the log entry for version 0 is cleaned up, then you cannot time travel to versions [1, 9]. Increasing the table property `delta.logRetentionDuration` can help avoid these situations.

31.8. Write to a table

31.8.1. Append

To atomically add new data to an existing Delta table, use `append` mode:

```
INSERT INTO default.people10m SELECT * FROM morePeople
```

31.8.2. Overwrite

To atomically replace all the data in a table, use `overwrite` mode:

```
INSERT OVERWRITE TABLE default.people10m SELECT * FROM morePeople
```

Using DataFrames, you can also selectively overwrite only the data that matches an arbitrary expression. This feature is available in Delta Lake 1.1.0 and above. The following command atomically replaces events in January in the target table, which is partitioned by `start_date`, with the data in `df`:

```
df.write \
  .format("delta") \
  .mode("overwrite") \
  .option("replaceWhere", "start_date >= '2017-01-01' AND end_date <= '2017-01-31'") \
  .save("/tmp/delta/events")
```

This sample code writes out the data in `df`, validates that it all matches the predicate, and performs an atomic replacement. If you want to write out data that doesn't all match the predicate, to replace the matching rows in the target table, you can disable the constraint check by setting

`spark.databricks.delta.replaceWhere.constraintCheck.enabled` to false:

```
spark.conf.set("spark.databricks.delta.replaceWhere.constraintCheck.enabled", False)
```

In Delta Lake 1.0.0 and below, `replaceWhere` overwrites data matching a predicate over partition columns only. The following command atomically replaces the month in January in the target table, which is partitioned by `date`, with the data in `df`:

```
df.write \
  .format("delta") \
  .mode("overwrite") \
  .option("replaceWhere", "birthDate >= '2017-01-01' AND birthDate <= '2017-01-31'") \
  .save("/tmp/delta/people10m")
```

In Delta Lake 1.1.0 and above, if you want to fall back to the old behavior, you can disable the

`spark.databricks.delta.replaceWhere.dataColumns.enabled` flag:

```
spark.conf.set("spark.databricks.delta.replaceWhere.dataColumns.enabled", False)
```

Dynamic Partition Overwrites

Delta Lake 2.0 and above supports *dynamic* partition overwrite mode for partitioned tables.

When in dynamic partition overwrite mode, we overwrite all existing data in each logical partition for which the write will commit new data. Any existing logical partitions for which the write does not contain data will remain unchanged. This mode is only applicable when data is being written in overwrite mode:
either `INSERT OVERWRITE` in SQL, or a DataFrame write with `df.write.mode("overwrite")`.

Configure dynamic partition overwrite mode by setting the Spark session configuration

`spark.sql.sources.partitionOverwriteMode` to `dynamic`.

You can also enable this by setting
the `DataFrameWriter` option `partitionOverwriteMode` to `dynamic`. If present, the query-specific option overrides the mode defined in the session configuration. The default
for `partitionOverwriteMode` is `static`.

```
SET spark.sql.sources.partitionOverwriteMode=dynamic;
INSERT OVERWRITE TABLE default.people10m SELECT * FROM morePeople;
```

Note

Dynamic partition overwrite conflicts with the option `replaceWhere` for partitioned tables.

If dynamic partition overwrite is enabled in the Spark session configuration, and `replaceWhere` is provided as a `DataFrameWriter` option, data will be overwritten according to the `replaceWhere` expression (query-specific options override session configurations).

If both dynamic partition overwrite and `replaceWhere` are enabled in the `DataFrameWriter` options, an error will be thrown.

Important: Validate that the data being written with dynamic partition overwrite touches only the expected partitions. A single row in the incorrect partition can lead to unintentionally overwritting an entire partition. We strongly recommend using `replaceWhere` to explicitly specify which data to overwrite.

If a partition has been accidentally overwritten, you can use Restore a Delta table to an earlier state to undo the change.

31.8.3. Limit rows written in a file

You can use the SQL session configuration `spark.sql.files.maxRecordsPerFile` to specify the maximum number of records to write to a single file for a Delta Lake table. Specifying a value of zero or a negative value represents no limit.

You can also use the DataFrameWriter option `maxRecordsPerFile` when using the DataFrame APIs to write to a Delta Lake table. When `maxRecordsPerFile` is specified, the value of the SQL session configuration `spark.sql.files.maxRecordsPerFile` is ignored.

```
df.write.format("delta") \
  .mode("append") \
  .option("maxRecordsPerFile", "10000") \
  .save("/tmp/delta/people10m")
```

31.8.4. Idempotent writes

Sometimes a job that writes data to a Delta table is restarted due to various reasons (for example, job encounters a failure). The failed job may or may not have written the data to Delta table before terminating. In the case where the data is written to the Delta table, the restarted job writes the same data to the Delta table which results in duplicate data.

To address this, Delta tables support the following `DataFrameWriter` options to make the writes idempotent:

- `txnAppId`: A unique string that you can pass on each `DataFrame` write. For example, this can be the name of the job.
- `txnVersion`: A monotonically increasing number that acts as transaction version. This number needs to be unique for data that is being written to the Delta table(s). For example, this can be the epoch seconds of the instant when the query is attempted for the first time. Any subsequent restarts of the same job needs to have the same value for `txnVersion`.

The above combination of options needs to be unique for each new data that is being ingested into the Delta table and the `txnVersion` needs to be higher than the last data that was ingested into the Delta table. For example:

- Last successfully written data contains option values as `dailyETL:23423` (`txnAppId:txnVersion`).
- Next write of data should have `txnAppId = dailyETL` and `txnVersion` as at least `23424` (one more than the last written data `txnVersion`).
- Any attempt to write data with `txnAppId = dailyETL` and `txnVersion` as `23422` or less is ignored because the `txnVersion` is less than the last recorded `txnVersion` in the table.
- Attempt to write data with `txnAppId:txnVersion` as `anotherETL:23424` is successful writing data to the table as it contains a different `txnAppId` compared to the same option value in last ingested data.

Warning

This solution assumes that the data being written to Delta table(s) in multiple retries of the job is same. If a write attempt in a Delta table succeeds but due to some downstream failure there is a second write attempt with same txn options but different data, then that second write attempt will be ignored. This can cause unexpected results.

Example

```
app_id = ... # A unique string that is used as an application ID.
version = ... # A monotonically increasing number that acts as transaction version.

dataFrame.write.format(...).option("txnVersion", version).option("txnAppId", app_id).save(...)
```

31.8.5. Set user-defined commit metadata

You can specify user-defined strings as metadata in commits made by these operations, either using the DataFrameWriter option `userMetadata` or the SparkSession configuration `spark.databricks.delta.commitInfo.userMetadata`. If both of them have been specified, then the option takes preference. This user-defined metadata is readable in the history operation.

```
SET spark.databricks.delta.commitInfo.userMetadata=overwritten-for-fixing-incorrect-data
INSERT OVERWRITE default.people10m SELECT * FROM morePeople
```

31.9. Schema validation

Delta Lake automatically validates that the schema of the DataFrame being written is compatible with the schema of the table. Delta Lake uses the following rules to determine whether a write from a DataFrame to a table is compatible:

- All DataFrame columns must exist in the target table. If there are columns in the DataFrame not present in the table, an exception is raised. Columns present in the table but not in the DataFrame are set to null.
- DataFrame column data types must match the column data types in the target table. If they don't match, an exception is raised.
- DataFrame column names cannot differ only by case. This means that you cannot have columns such as "Foo" and "foo" defined in the same table. While you can use Spark in case sensitive or insensitive (default) mode, Parquet is case sensitive when storing and returning column information. Delta Lake is case-preserving but insensitive when storing the schema and has this restriction to avoid potential mistakes, data corruption, or loss issues.

Delta Lake support DDL to add new columns explicitly and the ability to update schema automatically.

If you specify other options, such as `partitionBy`, in combination with append mode, Delta Lake validates that they match and throws an error for any mismatch. When `partitionBy` is not present, appends automatically follow the partitioning of the existing data.

31.10. Update table schema

Delta Lake lets you update the schema of a table. The following types of changes are supported:

- Adding new columns (at arbitrary positions)
- Reordering existing columns

You can make these changes explicitly using DDL or implicitly using DML.

Important

When you update a Delta table schema, streams that read from that table terminate. If you want the stream to continue you must restart it.

31.10.1. Explicitly update schema

You can use the following DDL to explicitly change the schema of a table.

Add columns

```
ALTER TABLE table_name ADD COLUMNS (col_name data_type [COMMENT col_comment] [FIRST|AFTER colA_name], ...)
```

By default, nullability is `true`.

To add a column to a nested field, use:

```
ALTER TABLE table_name ADD COLUMNS (col_name.nested_col_name data_type [COMMENT col_comment] [FIRST|AFTER colA_name], ...)
```

Example

If the schema before running

`ALTER TABLE boxes ADD COLUMNS (colB.nested STRING AFTER field1)` is:

```
- root
| - colA
| - colB
| +-field1
| +-field2
```

the schema after is:

```
- root
| - colA
| - colB
| +-field1
| +-nested
| +-field2
```

Note

Adding nested columns is supported only for structs. Arrays and maps are not supported.

Change column comment or ordering

```
ALTER TABLE table_name ALTER [COLUMN] col_name col_name data_type [COMMENT col_comment] [FIRST|AFTER colA_name]
```

To change a column in a nested field, use:

```
ALTER TABLE table_name ALTER [COLUMN] col_name.nested_col_name nested_col_name data_type [COMMENT col_comment] [FIRST|AFTER colA_name]
```

Example

If the schema before running

`ALTER TABLE boxes CHANGE COLUMN colB.field2 field2 STRING FIRST` is:

```
- root
| - colA
| - colB
| +-field1
| +-field2
```

the schema after is:

```
- root
| - colA
| - colB
| +-field2
| +-field1
```

Replace columns

ALTER TABLE table_name REPLACE COLUMNS (col_name1 col_type1 [COMMENT col_comment1], ...)

Example

When running the following DDL:

ALTER TABLE **boxes** REPLACE **COLUMNS (colC STRING, colB STRUCT<field2:STRING, nested:STRING, field1:STRING>, colA STRING)**

if the schema before is:

```
- root
| - colA
| - colB
| +-field1
| +-field2
```

the schema after is:

```
- root
| - colC
| - colB
| +-field2
| +-nested
| +-field1
| - colA
```

Rename columns

Note: This feature is available in Delta Lake 1.2.0 and above. This feature is currently experimental.

To rename columns without rewriting any of the columns' existing data, you must enable column mapping for the table. See enable column mapping.

To rename a column:

ALTER TABLE table_name RENAME COLUMN **old_col_name** TO **new_col_name**

To rename a nested field:

ALTER TABLE table_name RENAME COLUMN **col_name.old_nested_field** TO **new_nested_field**

Example

When you run the following command:

ALTER TABLE **boxes** RENAME COLUMN **colB.field1** TO **field001**

If the schema before is:

```
- root
| - colA
| - colB
| +-field1
```

```
| +-field2
```

Then the schema after is:

```
- root
| - colA
| - colB
| +-field001
| +-field2
```

Drop columns

Note: This feature is available in Delta Lake 2.0 and above. This feature is currently experimental.

To drop columns as a metadata-only operation without rewriting any data files, you must enable column mapping for the table. See enable column mapping.

Important: Dropping a column from metadata does not delete the underlying data for the column in files.

To drop a column:

```
ALTER TABLE table_name DROP COLUMN col_name
```

To drop multiple columns:

```
ALTER TABLE table_name DROP COLUMNS (col_name_1, col_name_2)
```

Change column type or name

You can change a column's type or name or drop a column by rewriting the table. To do this, use the `overwriteSchema` option:

Change a column type

```
spark.read.table(...) \
  .withColumn("birthDate", col("birthDate").cast("date")) \
  .write \
  .format("delta") \
  .mode("overwrite")
  .option("overwriteSchema", "true") \
  .saveAsTable(...)
```

Change a column name

```
spark.read.table(...) \
  .withColumnRenamed("dateOfBirth", "birthDate") \
  .write \
  .format("delta") \
  .mode("overwrite") \
  .option("overwriteSchema", "true") \
  .saveAsTable(...)
```

31.10.2. Automatic schema update

Delta Lake can automatically update the schema of a table as part of a DML transaction (either appending or overwriting), and make the schema compatible with the data being written.

Add columns

Columns that are present in the DataFrame but missing from the table are automatically added as part of a write transaction when:

- `write` or `writeStream` have `.option("mergeSchema", "true")`
- `spark.databricks.delta.schema.autoMerge.enabled` is `true`

When both options are specified, the option from the `DataFrameWriter` takes precedence. The added columns are appended to the end of the struct they are present in. Case is preserved when appending a new column.

`NullType` columns

Because Parquet doesn't support `NullType`, `NullType` columns are dropped from the DataFrame when writing into Delta tables, but are still stored in the schema. When a different data type is received for that column, Delta Lake merges the schema to the new data type. If Delta Lake receives a `NullType` for an existing column, the old schema is retained and the new column is dropped during the write.

`NullType` in streaming is not supported. Since you must set schemas when using streaming this should be very rare. `NullType` is also not accepted for complex types such as `ArrayType` and `MapType`.

31.11. Replace table schema

By default, overwriting the data in a table does not overwrite the schema. When overwriting a table using `mode("overwrite")` without `replaceWhere`, you may still want to overwrite the schema of the data being written. You replace the schema and partitioning of the table by setting the `overwriteSchema` option to `true`:

```
df.write.option("overwriteSchema", "true")
```

31.12. Views on tables

Delta Lake supports the creation of views on top of Delta tables just like you might with a data source table.

The core challenge when you operate with views is resolving the schemas. If you alter a Delta table schema, you must recreate derivative views to account for any additions to the schema. For instance, if you add a new column to a Delta table, you must make sure that this column is available in the appropriate views built on top of that base table.

31.13. Table properties

You can store your own metadata as a table property using `TBLPROPERTIES` in `CREATE` and `ALTER`. You can then `SHOW` that metadata. For example:

```
ALTER TABLE default.people10m SET TBLPROPERTIES ('department' = 'accounting',
'delta.appendOnly' = 'true');

-- Show the table's properties.
SHOW TBLPROPERTIES default.people10m;

-- Show just the 'department' table property.
SHOW TBLPROPERTIES default.people10m ('department');
```

`TBLPROPERTIES` are stored as part of Delta table metadata. You cannot define new `TBLPROPERTIES` in a `CREATE` statement if a Delta table already exists in a given location.

In addition, to tailor behavior and performance, Delta Lake supports certain Delta table properties:

- Block deletes and updates in a Delta table: `delta.appendOnly=true`.
- Configure the time travel retention properties: `delta.logRetentionDuration=<interval-string>` and `delta.deletedFileRetentionDuration=<interval-string>`. For details, see Data retention.
- Configure the number of columns for which statistics are collected: `delta.dataSkippingNumIndexedCols=n`. This property indicates to the writer that statistics are to be collected only for the first `n` columns in the table. Also the data skipping code ignores statistics for any column beyond this column index. This property takes affect only for new data that is written out.

Note

Modifying a Delta table property is a write operation that will conflict with other concurrent write operations, causing them to fail. We recommend that you modify a table property only when there are no concurrent write operations on the table.

You can also set `delta.`-prefixed properties during the first commit to a Delta table using Spark configurations. For example, to initialize a Delta table with the property `delta.appendOnly=true`, set the Spark configuration `spark.databricks.delta.properties.defaults.appendOnly` to `true`. For example:

```
spark.sql("SET spark.databricks.delta.properties.defaults.appendOnly = true")
```

31.14. Table metadata

Delta Lake has rich features for exploring table metadata.

It supports `DESCRIBE TABLE`.

It also provides the following unique commands:

- `DESCRIBE DETAIL`
- `DESCRIBE HISTORY`

31.14.1. DESCRIBE DETAIL

Provides information about schema, partitioning, table size, and so on. For details, see Retrieve Delta table details.

31.14.2. DESCRIBE HISTORY

Provides provenance information, including the operation, user, and so on, and operation metrics for each write to a table. Table history is retained for 30 days. For details, see Retrieve Delta table history.

31.15. Configure SparkSession

For many Delta Lake operations, you enable integration with Apache Spark DataSourceV2 and Catalog APIs (since 3.0) by setting the following configurations when you create a new `SparkSession`.

```
from pyspark.sql import SparkSession

spark = SparkSession \
  .builder \
  .appName("...") \
  .master("...") \
  .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \
  .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog") \
  .getOrCreate()
```

Alternatively, you can add configurations when submitting your Spark application using `spark-submit` or when starting `spark-shell` or `pyspark` by specifying them as command-line parameters.

Bash

```
spark-submit --conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" --conf
"spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"  ...
```

Bash

```
pyspark --conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" --conf
"spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

31.16. Configure storage credentials

Delta Lake uses Hadoop FileSystem APIs to access the storage systems. The credentails for storage systems usually can be set through Hadoop configurations. Delta Lake provides multiple ways to set Hadoop configurations similar to Apache Spark.

31.16.1. Spark configurations

When you start a Spark application on a cluster, you can set the Spark configurations in the form of `spark.hadoop.*` to pass your custom Hadoop configurations. For example, Setting a value for `spark.hadoop.a.b.c` will pass the value as a Hadoop configuration `a.b.c`, and Delta Lake will use it to access Hadoop FileSystem APIs.

31.16.2. SQL session configurations

Spark SQL will pass all of the current SQL session configurations to Delta Lake, and Delta Lake will use them to access Hadoop FileSystem APIs. For example, `SET a.b.c=x.y.z` will tell Delta Lake to pass the value `x.y.z` as a Hadoop configuration `a.b.c`, and Delta Lake will use it to access Hadoop FileSystem APIs.

31.16.3. DataFrame options

Besides setting Hadoop file system configurations through the Spark (cluster) configurations or SQL session configurations, Delta supports reading Hadoop file system configurations from `DataFrameReader` and `DataFrameWriter` options (that is, option keys that start with the `fs.` prefix) when the table is read or written, by using `DataFrameReader.load(path)` or `DataFrameWriter.save(path)`.

For example, you can pass your storage credentails through DataFrame options:

```
df1 = spark.read.format("delta") \
  .option("fs.azure.account.key.<storage-account-name>.dfs.core.windows.net", "<storage-
account-access-key-1>") \
```

```
  .read("...")
df2 = spark.read.format("delta") \
  .option("fs.azure.account.key.<storage-account-name>.dfs.core.windows.net", "<storage-
account-access-key-2>") \
  .read("...")
df1.union(df2).write.format("delta") \
  .mode("overwrite") \
  .option("fs.azure.account.key.<storage-account-name>.dfs.core.windows.net", "<storage-
account-access-key-3>") \
  .save("...")
```

.

31.17. Table streaming reads and writes

Delta Lake is deeply integrated with Spark Structured Streaming through `readStream` and `writeStream`. Delta Lake overcomes many of the limitations typically associated with streaming systems and files, including:

- Maintaining “exactly-once” processing with more than one stream (or concurrent batch jobs)
- Efficiently discovering which files are new when using files as the source for a stream

For many Delta Lake operations on tables, you enable integration with Apache Spark DataSourceV2 and Catalog APIs (since 3.0) by setting configurations when you create a new `SparkSession`. See Configure SparkSession.

31.18. Delta table as a source

When you load a Delta table as a stream source and use it in a streaming query, the query processes all of the data present in the table as well as any new data that arrives after the stream is started.

```
spark.readStream.format("delta")
  .load("/tmp/delta/events")

import io.delta.implicits._
spark.readStream.delta("/tmp/delta/events")
```

31.18.1. Limit input rate

The following options are available to control micro-batches:

- `maxFilesPerTrigger`: How many new files to be considered in every micro-batch. The default is 1000.
- `maxBytesPerTrigger`: How much data gets processed in each micro-batch. This option sets a “soft max”, meaning that a batch processes approximately this amount of data and may process more than the limit in order to make the streaming query move forward in cases when the smallest input unit is larger than this limit. If you use `Trigger.Once` for your streaming, this option is ignored. This is not set by default.

If you use `maxBytesPerTrigger` in conjunction with `maxFilesPerTrigger`, the micro-batch processes data until either the `maxFilesPerTrigger` or `maxBytesPerTrigger` limit is reached.

Note

In cases when the source table transactions are cleaned up due to the `logRetentionDuration` configuration and the stream lags in processing, Delta Lake processes the

data corresponding to the latest available transaction history of the source table but does not fail the stream. This can result in data being dropped.

31.18.2. Ignore updates and deletes

Structured Streaming does not handle input that is not an append and throws an exception if any modifications occur on the table being used as a source. There are two main strategies for dealing with changes that cannot be automatically propagated downstream:

- You can delete the output and checkpoint and restart the stream from the beginning.
- You can set either of these two options:
  - `ignoreDeletes`: ignore transactions that delete data at partition boundaries.
  - `ignoreChanges`: re-process updates if files had to be rewritten in the source table due to a data changing operation such as `UPDATE`, `MERGE INTO`, `DELETE` (within partitions), or `OVERWRITE`. Unchanged rows may still be emitted, therefore your downstream consumers should be able to handle duplicates. Deletes are not propagated downstream. `ignoreChanges` subsumes `ignoreDeletes`. Therefore if you use `ignoreChanges`, your stream will not be disrupted by either deletions or updates to the source table.

Example

For example, suppose you have a table `user_events` with `date`, `user_email`, and `action` columns that is partitioned by `date`. You stream out of the `user_events` table and you need to delete data from it due to GDPR.

When you delete at partition boundaries (that is, the `WHERE` is on a partition column), the files are already segmented by value so the delete just drops those files from the metadata. Thus, if you just want to delete data from some partitions, you can use:

```
spark.readStream.format("delta")
  .option("ignoreDeletes", "true")
  .load("/tmp/delta/user_events")
```

However, if you have to delete data based on `user_email`, then you will need to use:

```
spark.readStream.format("delta")
  .option("ignoreChanges", "true")
  .load("/tmp/delta/user_events")
```

If you update a `user_email` with the `UPDATE` statement, the file containing the `user_email` in question is rewritten. When you use `ignoreChanges`, the new record is propagated downstream with all other unchanged records that were in the same file. Your logic should be able to handle these incoming duplicate records.

31.18.3. Specify initial position

You can use the following options to specify the starting point of the Delta Lake streaming source without processing the entire table.

- `startingVersion`: The Delta Lake version to start from. All table changes starting from this version (inclusive) will be read by the streaming source. You can obtain the commit versions from the `version` column of the DESCRIBE HISTORY command output.
- To return only the latest changes, specify `latest`.
- `startingTimestamp`: The timestamp to start from. All table changes committed at or after the timestamp (inclusive) will be read by the streaming source. One of:

- A timestamp string. For example, `"2019-01-01T00:00:00.000Z"`.
- A date string. For example, `"2019-01-01"`.

You cannot set both options at the same time; you can use only one of them. They take effect only when starting a new streaming query. If a streaming query has started and the progress has been recorded in its checkpoint, these options are ignored.

Important

Although you can start the streaming source from a specified version or timestamp, the schema of the streaming source is always the latest schema of the Delta table. You must ensure there is no incompatible schema change to the Delta table after the specified version or timestamp. Otherwise, the streaming source may return incorrect results when reading the data with an incorrect schema.

Example

For example, suppose you have a table `user_events`. If you want to read changes since version 5, use:

```
spark.readStream.format("delta")
  .option("startingVersion", "5")
  .load("/tmp/delta/user_events")
```

If you want to read changes since 2018-10-18, use:

```
spark.readStream.format("delta")
  .option("startingTimestamp", "2018-10-18")
  .load("/tmp/delta/user_events")
```

31.19. Delta table as a sink

You can also write data into a Delta table using Structured Streaming. The transaction log enables Delta Lake to guarantee exactly-once processing, even when there are other streams or batch queries running concurrently against the table.

Note

The Delta Lake `VACUUM` function removes all files not managed by Delta Lake but skips any directories that begin with `_`. You can safely store checkpoints alongside other data and metadata for a Delta table using a directory structure such as `<table_name>/_checkpoints`.

31.19.1. Append mode

By default, streams run in append mode, which adds new records to the table.

You can use the path method:

```
events.writeStream
  .format("delta")
  .outputMode("append")
  .option("checkpointLocation", "/tmp/delta/_checkpoints/")
  .start("/delta/events")
```

or the `toTable` method in Spark 3.1 and higher (the Delta Lake library 8.3 and above), as follows. (In Spark versions before 3.1 (the Delta Lake library 8.2 and below), use the `table` method instead.)

```
events.writeStream
  .format("delta")
  .outputMode("append")
  .option("checkpointLocation", "/tmp/delta/events/_checkpoints/")
  .toTable("events")
```

31.19.2. Complete mode

You can also use Structured Streaming to replace the entire table with every batch. One example use case is to compute a summary using aggregation:

```
(spark.readStream
  .format("delta")
  .load("/tmp/delta/events")
  .groupBy("customerId")
  .count()
  .writeStream
  .format("delta")
  .outputMode("complete")
  .option("checkpointLocation", "/tmp/delta/eventsByCustomer/_checkpoints/")
  .start("/tmp/delta/eventsByCustomer")
)
```

The preceding example continuously updates a table that contains the aggregate number of events by customer.

For applications with more lenient latency requirements, you can save computing resources with one-time triggers. Use these to update summary aggregation tables on a given schedule, processing only new data that has arrived since the last update.

31.20. Idempotent table writes in foreachBatch

Note

Available in Delta Lake 2.0.0 and above.

The command foreachBatch allows you to specify a function that is executed on the output of every micro-batch after arbitrary transformations in the streaming query. This allows implementating a `foreachBatch` function that can write the micro-batch output to one or more target Delta table destinations. However, `foreachBatch` does not make those writes idempotent as those write attempts lack the information of whether the batch is being re-executed or not. For example, rerunning a failed batch could result in duplicate data writes.

To address this, Delta tables support the following `DataFrameWriter` options to make the writes idempotent:

- `txnAppId`: A unique string that you can pass on each `DataFrame` write. For example, you can use the StreamingQuery ID as `txnAppId`.
- `txnVersion`: A monotonically increasing number that acts as transaction version.

Delta table uses the combination of `txnAppId` and `txnVersion` to identify duplicate writes and ignore them.

If a batch write is interrupted with a failure, rerunning the batch uses the same application and batch ID, which would help the runtime correctly identify duplicate writes and ignore them. Application ID (`txnAppId`) can be any user-generated unique string and does not have to be related to the stream ID.

Warning

If you delete the streaming checkpoint and restart the query with a new checkpoint, you must provide a different `appId`; otherwise, writes from the restarted query will be ignored because it will contain the same `txnAppId` and the batch ID would start from 0.

The same `DataFrameWriter` options can be used to achieve the idempotent writes in non-Streaming job. For details _.

31.20.1. Example

```
app_id = ... # A unique string that is used as an application ID.

def writeToDeltaLakeTableIdempotent(batch_df, batch_id):
  batch_df.write.format(...).option("txnVersion", batch_id).option("txnAppId",
app_id).save(...) # location 1
  batch_df.write.format(...).option("txnVersion", batch_id).option("txnAppId",
app_id).save(...) # location 2
```

31.21. Table deletes, updates, and merges

Delta Lake supports several statements to facilitate deleting data from and updating data in Delta tables.

31.22. Delete from a table

You can remove data that matches a predicate from a Delta table. For instance, in a table named `people10m` or a path at `/tmp/delta/people-10m`, to delete all rows corresponding to people with a value in the `birthDate` column from before `1955`, you can run the following:

```
DELETE FROM people10m WHERE birthDate < '1955-01-01'

DELETE FROM delta.`/tmp/delta/people-10m` WHERE birthDate < '1955-01-01'
```

See Configure SparkSession for the steps to enable support for SQL commands.

See the Delta Lake APIs for details.

Important

`delete` removes the data from the latest version of the Delta table but does not remove it from the physical storage until the old versions are explicitly vacuumed. See vacuum for details.

Tip

When possible, provide predicates on the partition columns for a partitioned Delta table as such predicates can significantly speed up the operation.

31.23. Update a table

You can update data that matches a predicate in a Delta table. For example, in a table named `people10m` or a path at `/tmp/delta/people-10m`, to change an abbreviation in the `gender` column from `M` or `F` to `Male` or `Female`, you can run the following:

```
UPDATE people10m SET gender = 'Female' WHERE gender = 'F';
UPDATE people10m SET gender = 'Male' WHERE gender = 'M';

UPDATE delta.`/tmp/delta/people-10m` SET gender = 'Female' WHERE gender = 'F';
UPDATE delta.`/tmp/delta/people-10m` SET gender = 'Male' WHERE gender = 'M';
```

Tip

Similar to delete, update operations can get a significant speedup with predicates on partitions.

31.24. Upsert into a table using merge

You can upsert data from a source table, view, or DataFrame into a target Delta table by using the `MERGE` SQL operation. Delta Lake supports inserts, updates and deletes in `MERGE`, and it supports extended syntax beyond the SQL standards to facilitate advanced use cases.

Suppose you have a source table named `people10mupdates` or a source path at `/tmp/delta/people-10m-updates` that contains new data for a target table named `people10m` or a target path at `/tmp/delta/people-10m`. Some of these new records may already be present in the target data. To merge the new data, you want to update rows where the person's `id` is already present and insert the new rows where no matching `id` is present. You can run the following:

```
MERGE INTO people10m
USING people10mupdates
ON people10m.id = people10mupdates.id
WHEN MATCHED THEN
  UPDATE SET
    id = people10mupdates.id,
    firstName = people10mupdates.firstName,
    middleName = people10mupdates.middleName,
    lastName = people10mupdates.lastName,
    gender = people10mupdates.gender,
    birthDate = people10mupdates.birthDate,
    ssn = people10mupdates.ssn,
    salary = people10mupdates.salary
WHEN NOT MATCHED
  THEN INSERT (
    id,
    firstName,
    middleName,
    lastName,
    gender,
    birthDate,
    ssn,
    salary
  )
  VALUES (
    people10mupdates.id,
    people10mupdates.firstName,
    people10mupdates.middleName,
    people10mupdates.lastName,
    people10mupdates.gender,
    people10mupdates.birthDate,
    people10mupdates.ssn,
    people10mupdates.salary
  )
```

See Configure SparkSession for the steps to enable support for SQL commands.

See the Delta Lake APIs for Scala, Java, and Python syntax details.

Delta Lake merge operations typically require two passes over the source data. If your source data contains nondeterministic expressions, multiple passes on the source data can produce different rows causing incorrect results. Some common examples of nondeterministic expressions include the `current_date` and `current_timestamp` functions. If you cannot avoid using non-deterministic functions, consider saving the source data to storage, for example as a temporary Delta table. Caching the source data may not address this issue, as cache invalidation can cause the source data to be recomputed partially or completely (for example when a cluster loses some of it executors when scaling down).

31.24.1. Schema validation

`merge` automatically validates that the schema of the data generated by insert and update expressions are compatible with the schema of the table. It uses the following rules to determine whether the `merge` operation is compatible:

- For `update` and `insert` actions, the specified target columns must exist in the target Delta table.

- For `updateAll` and `insertAll` actions, the source dataset must have all the columns of the target Delta table. The source dataset can have extra columns and they are ignored.
- If you do not want the extra columns to be ignored and instead want to update the target table schema to include new columns, see Automatic schema evolution.
- For all actions, if the data type generated by the expressions producing the target columns are different from the corresponding columns in the target Delta table, `merge` tries to cast them to the types in the table.

31.24.2. Automatic schema evolution

By default, `updateAll` and `insertAll` assign all the columns in the target Delta table with columns of the same name from the source dataset. Any columns in the source dataset that don't match columns in the target table are ignored. However, in some use cases, it is desirable to automatically add source columns to the target Delta table. To automatically update the table schema during a `merge` operation with `updateAll` and `insertAll` (at least one of them), you can set the Spark session configuration `spark.databricks.delta.schema.autoMerge.enabled` to `true` before running the `merge` operation.

Note

- Schema evolution occurs only when there is either an `updateAll` (`UPDATE SET *`) or an `insertAll` (`INSERT *`) action, or both.
- `update` and `insert` actions cannot explicitly refer to target columns that do not already exist in the target table (even it there are `updateAll` or `insertAll` as one of the clauses). See the examples below.

31.24.3. Performance tuning

You can reduce the time taken by merge using the following approaches:

- **Reduce the search space for matches**: By default, the `merge` operation searches the entire Delta table to find matches in the source table. One way to speed up `merge` is to reduce the search space by adding known constraints in the match condition. For example, suppose you have a table that is partitioned by `country` and `date` and you want to use `merge` to update information for the last day and a specific country. Adding the condition
- `events.date = current_date() AND events.country = 'USA'`
- will make the query faster as it looks for matches only in the relevant partitions. Furthermore, it will also reduce the chances of conflicts with other concurrent operations. See Concurrency control for more details.
- **Compact files**: If the data is stored in many small files, reading the data to search for matches can become slow. You can compact small files into larger files to improve read throughput. See Compact files for details.
- **Control the shuffle partitions for writes**: The `merge` operation shuffles data multiple times to compute and write the updated data. The number of tasks used to shuffle is controlled by the Spark session configuration `spark.sql.shuffle.partitions`. Setting this parameter not only controls the parallelism but also determines the number of output files. Increasing the value increases parallelism but also generates a larger number of smaller data files.
- **Repartition output data before write**: For partitioned tables, `merge` can produce a much larger number of small files than the number of shuffle partitions. This is because every shuffle task can write multiple files in multiple partitions, and can become a performance bottleneck. In many cases, it helps to repartition the output data by the table's partition columns before writing it. You enable this by setting the

Spark session configuration `spark.databricks.delta.merge.repartitionBeforeWrite.enabled` to `true`.

31.24.4. Merge examples

Here are a few examples on how to use `merge` in different scenarios.

31.24.5. Data deduplication when writing into Delta tables

A common ETL use case is to collect logs into Delta table by appending them to a table. However, often the sources can generate duplicate log records and downstream deduplication steps are needed to take care of them. With `merge`, you can avoid inserting the duplicate records.

```
MERGE INTO logs
USING newDedupedLogs
ON logs.uniqueId = newDedupedLogs.uniqueId
WHEN NOT MATCHED
  THEN INSERT *
```

Note

The dataset containing the new logs needs to be deduplicated within itself. By the SQL semantics of merge, it matches and deduplicates the new data with the existing data in the table, but if there is duplicate data within the new dataset, it is inserted. Hence, deduplicate the new data before merging into the table.

If you know that you may get duplicate records only for a few days, you can optimized your query further by partitioning the table by date, and then specifying the date range of the target table to match on.

```
MERGE INTO logs
USING newDedupedLogs
ON logs.uniqueId = newDedupedLogs.uniqueId AND logs.date > current_date() - INTERVAL 7 DAYS
WHEN NOT MATCHED AND newDedupedLogs.date > current_date() - INTERVAL 7 DAYS
  THEN INSERT *
```

This is more efficient than the previous command as it looks for duplicates only in the last 7 days of logs, not the entire table. Furthermore, you can use this insert-only merge with Structured Streaming to perform continuous deduplication of the logs.

- In a streaming query, you can use merge operation in `foreachBatch` to continuously write any streaming data to a Delta table with deduplication. See the following streaming example for more information on `foreachBatch`.
- In another streaming query, you can continuously read deduplicated data from this Delta table. This is possible because an insert-only merge only appends new data to the Delta table.

31.24.6. Slowly changing data (SCD) Type 2 operation into Delta tables

Another common operation is SCD Type 2, which maintains history of all changes made to each key in a dimensional table. Such operations require updating existing rows to mark previous values of keys as old, and the inserting the new rows as the latest values. Given a source table with updates and the target table with the dimensional data, SCD Type 2 can be expressed with `merge`.

Here is a concrete example of maintaining the history of addresses for a customer along with the active date range of each address. When a customer's address needs to be updated, you have to mark the previous address as not the current one, update its active date range, and add the new address as the current one.

```
val customersTable: DeltaTable = ...   // table with schema (customerId, address, current,
effectiveDate, endDate)

val updatesDF: DataFrame = ...          // DataFrame with schema (customerId, address,
effectiveDate)

// Rows to INSERT new addresses of existing customers
val newAddressesToInsert = updatesDF
  .as("updates")
  .join(customersTable.toDF.as("customers"), "customerid")
  .where("customers.current = true AND updates.address <> customers.address")

// Stage the update by unioning two sets of rows
// 1. Rows that will be inserted in the whenNotMatched clause
// 2. Rows that will either update the current addresses of existing customers or insert the
new addresses of new customers
val stagedUpdates = newAddressesToInsert
  .selectExpr("NULL as mergeKey", "updates.*")   // Rows for 1.
  .union(
    updatesDF.selectExpr("updates.customerId as mergeKey", "*")  // Rows for 2.
  )

// Apply SCD Type 2 operation using merge
customersTable
  .as("customers")
  .merge(
    stagedUpdates.as("staged_updates"),
    "customers.customerId = mergeKey")
  .whenMatched("customers.current = true AND customers.address <> staged_updates.address")
  .updateExpr(Map(                                      // Set current to false and endDate to
source's effective date.
    "current" -> "false",
    "endDate" -> "staged_updates.effectiveDate"))
  .whenNotMatched()
  .insertExpr(Map(
    "customerid" -> "staged_updates.customerId",
    "address" -> "staged_updates.address",
    "current" -> "true",
    "effectiveDate" -> "staged_updates.effectiveDate",  // Set current to true along with the
new address and its effective date.
    "endDate" -> "null"))
  .execute()
```

31.24.7. Write change data into a Delta table

Similar to SCD, another common use case, often called change data capture (CDC), is to apply all data changes generated from an external database into a Delta table. In other words, a set of updates, deletes, and inserts applied to an external table needs to be applied to a Delta table. You can do this using `merge` as follows.

```
val deltaTable: DeltaTable = ... // DeltaTable with schema (key, value)

// DataFrame with changes having following columns
// - key: key of the change
// - time: time of change for ordering between changes (can replaced by other ordering id)
// - newValue: updated or inserted value if key was not deleted
// - deleted: true if the key was deleted, false if the key was inserted or updated
val changesDF: DataFrame = ...

// Find the latest change for each key based on the timestamp
// Note: For nested structs, max on struct is computed as
// max on first struct field, if equal fall back to second fields, and so on.
val latestChangeForEachKey = changesDF
  .selectExpr("key", "struct(time, newValue, deleted) as otherCols" )
```

```
  .groupBy("key")
  .agg(max("otherCols").as("latest"))
  .selectExpr("key", "latest.*")

deltaTable.as("t")
  .merge(
    latestChangeForEachKey.as("s"),
    "s.key = t.key")
  .whenMatched("s.deleted = true")
  .delete()
  .whenMatched()
  .updateExpr(Map("key" -> "s.key", "value" -> "s.newValue"))
  .whenNotMatched("s.deleted = false")
  .insertExpr(Map("key" -> "s.key", "value" -> "s.newValue"))
  .execute()
```

31.24.8. Upsert from streaming queries using foreachBatch

You can use a combination of `merge` and `foreachBatch` (see foreachbatch for more information) to write complex upserts from a streaming query into a Delta table. For example:

- **Write streaming aggregates in Update Mode**: This is much more efficient than Complete Mode.

```
import io.delta.tables.*

val deltaTable = DeltaTable.forPath(spark, "/data/aggregates")

// Function to upsert microBatchOutputDF into Delta table using merge
def upsertToDelta(microBatchOutputDF: DataFrame, batchId: Long) {
  deltaTable.as("t")
    .merge(
      microBatchOutputDF.as("s"),
      "s.key = t.key")
    .whenMatched().updateAll()
    .whenNotMatched().insertAll()
    .execute()
}

// Write the output of a streaming aggregation query into Delta table
streamingAggregatesDF.writeStream
  .format("delta")
  .foreachBatch(upsertToDelta _)
  .outputMode("update")
  .start()
```

- **Write a stream of database changes into a Delta table**: The merge query for writing change data can be used in `foreachBatch` to continuously apply a stream of changes to a Delta table.
- **Write a stream data into Delta table with deduplication**: The insert-only merge query for deduplication can be used in `foreachBatch` to continuously write data (with duplicates) to a Delta table with automatic deduplication.

Note

Make sure that your `merge` statement inside `foreachBatch` is idempotent as restarts of the streaming query can apply the operation on the same batch of data multiple times.

When `merge` is used in `foreachBatch`, the input data rate of the streaming query (reported through `StreamingQueryProgress` and visible in the notebook rate graph) may be reported as a multiple

of the actual rate at which data is generated at the source. This is because `merge` reads the input data multiple times causing the input metrics to be multiplied. If this is a bottleneck, you can cache the batch DataFrame before `merge` and then uncache it after `merge`.

31.25. Table utility commands

Delta tables support a number of utility commands.

For many Delta Lake operations, you enable integration with Apache Spark DataSourceV2 and Catalog APIs (since 3.0) by setting configurations when you create a new `SparkSession`. See Configure SparkSession.

31.25.1. Remove files no longer referenced by a Delta table

You can remove files no longer referenced by a Delta table and are older than the retention threshold by running the vacuum command on the table. vacuum is not triggered automatically. The default retention threshold for the files is 7 days. To change this behavior, see Data retention.

Important

- `vacuum` removes all files from directories not managed by Delta Lake, ignoring directories beginning with `_`. If you are storing additional metadata like Structured Streaming checkpoints within a Delta table directory, use a directory name such as `_checkpoints`.
- `vacuum` deletes only data files, not log files. Log files are deleted automatically and asynchronously after checkpoint operations. The default retention period of log files is 30 days, configurable through the `delta.logRetentionDuration` property which you set with the `ALTER TABLE SET TBLPROPERTIES` SQL method. See Table properties.
- The ability to time travel back to a version older than the retention period is lost after running `vacuum`.

```
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, pathToTable)  # path-based tables, or
deltaTable = DeltaTable.forName(spark, tableName)    # Hive metastore-based tables

deltaTable.vacuum()        # vacuum files not required by versions older than the default
retention period

deltaTable.vacuum(100)     # vacuum files not required by versions more than 100 hours old
```

Note

When using `VACUUM`, to configure Spark to delete files in parallel (based on the number of shuffle partitions) set the session
configuration "`spark.databricks.delta.vacuum.parallelDelete.enabled`" to "`true`".

See the Delta Lake APIs for Scala, Java, and Python syntax details.

Warning

It is recommended that you set a retention interval to be at least 7 days, because old snapshots and uncommitted files can still be in use by concurrent readers or writers to the table. If `VACUUM` cleans up active files, concurrent readers can fail or, worse, tables can be corrupted when `VACUUM` deletes files that have not yet been committed. You must choose an interval that is longer than the longest running concurrent transaction and the longest period that any stream can lag behind the most recent update to the table.

Delta Lake has a safety check to prevent you from running a dangerous `VACUUM` command. If you are certain that there are no operations being performed on this table that take longer than the retention interval you plan

to specify, you can turn off this safety check by setting the Spark configuration property `spark.databricks.delta.retentionDurationCheck.enabled` to `false`.

31.25.2. Retrieve Delta table history

You can retrieve information on the operations, user, timestamp, and so on for each write to a Delta table by running the `history` command. The operations are returned in reverse chronological order. By default table history is retained for 30 days.

```
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, pathToTable)

fullHistoryDF = deltaTable.history()    # get the full history of the table

lastOperationDF = deltaTable.history(1) # get the last operation
```

History schema

The output of the `history` operation has the following columns.

Column	Type	Description
version	long	Table version generated by the operation.
timestamp	timestamp	When this version was committed.
userId	string	ID of the user that ran the operation.
userName	string	Name of the user that ran the operation.
operation	string	Name of the operation.
operationParameters	map	Parameters of the operation (for example, predicates.)
job	struct	Details of the job that ran the operation.
notebook	struct	Details of notebook from which the operation was run.
clusterId	string	ID of the cluster on which the operation ran.
readVersion	long	Version of the table that was read to perform the write operation.
isolationLevel	string	Isolation level used for this operation.

Column	Type	Description
isBlindAppend	boolean	Whether this operation appended data.
operationMetrics	map	Metrics of the operation (for example, number of rows and files modified.)
userMetadata	string	User-defined commit metadata if it was specified

```
+-------+-------------------+------+--------+---------+--------------------+----+--------+---
-----+-----------+-------------+-------------+--------------------+
|version|          timestamp|userId|userName|operation| operationParameters|
job|notebook|clusterId|readVersion|isolationLevel|isBlindAppend|    operationMetrics|
+-------+-------------------+------+--------+---------+--------------------+----+--------+----
-----+-----------+-------------+-------------+--------------------+
|      5|2019-07-29 14:07:47|  null|    null|   DELETE|[predicate -> ["(...|null|    null|
null|          4|  Serializable|        false|[numTotalRows -> ...|
|      4|2019-07-29 14:07:41|  null|    null|   UPDATE|[predicate -> (id...|null|    null|
null|          3|  Serializable|        false|[numTotalRows -> ...|
|      3|2019-07-29 14:07:29|  null|    null|   DELETE|[predicate -> ["(...|null|    null|
null|          2|  Serializable|        false|[numTotalRows -> ...|
|      2|2019-07-29 14:06:56|  null|    null|   UPDATE|[predicate -> (id...|null|    null|
null|          1|  Serializable|        false|[numTotalRows -> ...|
|      1|2019-07-29 14:04:31|  null|    null|   DELETE|[predicate -> ["(...|null|    null|
null|          0|  Serializable|        false|[numTotalRows -> ...|
|      0|2019-07-29 14:01:40|  null|    null|    WRITE|[mode -> ErrorIfE...|null|    null|
null|       null|  Serializable|         true|[numFiles -> 2, n...|
+-------+-------------------+------+--------+---------+--------------------+----+--------+----
-----+-----------+-------------+-------------+--------------------+
```

Note

Some of the columns may be nulls because the corresponding information may not be available in your environment.

Columns added in the future will always be added after the last column.

Operation metrics keys

The `history` operation returns a collection of operations metrics in the `operationMetrics` column map.

The following table lists the map key definitions by operation.

Operation	Metric name	Description
WRITE, CREATE TABLE AS SELECT, REPLACE TABLE AS SELECT, COPY INTO		
	numFiles	Number of files written.
	numOutputBytes	Size in bytes of the written contents.
	numOutputRows	Number of rows written.

Operation	Metric name	Description
STREAMING UPDATE		
	numAddedFiles	Number of files added.
	numRemovedFiles	Number of files removed.
	numOutputRows	Number of rows written.
	numOutputBytes	Size of write in bytes.
DELETE		
	numAddedFiles	Number of files added. Not provided when partitions of the table are deleted.
	numRemovedFiles	Number of files removed.
	numDeletedRows	Number of rows removed. Not provided when partitions of the table are deleted.
	numCopiedRows	Number of rows copied in the process of deleting files.
	executionTimeMs	Time taken to execute the entire operation.
	scanTimeMs	Time taken to scan the files for matches.
	rewriteTimeMs	Time taken to rewrite the matched files.
TRUNCATE		
	numRemovedFiles	Number of files removed.
	executionTimeMs	Time taken to execute the entire operation.

Operation	Metric name	Description
MERGE		
	numSourceRows	Number of rows in the source DataFrame.
	numTargetRowsInserted	Number of rows inserted into the target table.
	numTargetRowsUpdated	Number of rows updated in the target table.
	numTargetRowsDeleted	Number of rows deleted in the target table.
	numTargetRowsCopied	Number of target rows copied.
	numOutputRows	Total number of rows written out.
	numTargetFilesAdded	Number of files added to the sink(target).
	numTargetFilesRemoved	Number of files removed from the sink(target).
	executionTimeMs	Time taken to execute the entire operation.
	scanTimeMs	Time taken to scan the files for matches.
	rewriteTimeMs	Time taken to rewrite the matched files.
UPDATE		
	numAddedFiles	Number of files added.
	numRemovedFiles	Number of files removed.
	numUpdatedRows	Number of rows updated.

Operation	Metric name	Description
	numCopiedRows	Number of rows just copied over in the process of updating files.
	executionTimeMs	Time taken to execute the entire operation.
	scanTimeMs	Time taken to scan the files for matches.
	rewriteTimeMs	Time taken to rewrite the matched files.
FSCK	numRemovedFiles	Number of files removed.
CONVERT	numConvertedFiles	Number of Parquet files that have been converted.
OPTIMIZE		
	numAddedFiles	Number of files added.
	numRemovedFiles	Number of files optimized.
	numAddedBytes	Number of bytes added after the table was optimized.
	numRemovedBytes	Number of bytes removed.
	minFileSize	Size of the smallest file after the table was optimized.
	p25FileSize	Size of the 25th percentile file after the table was optimized.
	p50FileSize	Median file size after the table was optimized.
	p75FileSize	Size of the 75th percentile file after the table was optimized.

Operation		Metric name	Description
		maxFileSize	Size of the largest file after the table was optimized.

Operation	Metric name	Description
RESTORE		
	tableSizeAfterRestore	Table size in bytes after restore.
	numOfFilesAfterRestore	Number of files in the table after restore.
	numRemovedFiles	Number of files removed by the restore operation.
	numRestoredFiles	Number of files that were added as a result of the restore.
	removedFilesSize	Size in bytes of files removed by the restore.
	restoredFilesSize	Size in bytes of files added by the restore.

31.25.3. Retrieve Delta table details

You can retrieve detailed information about a Delta table (for example, number of files, data size) using `DESCRIBE DETAIL`.

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

DESCRIBE DETAIL eventsTable
```

Detail schema

The output of this operation has only one row with the following schema.

Column	Type	Description
format	string	Format of the table, that is, `delta`.

Column	Type	Description
id	string	Unique ID of the table.
name	string	Name of the table as defined in the metastore.
description	string	Description of the table.
location	string	Location of the table.
createdAt	timestamp	When the table was created.
lastModified	timestamp	When the table was last modified.
partitionColumns	array of strings	Names of the partition columns if the table is partitioned.
numFiles	long	Number of the files in the latest version of the table.
sizeInBytes	int	The size of the latest snapshot of the table in bytes.
properties	string-string map	All the properties set for this table.
minReaderVersion	int	Minimum version of readers (according to the log protocol) that can read the table.
minWriterVersion	int	Minimum version of writers (according to the log protocol) that can write to the table.

```
+------+--------------------+------------------+-----------+--------------------+------------
-------+------------------+---------------+--------+-----------+----------+---------------
+----------------+
|format|                  id|              name|description|            location|
createdAt|
lastModified|partitionColumns|numFiles|sizeInBytes|properties|minReaderVersion|minWriterVersio
n|
+------+--------------------+------------------+-----------+--------------------+------------
-------+------------------+---------------+--------+-----------+----------+---------------
+----------------+
| delta|d31f82d2-a69f-42e...|default.deltatable|       null|file:/Users/tuor/...|2020-06-05
12:20:...|2020-06-05 12:20:20|              []|      10|      12345|        []|
1|               2|
+------+--------------------+------------------+-----------+--------------------+------------
-------+------------------+---------------+--------+-----------+----------+---------------
+----------------+
```

31.25.4. Generate a manifest file

You can a generate manifest file for a Delta table that can be used by other processing engines (that is, other than Apache Spark) to read the Delta table. For example, to generate a manifest file that can be used by Presto and Athena to read a Delta table, you run the following:

```
deltaTable = DeltaTable.forPath(<path-to-delta-table>)
deltaTable.generate("symlink_format_manifest")
```

31.25.5. Convert a Parquet table to a Delta table

Convert a Parquet table to a Delta table in-place. This command lists all the files in the directory, creates a Delta Lake transaction log that tracks these files, and automatically infers the data schema by reading the footers of all Parquet files. If your data is partitioned, you must specify the schema of the partition columns as a DDL-formatted string (that is, `<column-name1> <type>, <column-name2> <type>, ...`).

Note

If a Parquet table was created by Structured Streaming, the listing of files can be avoided by using the `_spark_metadata` sub-directory as the source of truth for files contained in the table setting the SQL configuration `spark.databricks.delta.convert.useMetadataLog` to `true`.

```
from delta.tables import *

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

# Convert partitioned parquet table at path '<path-to-table>' and partitioned by integer
column named 'part'
partitionedDeltaTable = DeltaTable.convertToDelta(spark, "parquet.`<path-to-table>`", "part
int")
```

Note

Any file not tracked by Delta Lake is invisible and can be deleted when you run `vacuum`. You should avoid updating or appending data files during the conversion process. After the table is converted, make sure all writes go through Delta Lake.

31.25.6. Convert a Delta table to a Parquet table

You can easily convert a Delta table back to a Parquet table using the following steps:

1. If you have performed Delta Lake operations that can change the data files (for example, `delete` or `merge`), run vacuum with retention of 0 hours to delete all data files that do not belong to the latest version of the table.
2. Delete the `_delta_log` directory in the table directory.

31.25.7. Restore a Delta table to an earlier state

You can restore a Delta table to its earlier state by using the `RESTORE` command. A Delta table internally maintains historic versions of the table that enable it to be restored to an earlier state. A version corresponding to the earlier state or a timestamp of when the earlier state was created are supported as options by the `RESTORE` command.

Important

You can restore an already restored table.

Restoring a table to an older version where the data files were deleted manually or by `vacuum` will fail. Restoring to this version partially is still possible if `spark.sql.files.ignoreMissingFiles` is set to `true`.

The timestamp format for restoring to an earlier state is `yyyy-MM-dd HH:mm:ss`. Providing only a date(`yyyy-MM-dd`) string is also supported.

```
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, <path-to-table>)  # path-based tables, or
deltaTable = DeltaTable.forName(spark, <table-name>)    # Hive metastore-based tables

deltaTable.restoreToVersion(0) # restore table to oldest version

deltaTable.restoreToTimestamp('2019-02-14') # restore to a specific timestamp
```

Important

Restore is considered a data-changing operation. Delta Lake log entries added by the `RESTORE` command contain dataChange set to true. If there is a downstream application, such as a Structured streaming job that processes the updates to a Delta Lake table, the data change log entries added by the restore operation are considered as new data updates, and processing them may result in duplicate data.

For example:

Table version	Operation	Delta log updates	Records in data change log updates
0	INSERT	AddFile(/path/to/file-1, dataChange = true)	(name = Viktor, age = 29, (name = George, age = 55)
1	INSERT	AddFile(/path/to/file-2, dataChange = true)	(name = George, age = 39)
2	OPTIMIZE	AddFile(/path/to/file-3, dataChange = false), RemoveFile(/path/to/file-1), RemoveFile(/path/to/file-2)	(No records as Optimize compaction does not change the data in the table)
3	RESTORE(version=1)	RemoveFile(/path/to/file-3), AddFile(/path/to/file-1, dataChange = true), AddFile(/path/to/file-2, dataChange = true)	(name = Viktor, age = 29), (name = George, age = 55), (name = George, age = 39)

In the preceding example, the `RESTORE` command results in updates that were already seen when reading the Delta table version 0 and 1. If a streaming query was reading this table, then these files will be considered as newly added data and will be processed again.

Restore metrics

`RESTORE` reports the following metrics as a single row DataFrame once the operation is complete:

- `table_size_after_restore`: The size of the table after restoring.

- `num_of_files_after_restore`: The number of files in the table after restoring.
- `num_removed_files`: Number of files removed (logically deleted) from the table.
- `num_restored_files`: Number of files restored due to rolling back.
- `removed_files_size`: Total size in bytes of the files that are removed from the table.
- `restored_files_size`: Total size in bytes of the files that are restored.

```
1  RESTORE TABLE flights TO VERSION AS OF 0
```

▸ (17) Spark Jobs

	table_size_after_restore	num_of_files_after_restore	num_removed_files	num_restored_files	removed_files_size	restored_files_size
1	198415033	6999	301	6996	115778290	198393015

31.26. Constraints

Delta tables support standard SQL constraint management clauses that ensure that the quality and integrity of data added to a table is automatically verified. When a constraint is violated, Delta Lake throws an `InvariantViolationException` to signal that the new data can't be added.

Important

Adding a constraint automatically upgrades the table writer protocol version. See Table protocol versioning to understand table protocol versioning and what it means to upgrade the protocol version.

Two types of constraints are supported:

- `NOT NULL`: indicates that values in specific columns cannot be null.
- `CHECK`: indicates that a specified Boolean expression must be true for each input row.

31.26.1. NOT NULL constraint

You specify `NOT NULL` constraints in the schema when you create a table and drop `NOT NULL` constraints using the `ALTER TABLE CHANGE COLUMN` command.

```
> CREATE TABLE default.people10m (
    id INT NOT NULL,
    firstName STRING,
    middleName STRING NOT NULL,
    lastName STRING,
    gender STRING,
    birthDate TIMESTAMP,
    ssn STRING,
    salary INT
  ) USING DELTA;

> ALTER TABLE default.people10m CHANGE COLUMN middleName DROP NOT NULL;
```

If you specify a `NOT NULL` constraint on a column nested within a struct, the parent struct is also constrained to not be null. However, columns nested within array or map types do not accept `NOT NULL` constraints.

31.26.2. CHECK constraint

You manage `CHECK` constraints using the `ALTER TABLE ADD CONSTRAINT` and `ALTER TABLE DROP CONSTRAINT` commands. `ALTER TABLE ADD CONSTRAINT` verifies that all existing rows satisfy the constraint before adding it to the table.

```
CREATE TABLE default.people10m (
  id INT,
  firstName STRING,
  middleName STRING,
  lastName STRING,
  gender STRING,
  birthDate TIMESTAMP,
  ssn STRING,
  salary INT
) USING DELTA;

> ALTER TABLE default.people10m ADD CONSTRAINT dateWithinRange CHECK (birthDate > '1900-01-
01');
> ALTER TABLE default.people10m DROP CONSTRAINT dateWithinRange;
```

`CHECK` constraints are table properties in the output of the `DESCRIBE DETAIL` and `SHOW TBLPROPERTIES` commands.

```
> ALTER TABLE default.people10m ADD CONSTRAINT validIds CHECK (id > 1 and id < 99999999);

> DESCRIBE DETAIL default.people10m;

> SHOW TBLPROPERTIES default.people10m;
```

31.27. Storage configuration

Delta Lake ACID guarantees are predicated on the atomicity and durability guarantees of the storage system. Specifically, Delta Lake relies on the following when interacting with storage systems:

- **Atomic visibility**: There must a way for a file to visible in its entirety or not visible at all.
- **Mutual exclusion**: Only one writer must be able to create (or rename) a file at the final destination.
- **Consistent listing**: Once a file has been written in a directory, all future listings for that directory must return that file.

Because storage systems do not necessarily provide all of these guarantees out-of-the-box, Delta Lake transactional operations typically go through the LogStore API instead of accessing the storage system directly. To provide the ACID guarantees for different storage systems, you may have to use different `LogStore` implementations. This article covers how to configure Delta Lake for various storage systems. There are two categories of storage systems:

- **Storage systems with built-in support**: For some storage systems, you do not need additional configurations. Delta Lake uses the scheme of the path (that is, `s3a` in `s3a://path`) to dynamically identify the storage system and use the corresponding `LogStore` implementation that provides the transactional guarantees. However, for S3, there are additional caveats on concurrent writes. See the section on S3 for details.

- **Other storage systems**: The `LogStore`, similar to Apache Spark, uses Hadoop `FileSystem` API to perform reads and writes. So Delta Lake supports concurrent reads on any storage system that provides an implementation of `FileSystem` API. For concurrent writes with transactional guarantees, there are two cases based on the guarantees provided by `FileSystem` implementation. If the implementation provides consistent listing and atomic renames-without-overwrite (that is, `rename(... , overwrite = false)` will either generate the target file atomically or fail if it already exists with `java.nio.file.FileAlreadyExistsException`), then the default `LogStore` implementation using renames will allow concurrent writes with guarantees. Otherwise, you must configure a custom implementation of `LogStore` by setting the following Spark configuration

```
spark.delta.logStore.<scheme>.impl=<full-qualified-class-name>
```

where `<scheme>` is the scheme of the paths of your storage system. This configures Delta Lake to dynamically use the given `LogStore` implementation only for those paths. You can have multiple such configurations for different schemes in your application, thus allowing it to simultaneously read and write from different storage systems.

Note

Delta Lake on local file system may not support concurrent transactional writes. This is because the local file system may or may not provide atomic renames. So you should not use the local file system for testing concurrent writes.

Before version 1.0, Delta Lake supported configuring LogStores by setting `spark.delta.logStore.class`. This approach is now deprecated. Setting this configuration will use the configured `LogStore` for all paths, thereby disabling the dynamic scheme-based delegation.

31.27.1. Microsoft Azure storage

Delta Lake has built-in support for the various Azure storage systems with full transactional guarantees for concurrent reads and writes from multiple clusters.

Delta Lake relies on Hadoop `FileSystem` APIs to access Azure storage services. Specifically, Delta Lake requires the implementation of `FileSystem.rename()` to be atomic, which is only supported in newer Hadoop versions (Hadoop-15156 and Hadoop-15086)). For this reason, you may need to build Spark with newer Hadoop versions and use them for deploying your application. See Specifying the Hadoop Version and Enabling YARN for building Spark with a specific Hadoop version and Quickstart for setting up Spark with Delta Lake.

Here is a list of requirements specific to each type of Azure storage system:

Azure Blob storage

Azure Data Lake Storage Gen1

Azure Data Lake Storage Gen2

Azure Blob storage

Requirements (Azure Blob storage)

- A shared key or shared access signature (SAS)
- Delta Lake 0.2.0 or above
- Hadoop's Azure Blob Storage libraries for deployment with the following versions:

- 2.9.1+ for Hadoop 2
- 3.0.1+ for Hadoop 3
- Apache Spark associated with the corresponding Delta Lake version (see the Quick Start page of the relevant Delta version's documentation) and compiled with Hadoop version that is compatible with the chosen Hadoop libraries.

Configuration (Azure Blob storage)

Here are the steps to configure Delta Lake on Azure Blob storage.

1. Include `hadoop-azure` JAR in the classpath. See the requirements above for version details.

2. Set up credentials.

   You can set up your credentials in the Spark configuration property.

   We recommend that you use a SAS token. In Scala, you can use the following:

```
spark.conf.set(
  "fs.azure.sas.<your-container-name>.<your-storage-account-
name>.blob.core.windows.net",
   "<complete-query-string-of-your-sas-for-the-container>")
```

   Or you can specify an account access key:

```
spark.conf.set(
  "fs.azure.account.key.<your-storage-account-name>.blob.core.windows.net",
   "<your-storage-account-access-key>")
```

Usage (Azure Blob storage)

```
spark.range(5).write.format("delta").save("wasbs://<your-container-name>@<your-storage-
account-name>.blob.core.windows.net/<path-to-delta-table>")
spark.read.format("delta").load("wasbs://<your-container-name>@<your-storage-account-
name>.blob.core.windows.net/<path-to-delta-table>").show()
```

Azure Data Lake Storage Gen1

Requirements (ADLS Gen1)

- A service principal for OAuth 2.0 access
- Delta Lake 0.2.0 or above
- Hadoop's Azure Data Lake Storage Gen1 libraries for deployment with the following versions:
- 2.9.1+ for Hadoop 2
- 3.0.1+ for Hadoop 3
- Apache Spark associated with the corresponding Delta Lake version (see the Quick Start page of the relevant Delta version's documentation) and compiled with Hadoop version that is compatible with the chosen Hadoop libraries.

Configuration (ADLS Gen1)

Here are the steps to configure Delta Lake on Azure Data Lake Storage Gen1.

1. Include `hadoop-azure-datalake` JAR in the classpath. See the requirements above for version details.

2. Set up Azure Data Lake Storage Gen1 credentials.

You can set the following Hadoop configurations with your credentials (in Scala):

```
spark.conf.set("dfs.adls.oauth2.access.token.provider.type", "ClientCredential")
spark.conf.set("dfs.adls.oauth2.client.id", "<your-oauth2-client-id>")
spark.conf.set("dfs.adls.oauth2.credential", "<your-oauth2-credential>")
spark.conf.set("dfs.adls.oauth2.refresh.url", "https://login.microsoftonline.com/<your-directory-id>/oauth2/token")
```

Usage (ADLS Gen1)

```
spark.range(5).write.format("delta").save("adl://<your-adls-account>.azuredatalakestore.net/<path-to-delta-table>")

spark.read.format("delta").load("adl://<your-adls-account>.azuredatalakestore.net/<path-to-delta-table>").show()
```

Azure Data Lake Storage Gen2

Requirements (ADLS Gen2)

- Account created in Azure Data Lake Storage Gen2)
- Service principal created and assigned the Storage Blob Data Contributor role for the storage account.
- Note the storage-account-name, directory-id (also known as tenant-id), application-id, and password of the principal. These will be used for configuring Spark.
- Delta Lake 0.7.0 or above
- Apache Spark 3.0 or above
- Apache Spark used must be built with Hadoop 3.2 or above.

Configuration (ADLS Gen2)

Here are the steps to configure Delta Lake on Azure Data Lake Storage Gen1.

1. Include the JAR of the Maven artifact `hadoop-azure-datalake` in the classpath. See the requirements for version details. In addition, you may also have to include JARs for Maven artifacts `hadoop-azure` and `wildfly-openssl`.

2. Set up Azure Data Lake Storage Gen2 credentials.

```
spark.conf.set("fs.azure.account.auth.type.<storage-account-name>.dfs.core.windows.net", "OAuth")
 spark.conf.set("fs.azure.account.oauth.provider.type.<storage-account-name>.dfs.core.windows.net",  "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider")
 spark.conf.set("fs.azure.account.oauth2.client.id.<storage-account-name>.dfs.core.windows.net", "<application-id>")
 spark.conf.set("fs.azure.account.oauth2.client.secret.<storage-account-name>.dfs.core.windows.net","<password>")
 spark.conf.set("fs.azure.account.oauth2.client.endpoint.<storage-account-name>.dfs.core.windows.net", "https://login.microsoftonline.com/<directory-id>/oauth2/token")
```

where `<storage-account-name>`, `<application-id>`, `<directory-id>` and `<password>` are details of the service principal we set as requirements earlier.

- `<scope>` with the Databricks secret scope name.
- `<service-credential-key>` with the name of the key containing the client secret.

- <storage-account> with the name of the Azure storage account.
- <application-id> with the **Application (client) ID** for the Azure Active Directory application.
- <directory-id> with the **Directory (tenant) ID** for the Azure Active Directory application.

3. Initialize the file system if needed

```
spark.conf.set("fs.azure.createRemoteFileSystemDuringInitialization", "true")
dbutils.fs.ls("abfss://<container-name>@<storage-account-name>.dfs.core.windows.net/")
spark.conf.set("fs.azure.createRemoteFileSystemDuringInitialization", "false")
```

Usage (ADLS Gen2)

```
spark.range(5).write.format("delta").save("abfss://<container-name>@<storage-account-name>.dfs.core.windows.net/<path-to-delta-table>")

spark.read.format("delta").load("abfss://<container-name>@<storage-account-name>.dfs.core.windows.net/<path-to-delta-table>").show()
```

where <container-name> is the file system name under the container.

```
CREATE TABLE <database-name>.<table-name>;

COPY INTO <database-name>.<table-name>
FROM 'abfss://container@storageAccount.dfs.core.windows.net/path/to/folder'
FILEFORMAT = CSV
COPY_OPTIONS ('mergeSchema' = 'true');
```

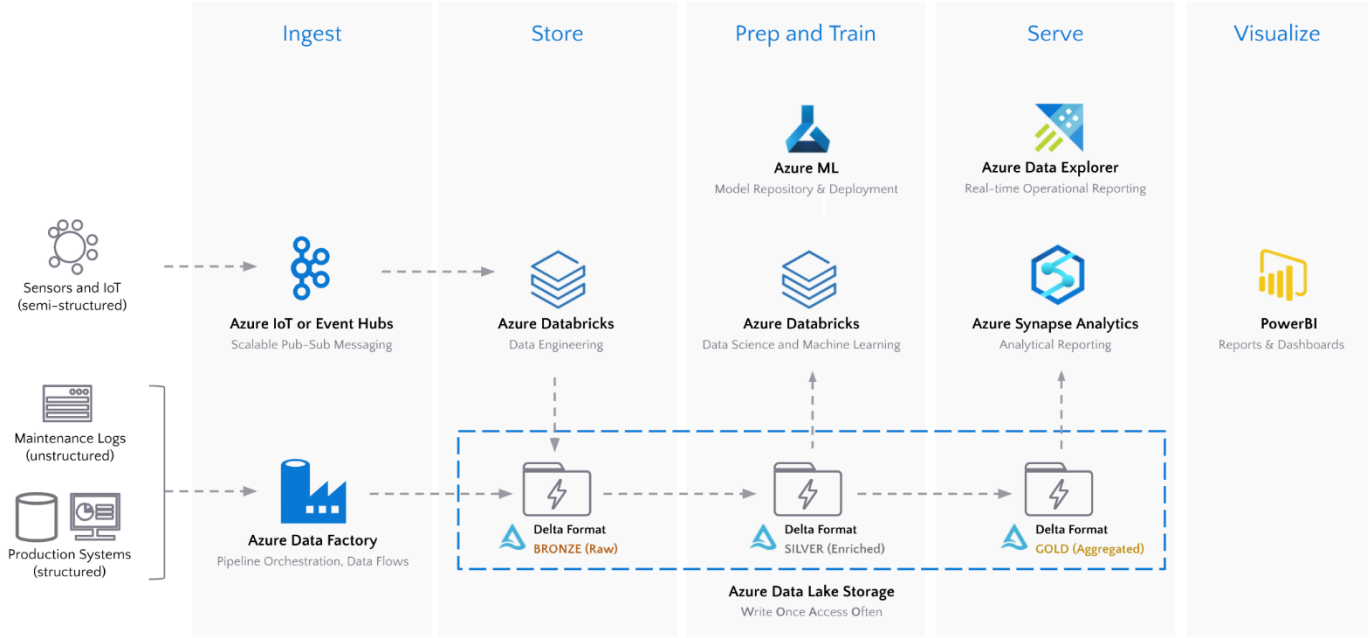
HDFS

Delta Lake has built-in support for HDFS with full transactional guarantees on concurrent reads and writes from multiple clusters. See Hadoop and Spark documentation for configuring credentials.

32. End to End Industrial IoT (IIoT) on Azure Databricks

Part 1: Data Engineering

This notebook demonstrates the following architecture for IIoT Ingest, Processing and Analytics on Azure. The following architecture is implemented for the demo.



The notebook is broken into sections following these steps:

1. **Data Ingest** - stream real-time raw sensor data from Azure IoT Hubs into the Delta format in Azure Storage
2. **Data Processing** - stream process sensor data from raw (Bronze) to silver (aggregated) to gold (enriched) Delta tables on Azure Storage

```
# AzureML Workspace info (name, region, resource group and subscription ID) for model deployment

dbutils.widgets.text("Storage Account","<your ADLS Gen 2 account name>","Storage Account")

dbutils.widgets.text("Event Hub Name","<your IoT Hub's Event Hub Compatible Name>","Event Hub Name")
```

Step 1- Environment Setup

The pre-requisites are listed below:

Azure Services Required

- Azure IoT Hub
- Azure IoT Simulator running with the code provided in this github repo and configured for your IoT Hub
- ADLS Gen 2 Storage account with a container called `iot`
- Azure Synapse SQL Pool call `iot`

Azure Databricks Configuration Required

- 3-node (min) Databricks Cluster running **DBR 7.0+** and the following libraries:
    - **Azure Event Hubs Connector for Databricks** - Maven coordinates `com.microsoft.azure:azure-eventhubs-spark_2.12:2.3.17`
- The following Secrets defined in scope `iot`
    - `iothub-cs` - Connection string for your IoT Hub **(Important - use the Event Hub Compatible connection string)**
    - `adls_key` - Access Key to ADLS storage account **(Important - use the Access Key)**
    - `synapse_cs` - JDBC connect string to your Synapse SQL Pool **(Important - use the SQL Authentication with username/password connection string)**
- The following notebook widgets populated:
    - `Storage Account` - Name of your storage account

```
# Setup access to storage account for temp data when pushing to Synapse

storage_account = dbutils.widgets.get("Storage Account")

spark.conf.set(f"fs.azure.account.key.{storage_account}.dfs.core.windows.net",
dbutils.secrets.get("iot","adls_key"))


# Setup storage locations for all data

ROOT_PATH = f"abfss://iot@{storage_account}.dfs.core.windows.net/"

BRONZE_PATH = ROOT_PATH + "bronze/"

SILVER_PATH = ROOT_PATH + "silver/"

GOLD_PATH = ROOT_PATH + "gold/"

SYNAPSE_PATH = ROOT_PATH + "synapse/"

CHECKPOINT_PATH = ROOT_PATH + "checkpoints/"


# Other initializations

IOT_CS = "Endpoint=sb://iothub-ns-sguptaioth-4012358-
1c55ddfc30.servicebus.windows.net/;SharedAccessKeyName=iothubowner;SharedAccessKey=Lcr
LjsLZKxjdzYklb4Dp2egNnKwjKLveywWUhVNIJyM=;EntityPath=sguptaiothub" #
dbutils.secrets.get('iot','iothub-cs') # IoT Hub connection string (Event Hub
Compatible)

ehConf = {
```

```
'eventhubs.connectionString':sc._jvm.org.apache.spark.eventhubs.EventHubsUtils.encrypt
(IOT_CS),
  'ehName':dbutils.widgets.get("Event Hub Name")
}

# Enable auto compaction and optimized writes in Delta
spark.conf.set("spark.databricks.delta.optimizeWrite.enabled","true")
spark.conf.set("spark.databricks.delta.autoCompact.enabled","true")

# Pyspark and ML Imports
import os, json, requests
from pyspark.sql import functions as F
from pyspark.sql.functions import pandas_udf, PandasUDFType
# Make sure root path is empty
dbutils.fs.rm(ROOT_PATH, True)
Out[2]: True
%sql
-- Clean up tables & views
DROP TABLE IF EXISTS turbine_raw;
DROP TABLE IF EXISTS weather_raw;
DROP TABLE IF EXISTS turbine_agg;
DROP TABLE IF EXISTS weather_agg;
DROP TABLE IF EXISTS turbine_enriched;
DROP TABLE IF EXISTS turbine_power;
DROP TABLE IF EXISTS turbine_maintenance;
DROP VIEW IF EXISTS turbine_combined;
DROP VIEW IF EXISTS feature_view;
DROP TABLE IF EXISTS turbine_life_predictions;
DROP TABLE IF EXISTS turbine_power_predictions;
OK
```

Step 2- Data Ingest from IoT Hubs

Azure Databricks provides a native connector to IoT and Event Hubs. Below, we will use PySpark Structured Streaming to read from an IoT Hub stream of data and write the data in it's raw format directly into Delta.

Make sure that your IoT Simulator is sending payloads to IoT Hub as shown below.

We have two separate types of data payloads in our IoT Hub:

1. **Turbine Sensor readings** - this payload contains `date`,`timestamp`,`deviceid`,`rpm` and `angle` fields
2. **Weather Sensor readings** - this payload contains `date`,`timestamp`,`temperature`,`humidity`,`windspeed`, and `winddirection` fields

We split out the two payloads into separate streams and write them both into Delta locations on Azure Storage. We are able to query these two Bronze tables *immediately* as the data streams in.

```
# Schema of incoming data from IoT hub

schema = "timestamp timestamp, deviceId string, temperature double, humidity double, windspeed double, winddirection string, rpm double, angle double"


# Read directly from IoT Hub using the EventHubs library for Databricks

iot_stream = (

  spark.readStream.format("eventhubs")                                                # Read from IoT Hubs directly

    .options(**ehConf)                                                                # Use the Event-Hub-enabled connect string

    .load()                                                                           # Load the data

    .withColumn('reading', F.from_json(F.col('body').cast('string'), schema))        # Extract the "body" payload from the messages

    .select('reading.*', F.to_date('reading.timestamp').alias('date'))               # Create a "date" field for partitioning

)


# Split our IoT Hub stream into separate streams and write them both into their own Delta locations

write_turbine_to_delta = (

  iot_stream.filter('temperature is null')                                            # Filter out turbine telemetry from other data streams

    .select('date','timestamp','deviceId','rpm','angle')                             # Extract the fields of interest

    .writeStream.format('delta')                                                      # Write our stream to the Delta format
```

```
    .partitionBy('date')                                                                    #
Partition our data by Date for performance
    .option("checkpointLocation", CHECKPOINT_PATH + "turbine_raw")                          #
Checkpoint so we can restart streams gracefully
    .start(BRONZE_PATH + "turbine_raw")                                                     #
Stream the data into an ADLS Path
)


write_weather_to_delta = (
  iot_stream.filter(iot_stream.temperature.isNotNull())                                     #
Filter out weather telemetry only

.select('date','deviceid','timestamp','temperature','humidity','windspeed','winddirect
ion')
    .writeStream.format('delta')                                                            #
Write our stream to the Delta format
    .partitionBy('date')                                                                    #
Partition our data by Date for performance
    .option("checkpointLocation", CHECKPOINT_PATH + "weather_raw")                          #
Checkpoint so we can restart streams gracefully
    .start(BRONZE_PATH + "weather_raw")                                                     #
Stream the data into an ADLS Path
)


# Create the external tables once data starts to stream in
while True:
  try:
    spark.sql(f'CREATE TABLE IF NOT EXISTS turbine_raw USING DELTA LOCATION
"{BRONZE_PATH + "turbine_raw"}"')
    spark.sql(f'CREATE TABLE IF NOT EXISTS weather_raw USING DELTA LOCATION
"{BRONZE_PATH + "weather_raw"}"')
    break
  except:
    pass
%sql
-- We can query the data directly from storage immediately as soon as it starts
streams into Delta
SELECT * FROM turbine_raw WHERE deviceid = 'WindTurbine-1'
```

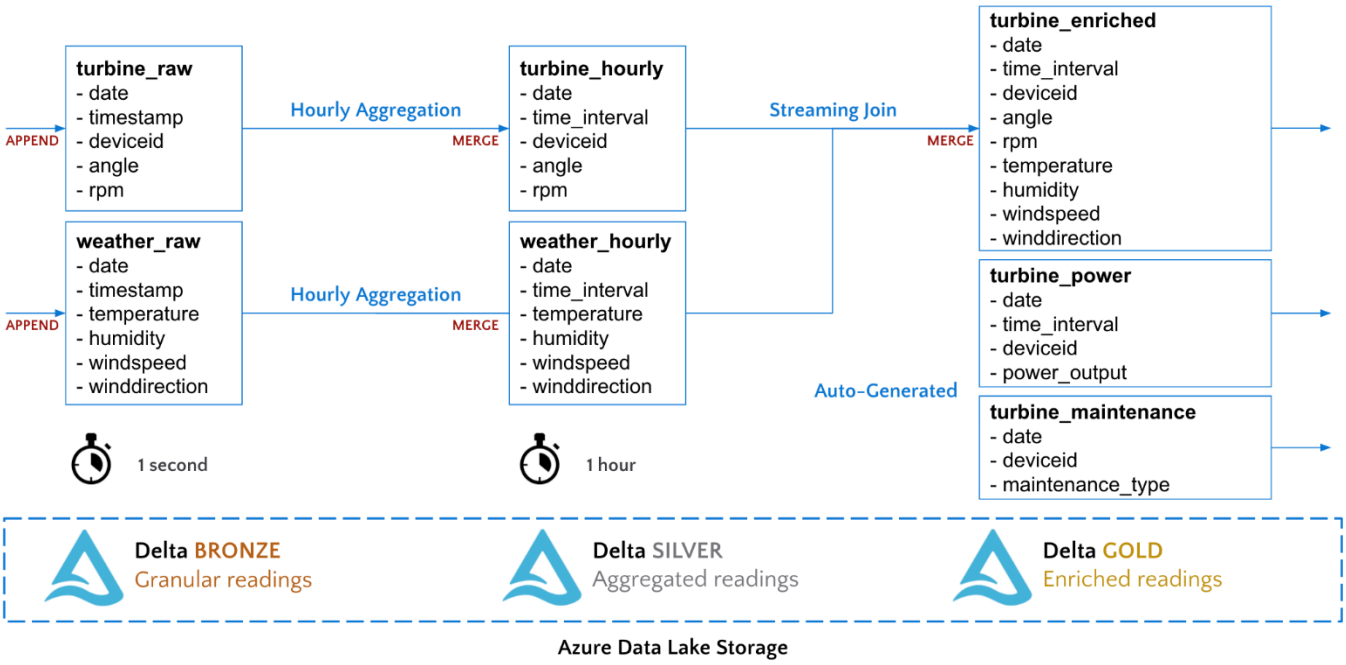
15:58:19Aug 20, 202015:58:2015:58:2115:58:2215:58:2315:58:2415:58:2515:58:2615:58:2715:58:2815:58:2915:58:3015:58:3115:58:3215:58:3315:58:3415:58:3515:58:3615:58:3715:58:388.458.58.558.68.658.7

WindTurbine-1 angleWindTurbine-1 rpmtimestampangle, rpmdeviceId

Step 2- Data Processing in Delta

While our raw sensor data is being streamed into Bronze Delta tables on Azure Storage, we can create streaming pipelines on this data that flow it through Silver and Gold data sets.

We will use the following schema for Silver and Gold data sets:



2a. Delta Bronze (Raw) to Delta Silver (Aggregated)

The first step of our processing pipeline will clean and aggregate the measurements to 1 hour intervals.

Since we are aggregating time-series values and there is a likelihood of late-arriving data and data changes, we will use the **MERGE** functionality of Delta to upsert records into target tables.

MERGE allows us to upsert source records into a target storage location. This is useful when dealing with time-series data as:

1. Data often arrives late and requires aggregation states to be updated
2. Historical data needs to be backfilled while streaming data is feeding into the table

When streaming source data, `foreachBatch()` can be used to perform a merges on micro-batches of data.

```
# Create functions to merge turbine and weather data into their target Delta tables
def merge_delta(incremental, target):

incremental.dropDuplicates(['date','window','deviceid']).createOrReplaceTempView("incr
emental")

  try:
    # MERGE records into the target table using the specified join key
```

```
    incremental._jdf.sparkSession().sql(f"""
      MERGE INTO delta.`{target}` t
      USING incremental i
      ON i.date=t.date AND i.window = t.window AND i.deviceId = t.deviceid
      WHEN MATCHED THEN UPDATE SET *
      WHEN NOT MATCHED THEN INSERT *
    """)
  except:
    # If the †arget table does not exist, create one
    incremental.write.format("delta").partitionBy("date").save(target)


turbine_b_to_s = (
  spark.readStream.format('delta').table("turbine_raw")                          # Read
data as a stream from our source Delta table
    .groupBy('deviceId','date',F.window('timestamp','5 minutes'))                #
Aggregate readings to hourly intervals
    .agg(F.avg('rpm').alias('rpm'), F.avg("angle").alias("angle"))
    .writeStream                                                                 # Write
the resulting stream
    .foreachBatch(lambda i, b: merge_delta(i, SILVER_PATH + "turbine_agg"))    # Pass
each micro-batch to a function
    .outputMode("update")                                                        # Merge
works with update mode
    .option("checkpointLocation", CHECKPOINT_PATH + "turbine_agg")               #
Checkpoint so we can restart streams gracefully
    .start()
)


weather_b_to_s = (
  spark.readStream.format('delta').table("weather_raw")                          # Read
data as a stream from our source Delta table
    .groupBy('deviceid','date',F.window('timestamp','5 minutes'))               #
Aggregate readings to hourly intervals

.agg({"temperature":"avg","humidity":"avg","windspeed":"avg","winddirection":"last"})
    .selectExpr('date','window','deviceid','`avg(temperature)` as
temperature','`avg(humidity)` as humidity',
                '`avg(windspeed)` as windspeed','`last(winddirection)` as
winddirection')
    .writeStream                                                                 # Write
the resulting stream
```

```
    .foreachBatch(lambda i, b: merge_delta(i, SILVER_PATH + "weather_agg"))    # Pass each micro-batch to a function
    .outputMode("update")                                                      # Merge works with update mode
    .option("checkpointLocation", CHECKPOINT_PATH + "weather_agg")             # Checkpoint so we can restart streams gracefully
    .start()
)


# Create the external tables once data starts to stream in
while True:
  try:
    spark.sql(f'CREATE TABLE IF NOT EXISTS turbine_agg USING DELTA LOCATION "{SILVER_PATH + "turbine_agg"}"')
    spark.sql(f'CREATE TABLE IF NOT EXISTS weather_agg USING DELTA LOCATION "{SILVER_PATH + "weather_agg"}"')
    break
  except:
    pass
```

5e6aaaf8-f40a-4bf8-9ad7-20592a38c17b

Last updated: 713 days ago

85681632-35be-4438-b044-950572a77664

Last updated: 713 days ago

```
py4j.Py4JException: An exception was raised by the Python Proxy. Return Message: Traceback (most recent call last):
```

```
%sql
-- As data gets merged in real-time to our hourly table, we can query it immediately
SELECT * FROM turbine_agg t JOIN weather_agg w ON (t.date=w.date AND t.window=w.window) WHERE t.deviceid='WindTurbine-1' ORDER BY t.window DESC
```

2020-08-07T23:25:00.000+0000,2020-08-07T23:30:00.000+00002020-08-07T23:20:00.000+0000,2020-08-07T23:25:00.000+0000012345678

rpmwindspeedwindowrpm, windspeed

2b. Delta Silver (Aggregated) to Delta Gold (Enriched)

Next we perform a streaming join of weather and turbine readings to create one enriched dataset we can use for data science and model training.

```
# Read streams from Delta Silver tables and join them together on common columns (date & window)
turbine_agg = spark.readStream.format('delta').option("ignoreChanges", True).table('turbine_agg')
```

```
weather_agg = spark.readStream.format('delta').option("ignoreChanges", True).table('weather_agg').drop('deviceid')
turbine_enriched = turbine_agg.join(weather_agg, ['date','window'])


# Write the stream to a foreachBatch function which performs the MERGE as before
merge_gold_stream = (
  turbine_enriched
    .selectExpr('date','deviceid','window.start as window','rpm','angle','temperature','humidity','windspeed','winddirection')
    .writeStream
    .foreachBatch(lambda i, b: merge_delta(i, GOLD_PATH + "turbine_enriched"))
    .option("checkpointLocation", CHECKPOINT_PATH + "turbine_enriched")
    .start()
)


# Create the external tables once data starts to stream in
while True:
  try:
    spark.sql(f'CREATE TABLE IF NOT EXISTS turbine_enriched USING DELTA LOCATION "{GOLD_PATH + "turbine_enriched"}"')
    break
  except:
    pass
```

23e5c17c-41bf-45eb-a127-d8659836006e

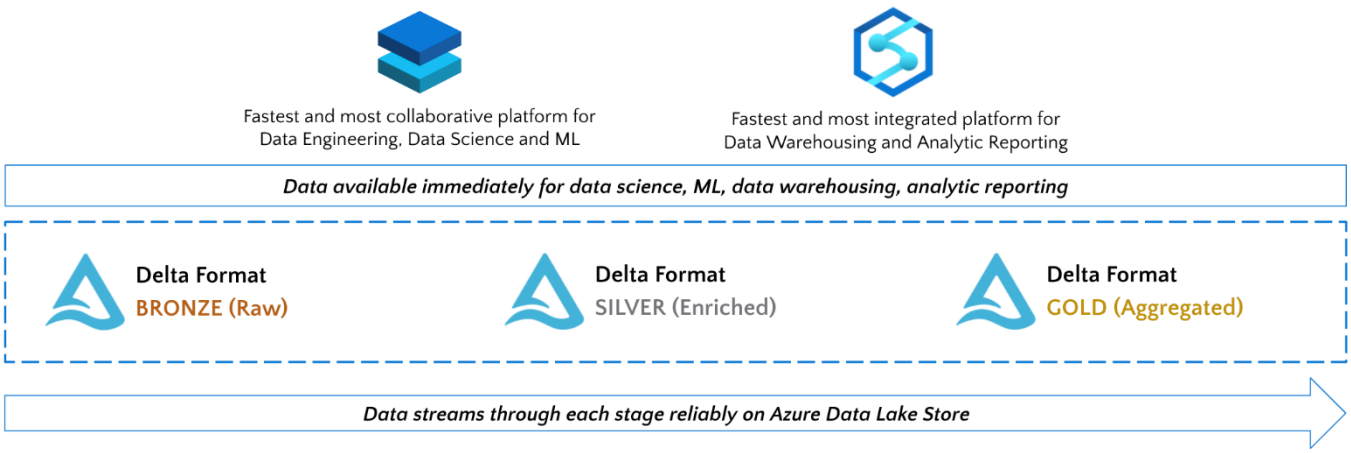
Last updated: 713 days ago

```
%sql SELECT * FROM turbine_enriched WHERE deviceid='WindTurbine-1'
```

Showing the first 1000 rows.

2c: Stream Delta GOLD Table to Synapse

Synapse Analytics provides on-demand SQL directly on Data Lake source formats. Databricks can also directly stream data to Synapse SQL Pools for Data Warehousing workloads like BI dashboarding and reporting.




```
spark.conf.set("spark.databricks.sqldw.writeSemantics", "copy")
# Use COPY INTO for faster loads to Synapse from Databricks


write_to_synapse = (


spark.readStream.format('delta').option('ignoreChanges',True).table('turbine_enriched'
) # Read in Gold turbine readings from Delta as a stream

    .writeStream.format("com.databricks.spark.sqldw")
# Write to Synapse (SQL DW connector)

    .option("url",dbutils.secrets.get("iot","synapse_cs"))
# SQL Pool JDBC connection (with SQL Auth) string

    .option("tempDir", SYNAPSE_PATH)
# Temporary ADLS path to stage the data (with forwarded permissions)

    .option("forwardSparkAzureStorageCredentials", "true")

    .option("dbTable", "turbine_enriched")
# Table in Synapse to write to

    .option("checkpointLocation", CHECKPOINT_PATH+"synapse")
# Checkpoint for resilient streaming

    .start()

)
```

ef808c48-f19c-4776-a934-0fc601b72402

Last updated: 713 days ago

2d. Backfill Historical Data

In order to train a model, we will need to backfill our streaming data with historical data. The cell below generates 1 year of historical hourly turbine and weather data and inserts it into our Gold Delta table.

```
import pandas as pd

import numpy as np


# Function to simulate generating time-series data given a baseline, slope, and some
seasonality
```

```
def generate_series(time_index, baseline, slope=0.01, period=365*24*12):
  rnd = np.random.RandomState(time_index)
  season_time = (time_index % period) / period
  seasonal_pattern = np.where(season_time < 0.4, np.cos(season_time * 2 * np.pi), 1 / np.exp(3 * season_time))
  return baseline * (1 + 0.1 * seasonal_pattern + 0.1 * rnd.randn(len(time_index)))

# Get start and end dates for our historical data
dates = spark.sql('select max(date)-interval 365 days as start, max(date) as end from turbine_enriched').toPandas()

# Get the baseline readings for each sensor for backfilling data
turbine_enriched_pd = spark.table('turbine_enriched').toPandas()
baselines = turbine_enriched_pd.min()[3:8]
devices = turbine_enriched_pd['deviceid'].unique()

# Iterate through each device to generate historical data for that device
print("---Generating Historical Enriched Turbine Readings---")
for deviceid in devices:
  print(f'Backfilling device {deviceid}')
  windows = pd.date_range(start=dates['start'][0], end=dates['end'][0], freq='5T') # Generate a list of hourly timestamps from start to end date
  historical_values = pd.DataFrame({
    'date': windows.date,
    'window': windows,
    'winddirection': np.random.choice(['N','NW','W','SW','S','SE','E','NE'], size=len(windows)),
    'deviceId': deviceid
  })
  time_index = historical_values.index.to_numpy()                                    # Generate a time index

  for sensor in baselines.keys():
    historical_values[sensor] = generate_series(time_index, baselines[sensor])    # Generate time-series data from this sensor

  # Write dataframe to enriched_readings Delta table

spark.createDataFrame(historical_values).write.format("delta").mode("append").saveAsTable("turbine_enriched")
```

```
# Create power readings based on weather and operating conditions

print("---Generating Historical Turbine Power Readings---")

spark.sql(f'CREATE TABLE turbine_power USING DELTA PARTITIONED BY (date) LOCATION "{GOLD_PATH + "turbine_power"}" AS SELECT date, window, deviceId, 0.1 * (temperature/humidity) * (3.1416 * 25) * windspeed * rpm AS power FROM turbine_enriched')


# Create a maintenance records based on peak power usage

print("---Generating Historical Turbine Maintenance Records---")

spark.sql(f'CREATE TABLE turbine_maintenance USING DELTA LOCATION "{GOLD_PATH + "turbine_maintenance"}" AS SELECT DISTINCT deviceid, FIRST(date) OVER (PARTITION BY deviceid, year(date), month(date) ORDER BY power) AS date, True AS maintenance FROM turbine_power')
```

```
---Generating Historical Enriched Turbine Readings--- Backfilling device WindTurbine-0 Backfilling device WindTurbine-7 Backfilling device WindTurbine-1 Backfilling device WindTurbine-9 Backfilling device WindTurbine-6 Backfilling device WindTurbine-2 Backfilling device WindTurbine-3 Backfilling device WindTurbine-8 Backfilling device WindTurbine-5 Backfilling device WindTurbine-4 ---Generating Historical Turbine Power Readings --- ---Generating Historical Turbine Maintenance Records--- Out[9]: DataFrame[]
```

```
%sql

-- Optimize all 3 tables for querying and model training performance

OPTIMIZE turbine_enriched WHERE date<current_date() ZORDER BY deviceid, window;

OPTIMIZE turbine_power ZORDER BY deviceid, window;

OPTIMIZE turbine_maintenance ZORDER BY deviceid;
```

path	metrics
1	null

{"numFilesAdded": 0, "numFilesRemoved": 0, "filesAdded": {"min": null, "max": null, "avg": 0, "totalFiles": 0, "totalSize": 0}, "filesRemoved": {"min": null, "max": null, "avg": 0, "totalFiles": 0, "totalSize": 0}, "partitionsOptimized": 0, "zOrderStats": {"strategyName": "minCubeSize(107374182400)", "inputCubeFiles": {"num": 0, "size": 0}, "inputOtherFiles": {"num": 1, "size": 1221}, "inputNumCubes": 0, "mergedFiles": {"num": 0, "size": 0}, "mergedNumCubes": 0}, "numBatches": 0}

Showing all 1 rows.

Our Delta Gold tables are now ready for predictive analytics! We now have hourly weather, turbine operating and power measurements, and daily maintenance logs going back one year. We can see that there is significant correlation between most of the variables.

```
%sql

-- Query all 3 tables

CREATE OR REPLACE VIEW gold_readings AS

SELECT r.*,

  p.power,

  ifnull(m.maintenance,False) as maintenance
```

```
FROM turbine_enriched r
  JOIN turbine_power p ON (r.date=p.date AND r.window=p.window AND
r.deviceid=p.deviceid)
  LEFT JOIN turbine_maintenance m ON (r.date=m.date AND r.deviceid=m.deviceid);
```

```
SELECT * FROM gold_readings ORDER BY deviceid, window
```

Showing the first 1000 rows.

Benefits of Delta Lake on Time-Series Data

A key component of this architecture is the Azure Data Lake Store (ADLS), which enables the write-once, access-often analytics pattern in Azure. However, Data Lakes alone do not solve challenges that come with time-series streaming data. The Delta storage format provides a layer of resiliency and performance on all data sources stored in ADLS. Specifically for time-series data, Delta provides the following advantages over other storage formats on ADLS:

Required Capability	Other formats on ADLS	Delta Format on ADLS
Unified batch & streaming	Data Lakes are often used in conjunction with a streaming store like CosmosDB, resulting in a complex architecture	ACID-compliant transactions enable data engineers to perform streaming ingest and historically batch loads into the same locations on ADLS
Schema enforcement and evolution	Data Lakes do not enforce schema, requiring all data to be pushed into a relational database for reliability	Schema is enforced by default. As new IoT devices are added to the data stream, schemas can be evolved safely so downstream applications don't fail
Efficient Upserts	Data Lakes do not support in-line updates and merges, requiring deletion and insertions of entire partitions to perform updates	MERGE commands are effective for situations handling delayed IoT readings, modified dimension tables used for real-time enrichment, or if data needs to be reprocessed
File Compaction	Streaming time-series data into Data Lakes generate hundreds or even thousands of tiny files	Auto-compaction in Delta optimizes the file sizes to increase throughput and parallelism
Multi-dimensional clustering	Data Lakes provide push-down filtering on partitions only	ZORDERing time-series on fields like timestamp or sensor ID allows Databricks to filter and join on those columns up to 100x faster than simple partitioning techniques

DELTA LAKE
WITH SPARK SQL

Delta Lake is an open source storage layer that brings ACID transactions to Apache Spark™ and big data workloads.

delta.io | Documentation | GitHub | Delta Lake on Databricks

CREATE AND QUERY DELTA TABLES

Create and use managed database

```
-- Managed database is saved in the Hive metastore.
Default database is named "default".
DROP DATABASE IF EXISTS dbName;
CREATE DATABASE dbName;
USE dbName  -- This command avoids having to specify
dbName.tableName every time instead of just tableName.
```

Query Delta Lake table by table name (preferred)

```
/* You can refer to Delta Tables by table name, or by
path. Table name is the preferred way, since named tables
are managed in the Hive Metastore (i.e., when you DROP a
named table, the data is dropped also — not the case for
path-based tables.) */
SELECT * FROM [dbName.] tableName
```

Query Delta Lake table by path

```
SELECT * FROM delta.`path/to/delta_table` -- note backticks
```

Convert Parquet table to Delta Lake format in place

```
-- by table name
CONVERT TO DELTA [dbName.]tableName
[PARTITIONED BY (col_name1 col_type1, col_name2
col_type2)]

-- path-based tables
CONVERT TO DELTA parquet.`/path/to/table` -- note backticks
[PARTITIONED BY (col_name1 col_type1, col_name2 col_type2)]
```

Create Delta Lake table as SELECT * with no upfront schema definition

```
CREATE TABLE [dbName.] tableName
USING DELTA
AS SELECT * FROM tableName | parquet.`path/to/data`
[LOCATION `/path/to/table`]
-- using location = unmanaged table
```

Create table, define schema explicitly with SQL DDL

```
CREATE TABLE [dbName.] tableName (
    id INT [NOT NULL],
    name STRING,
    date DATE,
    int_rate FLOAT)
USING DELTA
[PARTITIONED BY (time, date)] -- optional
```

Copy new data into Delta Lake table (with idempotent retries)

```
COPY INTO [dbName.] targetTable
  FROM "/path/to/table"
  FILEFORMAT = DELTA  -- or CSV, Parquet, ORC, JSON, etc.
```

DELTA LAKE DDL/DML: UPDATE, DELETE, MERGE, ALTER TABLE

Update rows that match a predicate condition

```
UPDATE tableName SET event = 'click' WHERE event = 'clk'
```

Delete rows that match a predicate condition

```
DELETE FROM tableName WHERE "date < '2017-01-01"
```

Insert values directly into table

```
INSERT INTO TABLE tableName VALUES (
    (8003, "Kim Jones", "2020-12-18", 3.875),
    (8004, "Tim Jones", "2020-12-20", 3.750)
);
-- Insert using SELECT statement
INSERT INTO tableName SELECT * FROM sourceTable
-- Atomically replace all data in table with new values
INSERT OVERWRITE loan_by_state_delta VALUES (...)
```

Upsert (update + insert) using MERGE

```
MERGE INTO target
USING updates
ON target.Id = updates.Id
WHEN MATCHED AND target.delete_flag = "true" THEN
  DELETE
WHEN MATCHED THEN
  UPDATE SET *  -- star notation means all columns
WHEN NOT MATCHED THEN
  INSERT (date, Id, data) -- or, use INSERT *
  VALUES (date, Id, data)
```

Insert with Deduplication using MERGE

```
MERGE INTO logs
USING newDedupedLogs
ON logs.uniqueId = newDedupedLogs.uniqueId
WHEN NOT MATCHED
  THEN INSERT *
```

Alter table schema — add columns

```
ALTER TABLE tableName ADD COLUMNS (
    col_name data_type
    [FIRST|AFTER colA_name])
```

Alter table — add constraint

```
-- Add "Not null" constraint:
ALTER TABLE tableName CHANGE COLUMN col_name SET NOT NULL
-- Add "Check" constraint:
ALTER TABLE tableName
ADD CONSTRAINT dateWithinRange CHECK date > "1900-01-01"
-- Drop constraint:
ALTER TABLE tableName DROP CONSTRAINT dateWithinRange
```

TIME TRAVEL

View transaction log (aka Delta Log)

```
DESCRIBE HISTORY tableName
```

Query historical versions of Delta Lake tables

```
SELECT * FROM tableName VERSION AS OF 0
SELECT * FROM tableName@v0 -- equivalent to VERSION AS OF 0
SELECT * FROM tableName TIMESTAMP AS OF "2020-12-18"
```

Find changes between 2 versions of table

```
SELECT * FROM tableName VERSION AS OF 12
EXCEPT ALL SELECT * FROM tableName VERSION AS OF 11
```

TIME TRAVEL (CONTINUED)

Rollback a table to an earlier version

```
-- RESTORE requires Delta Lake version 0.7.0+ & DBR 7.4+.
RESTORE tableName VERSION AS OF 0
RESTORE tableName TIMESTAMP AS OF "2020-12-18"
```

UTILITY METHODS

View table details

```
DESCRIBE DETAIL tableName
DESCRIBE FORMATTED tableName
```

Delete old files with Vacuum

```
VACUUM tableName [RETAIN num HOURS] [DRY RUN]
```

Clone a Delta Lake table

```
-- Deep clones copy data from source, shallow clones don't.
CREATE TABLE [dbName.] targetName
[SHALLOW | DEEP] CLONE sourceName [VERSION AS OF 0]
[LOCATION "path/to/table"]
-- specify location only for path-based tables
```

Interoperability with Python / DataFrames

```
-- Read name-based table from Hive metastore into DataFrame
df = spark.table("tableName")
-- Read path-based table into DataFrame
df = spark.read.format("delta").load("/path/to/delta_table")
```

Run SQL queries from Python

```
spark.sql("SELECT * FROM tableName")
spark.sql("SELECT * FROM delta.`/path/to/delta_table`")
```

Modify data retention settings for Delta Lake table

```
-- logRetentionDuration -> how long transaction log history
is kept, deletedFileRetentionDuration -> how long ago a file
must have been deleted before being a candidate for VACCUM.
ALTER TABLE tableName
SET TBLPROPERTIES(
    delta.logRetentionDuration = "interval 30 days",
    delta.deletedFileRetentionDuration = "interval 7 days"
);
SHOW TBLPROPERTIES tableName;
```

PERFORMANCE OPTIMIZATIONS

Compact data files with Optimize and Z-Order

```
*Databricks Delta Lake feature
OPTIMIZE tableName
[ZORDER BY (colNameA, colNameB)]
```

Auto-optimize tables

```
*Databricks Delta Lake feature
ALTER TABLE [table_name | delta.`path/to/delta_table`]
SET TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true)
```

Cache frequently queried data in Delta Cache

```
*Databricks Delta Lake feature
CACHE SELECT * FROM tableName
-- or:
CACHE SELECT colA, colB FROM tableName WHERE colNameA > 0
```


databricks

DELTA LAKE WITH PYTHON

Delta Lake is an open source storage layer that brings ACID transactions to Apache Spark™ and big data workloads.

delta.io | Documentation | GitHub | API reference | Databricks

READS AND WRITES WITH DELTA LAKE

Read data from pandas DataFrame

```
df = spark.createDataFrame(pdf)
# where pdf is a pandas DF
# then save DataFrame in Delta Lake format as shown below
```

Read data using Apache Spark™

```
# read by path
df = (spark.read.format("parquet"|"csv"|"json"|etc.)
        .load("/path/to/delta_table"))
# read by table name
df = spark.table("events")
```

Save DataFrame in Delta Lake format

```
(df.write.format("delta")
  .mode("append"|"overwrite")
  .partitionBy("date")  # optional
  .option("mergeSchema", "true")  # option - evolve schema
  .saveAsTable("events") | .save("/path/to/delta_table")
)
```

Streaming reads (Delta table as streaming source)

```
# by path or by table name
df = (spark.readStream
  .format("delta")
  .schema(schema)
  .table("events") | .load("/delta/events")
)
```

Streaming writes (Delta table as a sink)

```
(df.writeStream.format("delta")
  .outputMode("append"|"update"|"complete")
  .option("checkpointLocation", "/path/to/checkpoints")
  .trigger(once=True|processingTime="10 seconds")
  .table("events") | .start("/delta/events")
)
```

CONVERT PARQUET TO DELTA LAKE

Convert Parquet table to Delta Lake format in place

```
from delta.tables import *

deltaTable = DeltaTable.convertToDelta(spark,
"parquet.`/path/to/parquet_table`")

partitionedDeltaTable = DeltaTable.convertToDelta(spark,
"parquet.`/path/to/parquet_table`", "part int")
```

Provided to the open source community by Databricks

WORKING WITH DELTA TABLES

```
# A DeltaTable is the entry point for interacting with
tables programmatically in Python — for example, to
perform updates or deletes.
from delta.tables import *

deltaTable = DeltaTable.forName(spark, tableName)
deltaTable = DeltaTable.forPath(spark,
delta.`path/to/table`)
```

DELTA LAKE DDL/DML: UPDATES, DELETES, INSERTS, MERGES

Delete rows that match a predicate condition

```
# predicate using SQL formatted string
deltaTable.delete("date < '2017-01-01'")
# predicate using Spark SQL functions
deltaTable.delete(col("date") < "2017-01-01")
```

Update rows that match a predicate condition

```
# predicate using SQL formatted string
deltaTable.update(condition = "eventType = 'clk'",
          set = { "eventType": "'click'" } )
# predicate using Spark SQL functions
deltaTable.update(condition = col("eventType") == "clk",
          set = { "eventType": lit("click") } )
```

Upsert (update + insert) using MERGE

```
# Available options for merges [see docs for details]:
.whenMatchedUpdate(...)    | .whenMatchedUpdateAll(...)  |
.whenNotMatchedInsert(...) | .whenMatchedDelete(...)
(deltaTable.alias("target").merge(
         source = updatesDF.alias("updates"),
         condition = "target.eventId = updates.eventId")
  .whenMatchedUpdateAll()
  .whenNotMatchedInsert(
     values = {
         "date": "updates.date",
         "eventId": "updates.eventId",
         "data": "updates.data",
         "count": 1
         }
  ).execute()
)
```

Insert with Deduplication using MERGE

```
(deltaTable.alias("logs").merge(
     newDedupedLogs.alias("newDedupedLogs"),
     "logs.uniqueId = newDedupedLogs.uniqueId")
  .whenNotMatchedInsertAll()
.execute()
)
```

TIME TRAVEL

View transaction log (aka Delta Log)

```
fullHistoryDF = deltaTable.history()
```

Query historical versions of Delta Lake tables

```
# choose only one option: versionAsOf, or timestampAsOf
df = (spark.read.format("delta")
                .option("versionAsOf", 0)
                .option("timestampAsOf", "2020-12-18")
                .load("/path/to/delta_table"))
```

TIME TRAVEL (CONTINUED)

Find changes between 2 versions of a table

```
df1 = spark.read.format("delta").load(pathToTable)
df2 = spark.read.format("delta").option("versionAsOf",
2).load("/path/to/delta_table")
df1.exceptAll(df2).show()
```

Rollback a table by version or timestamp

```
deltaTable.restoreToVersion(0)
deltaTable.restoreToTimestamp('2020-12-01')
```

UTILITY METHODS

Run Spark SQL queries in Python

```
spark.sql("SELECT * FROM tableName")
spark.sql("SELECT * FROM delta.`/path/to/delta_table`")
spark.sql("DESCRIBE HISTORY tableName")
```

Compact old files with Vacuum

```
deltaTable.vacuum() # vacuum files older than default
retention period (7 days)
deltaTable.vacuum(100) # vacuum files not required by
versions more than 100 hours old
```

Clone a Delta Lake table

```
deltaTable.clone(target="/path/to/delta_table/",
isShallow=True, replace=True)
```

Get DataFrame representation of a Delta Lake table

```
df = deltaTable.toDF()
```

Run SQL queries on Delta Lake tables

```
spark.sql("SELECT * FROM tableName")
spark.sql("SELECT * FROM delta.`/path/to/delta_table`")
```

PERFORMANCE OPTIMIZATIONS

Compact data files with Optimize and Z-Order

```
*Databricks Delta Lake feature
spark.sql("OPTIMIZE tableName [ZORDER BY (colA, colB)]")
```

Auto-optimize tables

```
*Databricks Delta Lake feature. For existing tables:
spark.sql("ALTER TABLE [table_name |
delta.`path/to/delta_table`]
SET TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true)
To enable auto-optimize for all new Delta Lake tables:
spark.sql("SET spark.databricks.delta.properties.
defaults.autoOptimize.optimizeWrite = true")
```

Cache frequently queried data in Delta Cache

```
*Databricks Delta Lake feature
spark.sql("CACHE SELECT * FROM tableName")
-- or:
spark.sql("CACHE SELECT colA, colB FROM tableName
           WHERE colNameA > 0")
```

databricks