DELTA LAKE: As per data bricks definition , delta lake is open format storage layer that delivers reliability, security, and performance on your data lake -for both streaming and batch operations. Delta lake is also single home for structured, semi-structured and unstructured data . Delta lake is extension of data lake .

DATA WAREHOUSE vs DATA LAKE vs DELTA LAKE

Data warehousing is one of the traditional concepts where we will do analytical processing with huge amount of data. In Data warehouse we are processing and doing analytical operations of data. Data Warehouse accept only structured data and easier to perform DML operation. Disadvantage is it supports only structured data. Data warehouse lacks support for unstructured and semi-structured data. In traditional architecture, computing resources such as RAM, processor, storage disk, network components are tightly coupled to each other which means we cannot increase one particular component linearly without increasing other components. Scaling up and scaling down also biggest challenge in data warehouse. We need to exponentially increase all resourcing components in DWH. That is the reason data lake came into picture. In data lake we can dump huge amount of data and it can accept any format structured, unstructured, or semi structured. Data lake is limitlessly scalable which means it can scale up/scale down any time with minimal support and also it can support huge amount of data . There is no limitation of storage, that is the big advantage of data lake. Disadvantage of data lake is data lake can't support DML operations, it can provide only minimal support to DML operations because data lake is nothing but storage layer where we will store the data in raw format so it's very difficult to scan and identify particular record and making some DML operations on top of that. So, it's one of the costliest operation. So, data lake is not suitable for DML operation. That is one of draw back in data lake. Another major problem with data lake is when we are processing/loading certain data or doing some transformations in between process got failed then process will leave system in corrupted state. That is also one of the major problem with data lake.

If we consider data warehouse , we are processing 100 records , if there is some problem, entire process /records will be rolled back. Entire process will be uncommitted. But coming to delta lake , partially we have done certain transformations and process failed, then process will leave the system as it is in the corrupted state. That is another problem with the data lake. By combining best features of DWH and Data Lake , Delta Lake came into picture. In Delta Lake, it can act like data lake which means it can accept structured, semi structured, and unstructured data . At the same time, easily we can scale up and scale down. Coming to DWH features, it can support ACID transactions easily. Also, if there is some process failure it will not leave the system corrupted. These are the advantages of delta lake. That's why delta lake came into existence.

Data Warehouse	Data Lake	Delta Lake
Only Structured Data	Structured/Semi-structured/Unstructured	Structured/Semi-structured/Unstructured/Streaming
Schema-on-Write	Schema-on-Read	Schema-on-Read
Supports ACID Transaction	Minimal support to ACID Transactions	Supports ACID Transaction
Does not corrupt the system	Leaves system in corrupted state	Does not corrupt the system

In DWH, when data is inserted into DWH,schema would be validated. If there is mismatch the data would be discarded. Let's say we are having one table employee with emp_id,emp_name and salary but incoming record is having extra field doj, then that record is not compatible with schema so on write would be discarded , it will be removed. Coming to Data Lake/Delta Lake, there is concept of schema evolution which means even though there is mismatch with schema still it would be accepted. So that is the advantage of data lake and delta lake.

DWH supports ACID Transaction. ACID means Atomicity, Consistency, Isolated and Durable. ACID is nothing but DML operations.DWH can support DML operations. Coming to data lake, its poor support to ACID transactions but delta lake it supports ACID transactions.

If there is any failure/error, DWH would not leave system in corrupted status, but coming to data lake it leaves system in corrupted state. Similar to DWH, in delta lake, it will not leave the system in corrupted status.

DELTA LAKE: Delta Lake is one of the storage format i.e., one of storage layer sitting on top of data lake. Basically, data lake would contain raw data in the form of big data file formats. It could be parquet,csv or anything. But delta lake is sitting on data lake. It is another storage layer that is managing metadata like information. For example, in DWH, if we are having table, it will contain metadata like what is name of column, what is data type and some other statistics about the table what is the minimum value, what is maximum value it contains metadata.

Data Lake will contain raw data but on top of that, delta lake will capture all metadata related to the table.so using that we can easily perform DML operations. Delta lake can accept any format of the data structured, semi-structured and unstructured. Delta Lake is foundation of cost effective ,highly scalable lake house.

DELTA LAKE PROPERTIES:

- **RELIABILITY :** Delta Lake is reliable because it does not leave system in corrupted state.
- **SECURITY:** We can control access to the users in delta lake similar to any DWH / data base tables.
- **PERFORMANCE:** Delta Lake is one of the layer sitting on top of data lake. It will contain all the metadata about data. So, with the help of metadata , it can perform better.

DELTA LAKE INTERNAL ARCHITECTURE- INTERNAL WORKING MECHANISM

Delta Lake is open-source storage layer sitting on top of data lake that is mainly used to support ACID transactions. ACID transactions are nothing but DML operations. Data Lake provides poor support for ACID operations.so it was shortcoming in data lake. As a result , delta lake came into picture and supports ACID transactions. Apart from ACID transactions, delta lake ensures data reliability and boost performance. Apart from this benefits, delta lake also simplifies lambda architecture to combine batch and streaming applications.

JSON Transaction Log File : Transaction logs will be created as part of delta table operations that is in the form of JSON file. Basically, JSON Transaction Log File will capture logs . so, when we are creating delta lake table and doing some modifications or performing DML operation, whatever operations we are performing on delta table, for each operation log file will be created in the sequence of 0,1,2,3 it will create in the sequential order. So, each and every json file will capture particular operation that got

executed on top of the delta lake. So, it keeps on creating many json log files. So, using this log files we can do time travel in delta lake. For example, we have created delta lake the version would be 0 and we have inserted /deleted (we have done many DML operations). Let's say it has created 10 log files , I can go back and forth using this json log file. This is called time travel. In Delta Lake we can perform time travel mainly because of json transaction log file.

Parquet Checkpoint File: Checkpoint means we are materializing some information to latest status. That is called check point. In parquet check point file, we are combining log information from json file .when we are doing certain operation in delta lake table, it will keep on creating json files.so let's say it has created 1000 files and later based on this log file, we have to traverse back and forth. So, there are 1000 log files it will be very difficult for the engine to traverse back through one by one. So, in order to speed up the process, delta lake came up with approach of check point file which means for every 10 JSON transaction log file,1 particular check point will be created in form of parquet.so this checkpoint will combine all the information that got generated in the previous 10 json files. It will combine all the log files. It will keep information of the current state of the table. When we have millions of json log files, it will be easier for the delta lake engine to traverse back using check point file instead of going back one by one through json file.

Basically, JSON file, each and every operations that got executed on top of delta lake that would be recorded in JSON log file. So, for every 10 JSON log files, one parquet check point will be created.

Cyclic Redundant Check File: This file is created not only in spark but also in Hadoop environment. Basically, this file is used to prevent any accidental damage to the data for the delta lake table.

CREATE DELTA TABLE:

If we are creating any delta table, metadata would be stored in database table in DATA menu, but actual data will be stored in dbfs location Data menu.

```
from delta.tables import *
DeltaTable.create(spark) \
    .tableName("delta_internal_demo") \
    .addColumn("emp_id","INT") \
    .addColumn("emp_name","STRING") \
    .addColumn("gender","STRING") \
    .addColumn("salary","INT") \
    .addColumn("Dept","STRING") \
    .property("description","table created for demo purpose") \
    .location("/FileStore/tables/delta/arch_demo") \
    .execute()
```


```
<delta.tables.DeltaTable at 0x7f54b45e68c0>
```

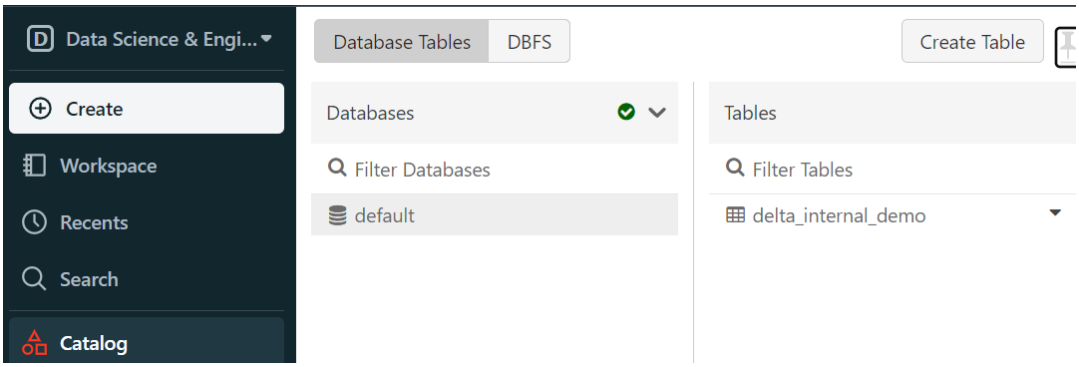
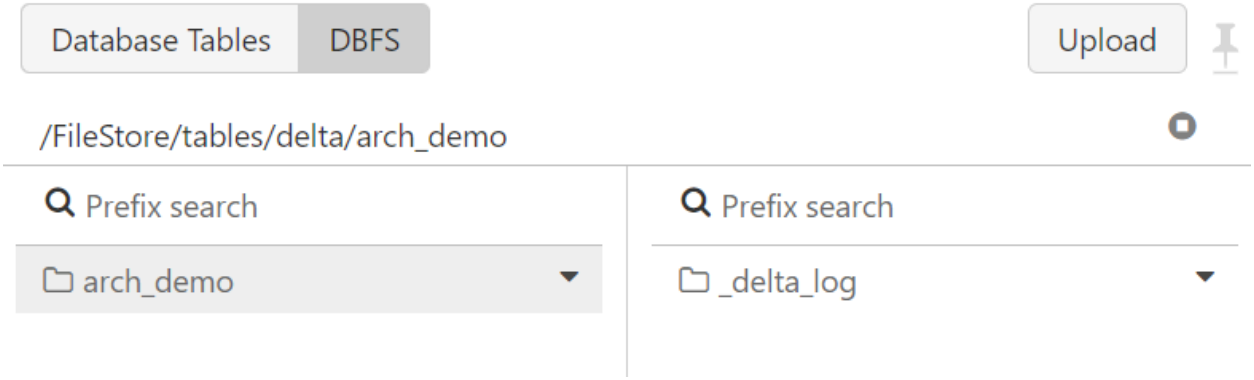
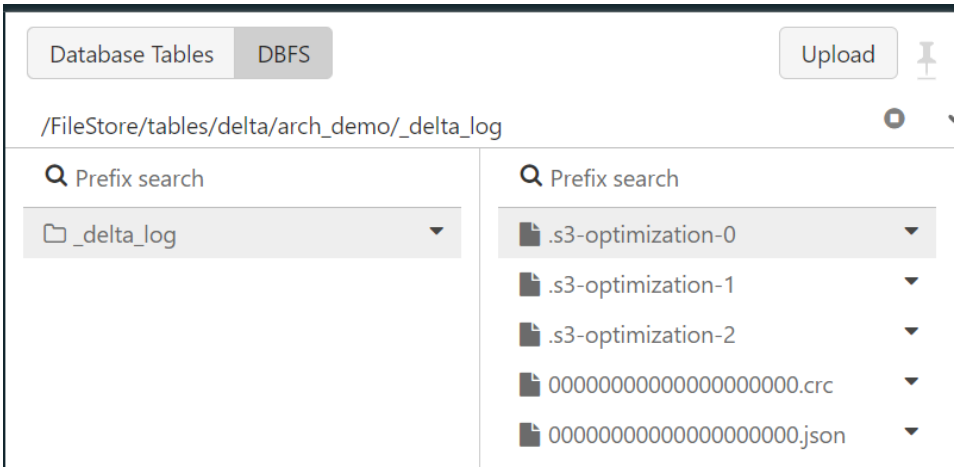


Table got created as shown above.



We haven't inserted any data that's why we can't see data file , in case we have inserted some data we will be able to see data file in this location arch_demo. And whatever operations we are performing on top of delta table including table creation everything would be recorded within folder **_delta_log.** This particular folder will be created while creating delta lake. It would be associated with all delta lake tables.

In _delta_log, we are able to see some files as shown below. Currently it has created 1 json file and 1 crc file. coming to json file, this will capture information related to particular operation. Initially in this first version, version number is 0, it starts with 20 zeroes. First initial version that is 0 containing the information related to table creation but still we haven't inserted any data.



#Read Data

Using list command , we are listing down all files which got created in _delta_log

%fs

ls /FileStore/tables/delta/arch_demo/_delta_log

Table +

	path	name	size	modificationTime
1	dbfs:/FileStore/tables/delta/arch_demo/_delta_log/.s3-optimization-0	.s3-optimization-0	0	1703418678000
2	dbfs:/FileStore/tables/delta/arch_demo/_delta_log/.s3-optimization-1	.s3-optimization-1	0	1703418678000
3	dbfs:/FileStore/tables/delta/arch_demo/_delta_log/.s3-optimization-2	.s3-optimization-2	0	1703418678000
4	dbfs:/FileStore/tables/delta/arch_demo/_delta_log/00000000000000000000.crc	00000000000000000000.crc	2253	1703418702000
5	dbfs:/FileStore/tables/delta/arch_demo/_delta_log/00000000000000000000.json	00000000000000000000.json	1319	1703418679000

5 rows | 14.51 seconds runtime Refre

If I want to see contents which are present in json file, we will use with below command.

%fs

head /FileStore/tables/delta/arch_demo/_delta_log/00000000000000000000.json

Output:

{"commitInfo":{"**timestamp**":1703418677798,"**userId**":"1086417094455142","**userName**":"paladugusravanthi005@gmail.com","**operation**":"CREATE TABLE","**operationParameters**":{"partitionBy":"[]","description":null,"isManaged":"false","properties":"{\"description\":\"table created for demo purpose\"}","statsOnLoad":false},"notebook":{"notebookId":"1328898932359836"},"clusterId":"1224-114433-rhshem1s","isolationLevel":"WriteSerializable","isBlindAppend":true,"operationMetrics":{},"tags":{"restoresDeletedRows":"false"},"engineInfo":"Databricks-Runtime/14.2.x-scala2.12","txnId":"8cf445f1-52d9-4236-8887-8d937df44804"}}

{"metaData":{"id":"1e926789-f924-475b-ba51-14581884a8bd","format":{"provider":"parquet","options":{}},"schemaString":"{\"type\":\"struct\",\"fields\":[{\"name\":\"**emp_id**\",\"type\":\"**integer**\",\"nullable\":true,\"metadata\":{}},{\"name\":\"**emp_name**\",\"type\":\"**string**\",\"nullable\":true,\"metadata\":{}},{\"name\":\"gender\",\"type\":\"string\",\"nullable\":true,\"metadata\":{}},{\"name\":\"salary\",\"type\":\"integer\",\"nullable\":true,\"metadata\":{}},{\"name\":\"**Dept**\",\"type\":\"**string**\",\"nullable\":true,\"metadata\":{}}]}","partitionColumns":[],"configuration":{"description":"table created for demo purpose"},"createdTime":1703418676056}}

{"protocol":{"minReaderVersion":1,"minWriterVersion":2}}

Here from above output ,operation is CREATE TABLE. And at what time this operation got executed ? who executed? We can also see information about table. For example, here we can see what is metadata of this table i.e.; what is the first column emp_id and its type integer . Similarly, what is emp_name type STRING, gender string and salary integer and finally Dept that is string so we can see all information related to table.

```
%sql
select * from delta_internal_demo
Output: Query returned no results.

#Insert Data
```

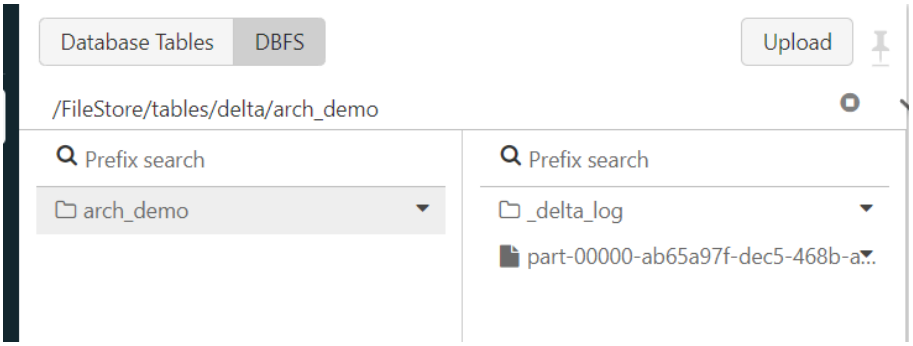
Any operations we are performing on delta lake table ,it will create json file. when we created table , json file created with version 0. When we perform insert , it will create one more version .it will create data files also and it will create json log file . In delta lake table ,that are created as part of delta lake table and it will be in form of parquet file so actual data would be created in _delta_log location in the form of parquet file and log associated to that operation would be created in _delta_log in json format.

```
%sql
insert into delta_internal_demo values(100,"Stephen","M",2000,"IT")
4 spark jobs
```

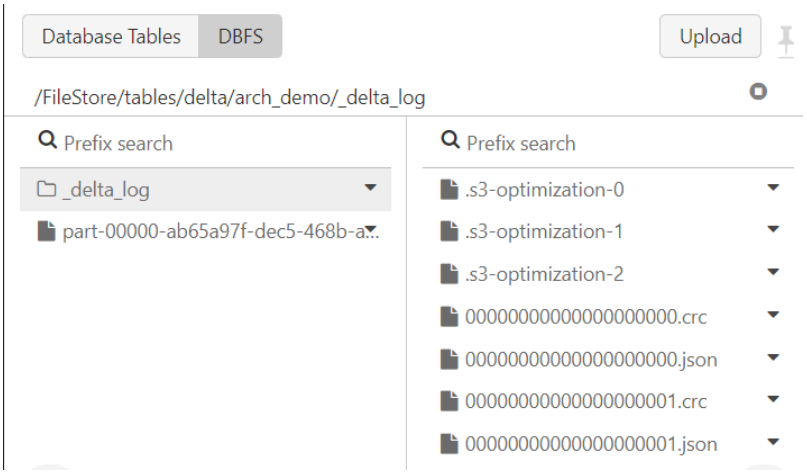
Table +

	num_affected_rows	num_inserted_rows
1	1	1

In below screenshot , we can see one file got created in the form of parquet file.



In below screenshot, we can also see that it has created next version of json file



%fs

head /FileStore/tables/delta/arch_demo/_delta_log/00000000000000000001.json

Output:

{"**add**":{"**path**":"**part-00000-ab65a97f-dec5-468b-aac8-570d0416718f-c000.snappy.parquet**","partitionValues":{},"**size**":1521,"**modificationTime**":1703424449000,"**dataChange**":true,"stats":"{\"**numRecords**\":1,\"**minValues**\":{\"emp_id\":100,\"emp_name\":\"Stephen\",\"gender\":\"M\",\"salary\":2000,\"Dept\":\"IT\"},\"**maxValues**\":{\"emp_id\":100,\"emp_name\":\"Stephen\",\"gender\":\"M\",\"salary\":2000,\"Dept\":\"IT\"},\"**nullCount**\":{\"emp_id\":0,\"emp_name\":0,\"gender\":0,\"salary\":0,\"Dept\":0}}","tags":{"**INSERTION_TIME**":"1703424449000000","**MIN_INSERTION_TIME**":"1703424449000000","**MAX_INSERTION_TIME**":"1703424449000000","**OPTIMIZE_TARGET_SIZE**":"268435456"}}}

{"commitInfo":{"timestamp":1703424449443,"**userId**":"1086417094455142","**userName**":"paladugusravanthi005@gmail.com","**operation**":"WRITE","operationParameters":{"mode":"Append","statsOnLoad":false,"partitionBy":"[]"},"notebook":{"notebookId":"1328898932359836"},"clusterId":"1224-114433-rhshem1s","readVersion":0,"isolationLevel":"WriteSerializable","isBlindAppend":true,"operationMetrics":{"numFiles":"1","numOutputRows":"1","numOutputBytes":"1521"},"tags":{"restoresDeletedRows":"false"},"engineInfo":"Databricks-Runtime/14.2.x-scala2.12","txnId":"58c6b9ef-20e8-48dd-8738-0d44be3da5c3"}}

Here operation is add , we have added some data and path is the partition file name in parquet format. This insert operation created 1 parquet output file. what is the size of the data , what is modification time, is it data change ? yes, and how many records got affected as part of this operation ? 1. When it captures the data minimum value/maximum value, it captures for all the columns so for emp_id that is minimal value. For emp_name minimal value, for gender minimal value, for salary minimal value. For Dept minimal value and also it has captured nullCount, it will check nullCount for

all the columns. Apart from that it will capture information such as insertion time,userId,username and what kind of operation? It's an insert so it's a write operation and what is the mode ? append mode and what is notebookId,clusterId and it will give all the information related to that particular operation that we executed on delta lake table. This is the log that it was produced as part of insert execution.

#Inserting few more records

When we are inserting some data, each insertion statement will be considered as a separate operation so there would be 2 different log files will be created for this operation. When we are executing 2 insert operations, it performing 2 DML operations, 2 separate DML operations so it will create two separate log files for each operation so in some cases we can insert based on select * from some table then in that case we may insert may be hundreds or millions of records also in one shot that would be treated as a one single transaction but here we are using insert into as a values and separated by semicolon which means when we are applying semi colon ,it will be considered as a separate transaction. When we execute 2 insert operations at a time it will show only 1 record got inserted in output but internally it has inserted 2 times but output, we will be able to see only 1 output because first it got executed, we could not see output and second it got executed this insert and it will give metrics related to only insert operation.

```
%sql
insert into delta_internal_demo values(200,"Philipp","M",8000,"HR");
insert into delta_internal_demo values(300,"Lara","F",6000,"SALES")
```

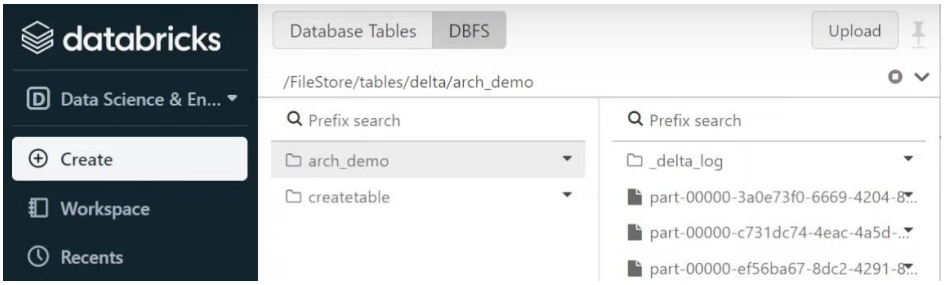
▸ (8) Spark Jobs

	num_affected_rows	num_inserted_rows
1	1	1

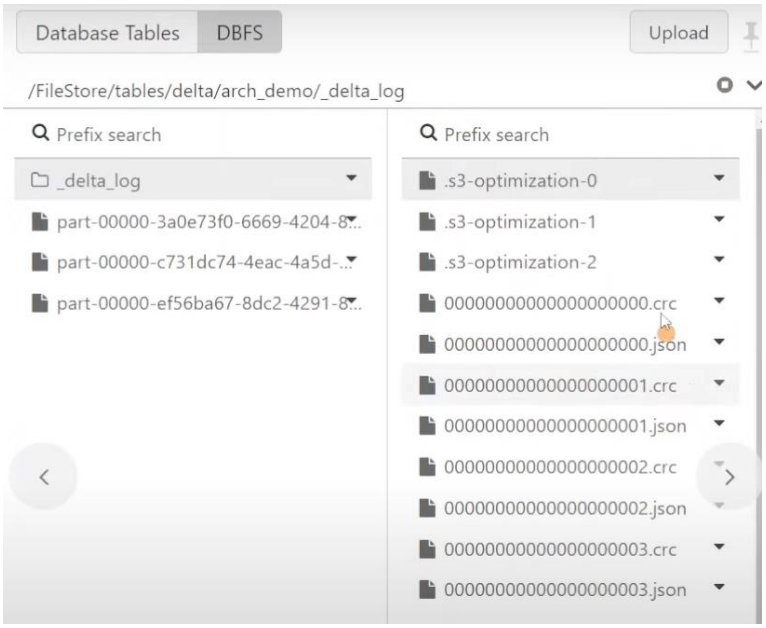
Even when we inserted 2 records, why it came as single record in output means record is separated by a semicolon which means both will be treated as a separate transactions so we will get this information only for the last DML operations that we are performing, here in this case for Lara is the only last DML operation we are performing and we are getting output only for that but internally both got executed.

Now initially table got created , on top of table we inserted one record, now we inserted 2 more records so there should be 4 log files got created zero version, 1, 2,3 .

In below screenshot it has created 2 more parquet data files because we have performed 2 write operation insert operation so for each operation, it has created parquet file.



Within _delta_log table , it has created 2 more json log files ,2nd version and 3rd version as shown below.



Now we are inserting few more duplicated records as shown below,

```
%sql
insert into delta_internal_demo values(100,"Stephen","M",2000,"IT");
insert into delta_internal_demo values(200,"Philipp","M",8000,"HR");
insert into delta_internal_demo values(300,"Lara","F",6000,"SALES");

insert into delta_internal_demo values(100,"Stephen","M",2000,"IT");
insert into delta_internal_demo values(200,"Philipp","M",8000,"HR");
insert into delta_internal_demo values(300,"Lara","F",6000,"SALES");
```

(24) Spark Jobs

	num_affected_rows	num_inserted_rows
1	1	1

Now there are 9 records in table totally.

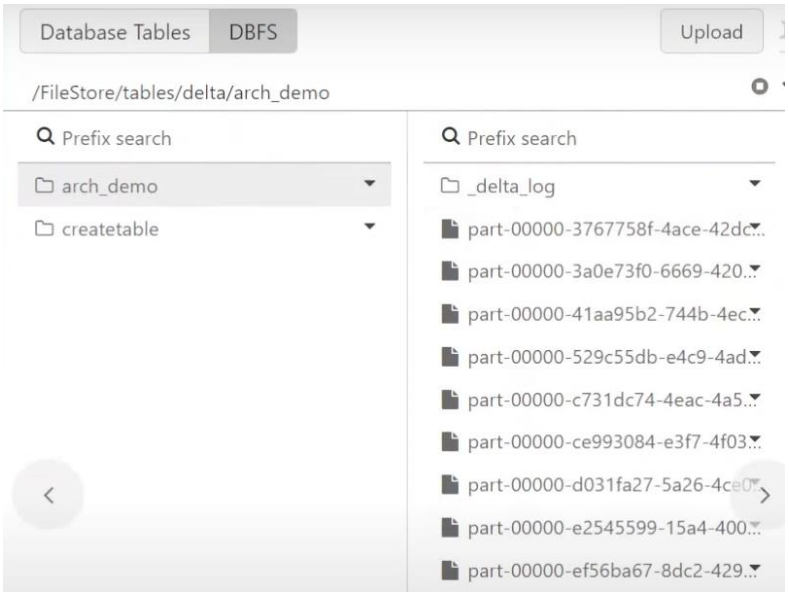
```
%sql

select * from delta_internal_demo
```

▸ (2) Spark Jobs

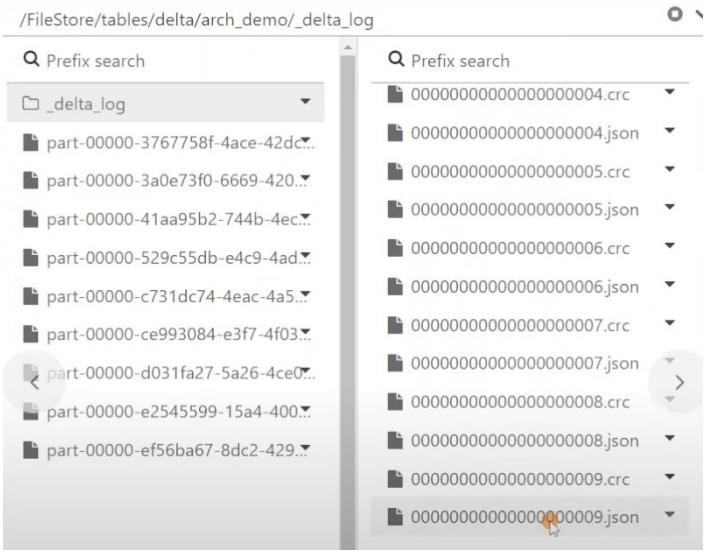
	emp_id	emp_name	gender	salary	Dept
1	100	Stephen	M	2000	IT
2	200	Philipp	M	8000	HR
3	300	Lara	F	6000	SALES
4	100	Stephen	M	2000	IT
5	200	Philipp	M	8000	HR
6	300	Lara	F	6000	SALES
7	100	Stephen	M	2000	IT

Showing all 9 rows.

Now it has created 9 parquet files as shown below. These are input files.



In the below screenshot, it has created few more json and crc files , it ended up with 9th version of json files. Version starts from 0(total 10 versions created , 1 version for create and remaining 9 versions for inserting records.



Still parquet check point is not created. For every 10th version parquet checkpoint file is created. Till now only till 9th version is created. Now going to execute one more operation on table as shown below.

```
%sql
delete from delta_internal_demo where emp_id=100;
```

(8) Spark Jobs

	num_affected_rows
1	3

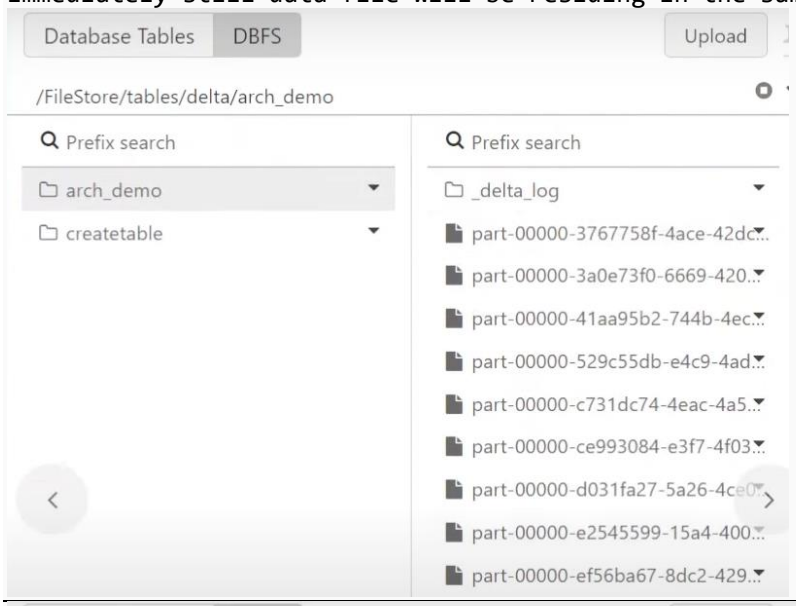
```
3 rows got deleted from table.
%sql

select * from delta_internal_demo
```

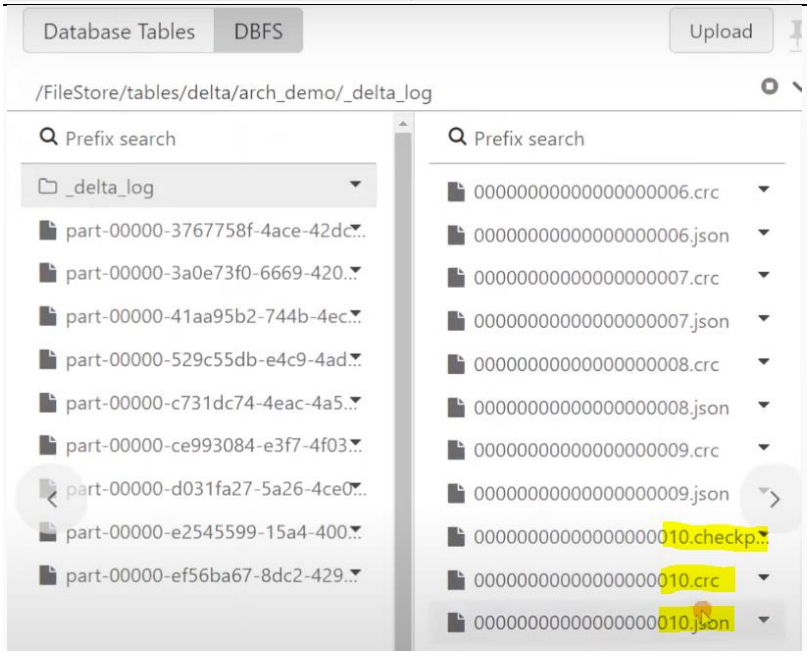
(3) Spark Jobs

	emp_id	emp_name	gender	salary	Dept
1	200	Philipp	M	8000	HR
2	300	Lara	F	6000	SALES
3	200	Philipp	M	8000	HR
4	300	Lara	F	6000	SALES
5	200	Philipp	M	8000	HR
6	300	Lara	F	6000	SALES

Showing all 6 rows.

Now if we see arch_demo folder as shown below, still 9 parquet files there even we removed 3 records from table. I can't see deleted 3 records in table but still actual data file is residing in below location why? Because when we are executing select query, it will go through log files , based on log file, in log file it will be mentioned we should not consider 3 files out of 9 files but it has not removed file immediately because as per delta lake architecture it follows soft delete pattern which means even though we are not interested in any particular data file, it will not be removed immediately, there is a default time duration of 7 days, it will be removed after 7 days. But that parameter is configurable even if we want to remove immediately, we can do it or if we want to remove after 10 days or 1 month still, we can configure that parameter but by default if we are deleting any data , it will not be removed immediately still data file will be residing in the same location for certain duration





If we see above screenshot, it has created 10th version of json file and parquet check point file also created for this operation.

#Reading Log Files for 10th version

%fs

head /FileStore/tables/delta/arch_demo/_delta_log/00000000000000000010.json

Output:

{"**remove**":{"**path**":"part-00000-89d11497-5d69-4c00-ac41-737794bdbf7a-c000.snappy.parquet","**deletionTimestamp**":1703485367263,"**dataChange**":true,"**extendedFileMetadata**":true,"**partitionValues**":{},"size":1521,"tags":{"INSERTION_TIME":"1703484023000000","**MIN_INSERTION_TIME**":"1703484023000000","**MAX_INSERTION_TIME**":"1703484023000000","**OPTIMIZE_TARGET_SIZE**":"268435456"}}}

{"remove":{"path":"part-00000-5de4e13c-a588-4474-bc27-849e1537eab1-c000.snappy.parquet","deletionTimestamp":1703485367263,"dataChange":true,"extendedFileMetadata":true,"partitionValues":{},"size":1521,"tags":{"INSERTION_TIME":"1703484008000000","MIN_INSERTION_TIME":"1703484008000000","MAX_INSERTION_TIME":"1703484008000000","OPTIMIZE_TARGET_SIZE":"268435456"}}}

{"remove":{"path":"part-00000-2e9b0d57-ff31-4075-a003-f110646f78dc-c000.snappy.parquet","deletionTimestamp":1703485367263,"dataChange":true,"extendedFileMetadata":true,"partitionValues":{},"size":1521,"tags":{"INSERTION_TIME":"1703482242000000","MIN_INSERTION_TIME":"1703482242000000","MAX_INSERTION_TIME":"1703482242000000","OPTIMIZE_TARGET_SIZE":"268435456"}}}

{"commitInfo":{"timestamp":1703485367284,"userId":"1086417094455142","userName":"paladugusravanthi005@gmail.com","**operation**":"DELETE","operationParameters":{"predicate":"[\"(emp_id#5846 = 100)\"]"},"notebook":{"notebookId":"1328898932359836"},"clusterId":"1225-051751-96ef2ppd","readVersion":10,"isolationLevel":"WriteSerializable","isBlindAppend":false,"operationMetrics":{"numRemovedFiles":"4","numRemovedBytes":"6084","numCopiedRows":"0","numDeletionVectorsAdded":"0","numDeletionVectorsRemoved":"0","numAddedChangeFiles":"0","executionTimeMs":"4098","numDeletedRows":"4","scanTimeMs":"2597","numAddedFiles":"0","numAddedBytes":"0","rewriteTimeMs":"1453"},"engineInfo":"Databricks-Runtime/12.2.x-scala2.12","txnId":"f099fc4c-1a14-4383-99d0-31217a389f84"}}

--

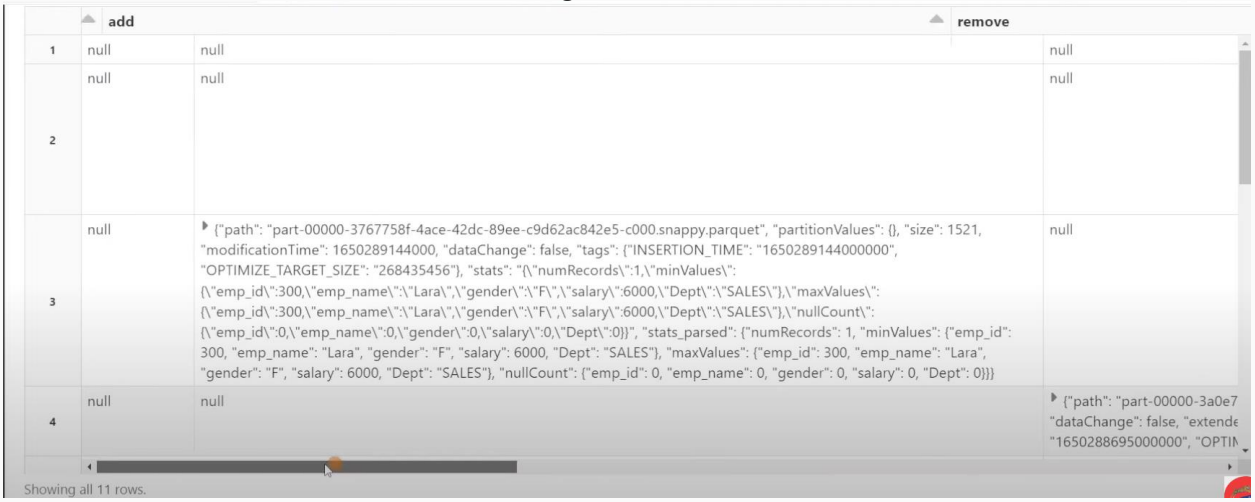
OUTPUT EXPLANATION: Here delete operation done so started output with remove, and delete operation done so operation=DELETE and it will have all the information related to data that we are removing. For example, 3 records we removed from delete operation, so 3 files will be removed from path as shown above.so based on the log files when we are querying on the delta lake table, the engine will come to this log files and it will understand these 3 files should not be considered for user output. That is the reason when we are querying on the select on delta table using select query, we were not able to see 9 records. Instead of that there is only 6 files, even though data files are still residing in the delta lake folder. It will also capture all information such as time stamp, user id, notebookid, clusterid etc.

Now I want to see parquet checkpoint result , using head we can't see because parquet format is not human readable format so if we want to see output of parquet checkpoint file, we will use format as shown below.

```
display(spark.read.format("parquet").load("/FileStore/tables/delta/arch_demo/_delta_log/00000000000000000010.checkpoint.parquet"))
```

(2) spark jobs

This file is having data of transaction, add, remove,metadata,protocol. This kind of information it is having. And in add column there is null because after inserting , we have removed 3 files so it shows null so only 6 log json files it has shown. And in remove column we can see data for 3 log files for which we removed the data.



So current snapshot current state is captured in the checkpoint file. so, check point file is mentioning active 6 files for the particular table and 3 files should not be considered which are removed from delete operation. This is the high-level information given to engine by parquet file. so, using checkpoint file, data engine can easily understand which file it has to consider which file it should not consider. If we want to do time travel , for example if we want to see previous version of data where we are considering this file then we can do time travel using log messages .

Log files are created in form of json files and checkpoint is created in form of parquet so for each and every operation json log files will be created and for every 10 json log files, one parquet file would be created.

When we are performing operations, it could be update,insert,delete, or merge. When we are performing these operations, how it works internally is let's say we are giving one update statement to delta lake table so what happens is based on that command, delta engine will scan all files related to for example need to update first_name for emp_id=100. When I am updating for emp_id=100, delta engine will scan the entire files. Out of that it will locate only the files where we are having suitable record for emp_id=100,it will scan and retrieve only those files that is the 1st step. Second step is only for the identified files(out of 9 files only 3 files are identified) for update then only those 3 files will be loaded into memory RAM. then in memory once again

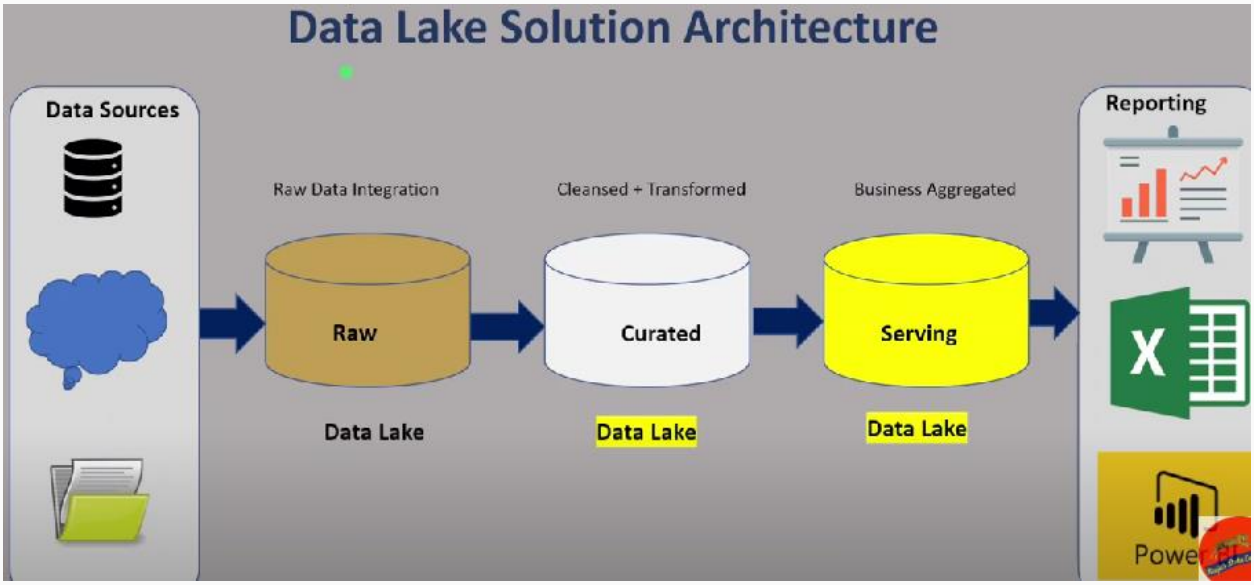
engine will start scanning entire data within the file so in our example 3 files got loaded into memory then processor core will start scanning the entire data within each file then it will make update. Once update is done, once again it will write the data back to storage system. This is how this works internally.

Summary: For insert/update/delete/merge operations, what happens is in first step it will scan entire file in the storage it could be dbfs/hdfs or s3 bucket or ADLS it could be anything. It will scan the storage in the delta lake table location then it will scan the storage, it will retrieve only files which are eligible for update then it will load those files into memory that is step2. And 3rd step is it will scan entire file in the memory. It will scan file by file the entire data and it will make respective update. Once update is done it will write data back to storage once again. This is how it works.

DELTA LAKE SOLUTION ARCHITECTURE

When there is requirement to built data ware solution using delta lake in data bricks , the first step is we need to create solution architecture for our project.

DATA LAKE SOLUTION ARCHITECTURE



Data Lake Solution Architecture contains 3 layers raw, curated and serving. Serving layer also calls as business layer/target layer. Generally, we will have 3 different layers. In Raw layer, one of the ETL tool might pull data from the data sources and it will dump the raw data into raw layer.it could be in one of the big data file format(csv,json,parquet,avro) so the data would be dumped into raw layer. This is mainly used for sourcing and storage purpose. This is built on top of data lake.

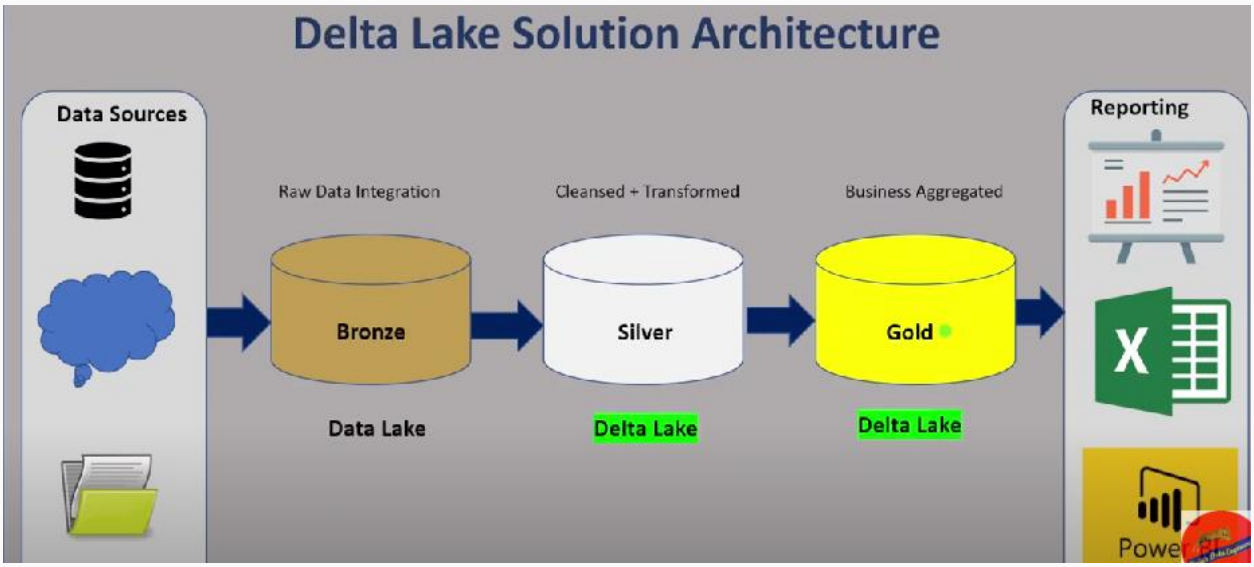
And in the next layer again ETL Tool, it could be ADF/ADB/Amazon glue/data flow in GCP and it could be some ETL tool that might pull the data from raw layer, it will do the cleansing and transformation operations and the curated data would be loaded into curated layer. Even this also built on top of data lake and once again etl tool will pull the data from curated layer and it will apply all the business kv logics.

Business KV aggregations and transformation and finally the final aggregated data would be moved to serving layer. This is also built on top of data lake .

Difference between data lake and delta lake is in data lake it will not support DML operations.it provides poor support for ACID transactions. Coming to Delta Lake, we can perform ACID transactions.

In data lake architecture, in any layer it's not queriable so we cannot perform any data warehousing like operations. In DWH operations generally we used to perform data analysis so directly we cannot perform data analysis on top of these data lake architectures. That was the short coming here.

DELTA LAKE SOLUTION ARCHITECTURE



In Delta Lake Architecture, we will call 3 layers as Bronze, Silver, and Gold. In data lake we used to have raw, curated and serving layer. In Bronze zone that is the integration point so etl tool may be data bricks pulling data from desperate data sources and dumping the raw data into bronze layer.it could be in any big data file format. This is built on top of data lake.

Coming to next layer Silver and Gold both are built in delta layer. Silver and Gold sitting on top of delta lake. Why bronze layer sitting on data lake and Silver and Gold sitting on delta lake? Because in bronze layer that is raw data, we are simply dumping all the raw data as it is . so here this is mainly for storage purpose. In most of the companies as per legal regulation we have to store data for many years so this is for long redemption period so that's the reason we don't perform any analysis on top of raw data . so that's the reason we are not going to perform any data warehouse operations on top of raw data so that's the reason we are using data lake. So etl process will consume data from the sources then it would dump into bronze layer and in the next step again it could be data bricks, data bricks will consume data from the bronze layer, it will clean and it will do basic transformations and it will dump the data into silver layer. In cleansing, we used to perform duplicate handling, null value handling or few basic transformations, those kind of operations we used to perform in

this layer cleansing layer. So cleansed data will be moved to silver layer. This is built on top of delta lake. Generally, after performing cleansing and transformation normally we used to query. We used to get many questions for data validation/data quality check so for that we used to write a query and we should check. So, this is sitting on top of delta lake so this is queryable. So, in this layer we can perform any Data warehouse operations. Once data is populated to silver layer then again etl process etl tool (data bricks) it can consume data from silver layer and then on top of that it can apply all the business kv operations like transformations and complex aggregations everything will be executed then final processed business aggregated data will be moved to gold layer and this is also data warehouse like layer so we can do any kind of data analysis and this is queryable. This gold layer in delta lake is similar to data warehousing serving layer. Any reporting tools any platform reporting tools can connect to this gold layer and it can consume data for the reporting purpose. So, this is high level data lake solution architecture.

Basically, this will contain 3 layers bronze, silver, and gold. Out of that bronze will continue in data lake architecture and silver and gold both layers will be sitting on top of delta lake.

CREATE DELTA TABLE

We can create delta table using 3 approaches.

First approach is creating delta table using pyspark programming. Through pyspark programming we are going to define delta table in a structured manner which means we are going to define delta table name and then after we are going to give all the columns along with data type and also, we can mention location of the delta table.

Second approach is we are going to follow SQL standard. Third approach is we are going to create delta table based on existing data frame. This is most commonly used approach in most of the projects.so for this approach we are going to use pyspark programming but still we are not going to define the table structure. Instead of that, we are going to write data frame in delta format which will create delta table with same structure as data frame.

METHOD-1 CREATE DELTA TABLE BY PYSPARK APPROACH

In order to create delta lake table using pyspark programming first we need to import libraries delta.tables. property in syntax is we can give any comment. For example, if you want to give some description about the table like what is purpose of this table. And coming to location, on which location this table should be created that we can specify using location parameter. Incase if we don’t specify location, then table will be created in hive metastore. Incase if we are giving any path then table would be sitting on top of that storage. That storage system should be dbfs/hdfs or any external storage system such as S3 bucket,adls or anything.

Code: 1st variant in method-1

```
from delta.tables import *
DeltaTable.create(spark) \
    .tableName("employee_demo") \
    .addColumn("emp_id","INT") \
    .addColumn("emp_name","STRING") \
    .addColumn("gender","STRING") \
    .addColumn("salary","INT") \
    .addColumn("Dept","STRING") \
    .property("description","table created for demo purpose") \
    .location("/FileStore/tables/delta/createtable") \
    .execute()
```

(3) spark jobs

```
Out[1]: <delta.tables.DeltaTable at 0x7f7e827da220>
```

Now let's say if table is already present in database employee_demo. If we again create employee_demo table, it will throw error.it won't create table again.so we will mention createIfNotExists, this will create table if table doesn't exists. Property and location are optional in the syntax.

2nd variant in method-1

```
from delta.tables import *
DeltaTable.createIfNotExists(spark) \
    .tableName("employee_demo") \
    .addColumn("emp_id","INT") \
    .addColumn("emp_name","STRING") \
    .addColumn("gender","STRING") \
    .addColumn("salary","INT") \
    .addColumn("Dept","STRING") \
    .execute()
```

No Spark Job is run.

--

We can also use create or replace let's say I am planning to create employee_demo table but incase if it is already present in the system instead of failing it will drop the previous version of the table and recreate with the structure. So, if that is our requirement then we can go with create or replace.

3rd variant in method-1

```
from delta.tables import *
DeltaTable.createOrReplace(spark) \
    .tableName("employee_demo") \
    .addColumn("emp_id","INT") \
    .addColumn("emp_name","STRING") \
    .addColumn("gender","STRING") \
    .addColumn("salary","INT") \
    .addColumn("Dept","STRING") \
```

```
    .execute()
```

(3) Spark Jobs

There are totally 3 variants create,createIfNotExists and createOrReplace

While giving table name in syntax, we can give table name or database name.table name. By default, table will be created in default data base. If we have any other data bases,then we can specify data base name explicitly.

METHOD-2 CREATE DELTA TABLE BY SQL APPROACH

1st variant in method-2 approach

```
%sql
CREATE TABLE employee_demoo (
  emp_id INT,
  emp_Name STRING,
  gender STRING,
  salary INT,
  dept STRING
) USING DELTA
```

Normal table and delta table is differentiated by USING DELTA keyword.

2nd variant in method-2 approach

```
%sql
CREATE TABLE IF NOT EXISTS employee_demoo (
  emp_id INT,
  emp_Name STRING,
  gender STRING,
  salary INT,
  dept STRING
) USING DELTA
```

0 spark jobs

In pyspark, there won't be space for IF NOT EXISTS but for SQL there will be space for IF NOT EXISTS. Also, in SQL it is not case sensitive. We can give lower case / upper case in SQL but coming to pyspark it is case sensitive. We cannot mix lower case/upper case in pyspark. And in pyspark first word is lower case and remaining words first character is upper case. But in sql we can add lower or upper.

3rd variant in method-2 approach

```
%sql

CREATE OR REPLACE TABLE employee_demooo (

  emp_id INT,
  emp_Name STRING,
```

```
  gender STRING,
  salary INT,
  dept STRING
) USING DELTA
LOCATION '/FileStore/tables/delta/createtable'
```

METHOD-3 CREATE DELTA TABLE BY DATA FRAME APPROACH

```
employee_data =[(100,"Stephen","M",2000,"IT"),
                (200,"Philipp","M",8000,"HR"),
                (300,"Lara","F",6000,"SALES")
            ]
employee_schema =["emp_id","emp_name","gender","salary","dept"]
df=spark.createDataFrame(employee_data,employee_schema)
display(df)
```

(3) spark jobs ran

Table +

	emp_id	emp_name	gender	salary	dept
1	100	Stephen	M	2000	IT
2	200	Philipp	M	8000	HR
3	300	Lara	F	6000	SALES

Now we want to create delta table on top of data frame.

```
#create table in the metastore using DataFrame's schema and write data to it
df.write.format("delta").saveAsTable("employee_demo1") or
df.write.format("delta").saveAsTable("default.employee_demo1")
```

(4) Spark Jobs ran

Table is created along with data. Table will be created based on same structure of data frame which means delta table will be created based on five columns with corresponding data type of df data frame

If we won't mention database name also by default it will be created in default data base. Otherwise, it will be created in specified database.

```
%sql
select * from employee_demo1
```

(2) Spark Jobs ran

Table +

	emp_id	emp_name	gender	salary	dept
1	100	Stephen	M	2000	IT
2	200	Philipp	M	8000	HR
3	300	Lara	F	6000	SALES

DELTA LAKE TABLE INSTANCE

In Databricks development when we are developing data lake we can create multiple instances for each delta table. Delta table instance is nothing but replica of actual delta table. This table instance will just create soft link to the actual table so basically it will not create another copy of the actual delta table. But instead of that it will just create a soft link which means even if we are going to perform DML operations on the table instance still it will be reflecting in the actual table or if we are performing any modification in the actual table that would be reflected in the table instance. So, delta table instance is nothing but soft link to the actual table. We already have table name to refer table but why do we need table instance? What is the usage of it? Basically, in order to perform any DML operation in delta table we are depending on SQL language. **Using SQL query, we can perform any DML operation using SQL statement on top of delta lake table.** But in some project or some developer, they are not interested to use SQL standard. May be as per project standard, they have to follow only pyspark. **Incase if we have to perform any DML operation on top of delta table then using table name we cannot do that.so for that first we have to create table instance then using table instance using pyspark language we can perform any DML operations so that is the usage of table instance.**

In order to create table instance , we can create it by 2 ways. One is using path or with Table Name.

Using Path Syntax is : DeltaTable.forPath(spark,"/path/to/table")
Using Table Name Syntax is : DeltaTable.forName(spark,"table_name")
Code:

```
from delta.tables import *
DeltaTable.create(spark) \
    .tableName("employee_demo") \
    .addColumn("emp_id","INT") \
    .addColumn("emp_name","STRING") \
    .addColumn("gender","STRING") \
    .addColumn("salary","INT") \
    .addColumn("Dept","STRING") \
    .property("description","table created for demo purpose") \
    .location("/FileStore/tables/delta/createtable") \
    .execute()
```

(3) spark jobs

Inserting few records in table.

```
%sql
insert into employee_demo values(100,"Stephen","M",2000,"IT");
insert into employee_demo values(200,"Philipp","M",8000,"HR");
insert into employee_demo values(300,"Lara","F",6000,"SALES");
```

(12) spark jobs ran.

And in output number effected rows and num inserted rows shows as 1 .

But in real time projects we don't insert data using this method. We used to create delta table directly based on the data frame. So, we don't need to consider these 2 steps in the real time project.

```
%sql
select * from employee_demo
```

(2) spark jobs ran.

	emp_id	emp_name	gender	salary	Dept
1	100	Stephen	M	2000	IT
2	200	Philipp	M	8000	HR
3	300	Lara	F	6000	SALES

Showing all 3 rows.

As per project standard we are not supposed to use any sql statement so I have to query table ,I have to create some command equivalent to this sql statement so for that we have to create table instance.

Creating table instance and naming it to deltaInstance1 and we will write syntax.

```
deltaInstance1 = DeltaTable.forPath(spark,"/FileStore/tables/delta/createtable")
```

Now deltaInstance1 is created. Now this instance is converting to DF and then display command to display data.

```
display(deltaInstance1.toDF())
```

(2) spark jobs ran

	emp_id	emp_name	gender	salary	Dept
1	100	Stephen	M	2000	IT
2	200	Philipp	M	8000	HR
3	300	Lara	F	6000	SALES

Showing all 3 rows.

Incase if we want to avoid sql statement approach, we can develop solution using pyspark approach and then we can create delta instance and based on that we can query the table as shown above like converting to df and displaying the result.

```
%sql
delete from employee_demo where emp_id=100
```

(6) spark jobs ran

We have delete data from sql statement. Instead of deleting data from sql table, we can delete it via table instance.

`display(deltaInstance1.toDF())` → We haven't deleted records from instance but If we query it will return 2 records , because this instance and actual table both are pointing to same memory, so that's why if we query from pyspark it will query 2 records only. Whatever operations we are performing on instance or table both will reflect to each other.

Output: (2) spark jobs

	emp_id	emp_name	gender	salary	Dept
1	200	Philipp	M	8000	HR
2	300	Lara	F	6000	SALES

--

deltaInstance1.delete("emp_id=200") → in brackets we need to give condition

(6) spark jobs ran

display(deltaInstance1.toDF())

(1) Spark job ran

	emp_id	emp_name	gender	salary	Dept
1	300	Lara	F	6000	SALES

Showing all 1 rows.

SECOND APPROACH OF SYNTAX:

Instead of path we can give forName also.

deltaInstance2= DeltaTable.forName(spark,"employee_demo")

display(deltaInstance2.toDF())

```
1  display(deltaInstance2.toDF())
```

▸ (1) Spark Jobs

	emp_id	emp_name	gender	salary	Dept
1	300	Lara	F	6000	SALES

Showing all 1 rows.

We can perform DML operations in pyspark programming using delta instances. Not only DML operations, whatever operations we are performing using SQL commands everything we can be performed using delta instance. For example, in order to get history for delta tables, normally in sql we used to write command as below

%sql

DESCRIBE HISTORY employee_demo

(1) Spark job ran

Its giving table history what are the operations we have performed on this table

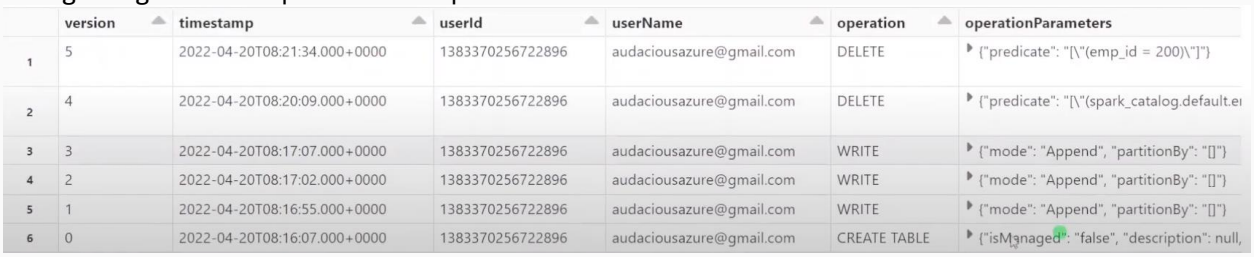
	version	timestamp	userId	userName	operation	operationParameters
1	5	2022-04-20T08:21:34.000+0000	1383370256722896	audaciousazure@gmail.com	DELETE	{"predicate": "[\"(emp_id = 200)\"]"}
2	4	2022-04-20T08:20:09.000+0000	1383370256722896	audaciousazure@gmail.com	DELETE	{"predicate": "[\"(spark_catalog.default.e
3	3	2022-04-20T08:17:07.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitionBy": "[]"}
4	2	2022-04-20T08:17:02.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitionBy": "[]"}
5	1	2022-04-20T08:16:55.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitionBy": "[]"}
6	0	2022-04-20T08:16:07.000+0000	1383370256722896	audaciousazure@gmail.com	CREATE TABLE	{"isManaged": "false", "description": null,

Now using pyspark approach also we can see this history.

```
display(deltaInstance2.history())
```

(1) Spark job ran.

Its giving same output as in sql.



	version	timestamp	userId	userName	operation	operationParameters
1	5	2022-04-20T08:21:34.000+0000	1383370256722896	audaciousazure@gmail.com	DELETE	{"predicate": "[\"(emp_id = 200)\"]"}
2	4	2022-04-20T08:20:09.000+0000	1383370256722896	audaciousazure@gmail.com	DELETE	{"predicate": "[\"(spark_catalog.default.e
3	3	2022-04-20T08:17:07.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitionBy": "[]"}
4	2	2022-04-20T08:17:02.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitionBy": "[]"}
5	1	2022-04-20T08:16:55.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitionBy": "[]"}
6	0	2022-04-20T08:16:07.000+0000	1383370256722896	audaciousazure@gmail.com	CREATE TABLE	{"isManaged": "false", "description": null,

Delta Table Instance is nothing but replica of master table(actual table) but still it is pointing to same memory. It's not creating any separate copy so whatever the operations we are performing on actual table or instance it would be reflecting to each other. What is the use of this? If we want to follow only pyspark or Scala programming within the notebook then we can go with delta table instance creation>

DIFFERENT APPROACHES TO INSERT DATA INTO DELTA TABLE

DELTA TABLE CREATION

```
from delta.tables import *
DeltaTable.create(spark) \
    .tableName("employee_demo") \
    .addColumn("emp_id","INT") \
    .addColumn("emp_name","STRING") \
    .addColumn("gender","STRING") \
    .addColumn("salary","INT") \
    .addColumn("Dept","STRING") \
    .property("description","table created for demo purpose") \
    .location("/FileStore/tables/delta/path_employee_demo") \
    .execute()
```

(3)spark jobs ran

```
Out[1] : <delta.tables.DeltaTable at 0x7f466cd18910>
```

SQL STYLE INSERT

```
%sql
insert into employee_demo values(100,"Stephen","M",2000,"IT");
```

(4) spark jobs ran

And in output number affected rows and number inserted rows is 1

Table +

	num_affected_rows	num_inserted_rows
1	1	1

1 row | 13.19 seconds runtime

Another approach to display SQL style insert

```
display(spark.sql("select * from employee_demo"))
```

(1) Spark jobs ran

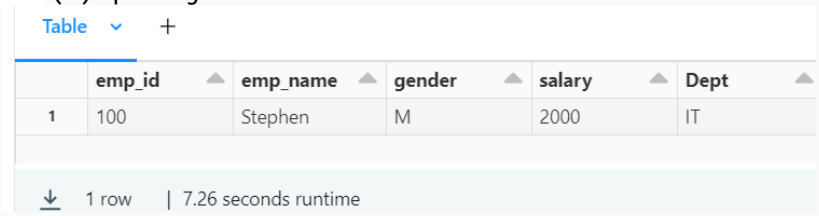


Table +

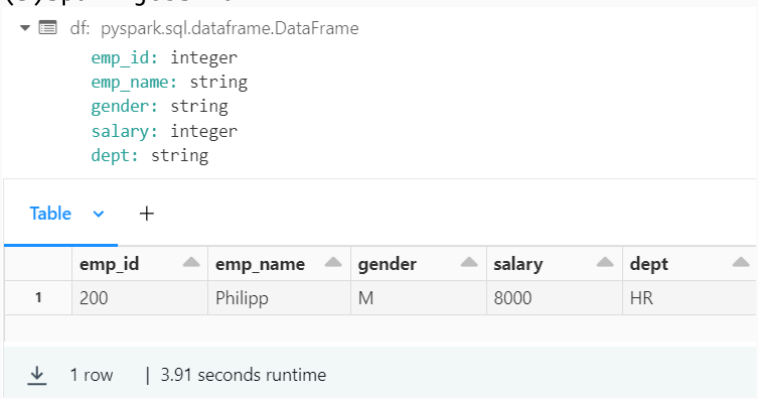
	emp_id	emp_name	gender	salary	Dept
1	100	Stephen	M	2000	IT

1 row | 7.26 seconds runtime

DATAFRAME TYPE INSERT

```
from pyspark.sql.types import IntegerType,StringType
employee_data =[(200,"Philipp","M",8000,"HR")]
employee_schema= StructType([
    StructField("emp_id",IntegerType(),False), \
    StructField("emp_name",StringType(),True), \
    StructField("gender",StringType(),True), \
    StructField("salary",IntegerType(),True), \
    StructField("dept",StringType(),True) \
    ])
df= spark.createDataFrame(employee_data,employee_schema)
display(df)
```

(3)spark jobs ran



df: pyspark.sql.dataframe.DataFrame

emp_id: integer
emp_name: string
gender: string
salary: integer
dept: string

Table +

	emp_id	emp_name	gender	salary	dept
1	200	Philipp	M	8000	HR

1 row | 3.91 seconds runtime

Now data frame got created.

Now if we want to insert data frame into delta table, then for that we can use data frame reader(write)

```
df.write.format("delta").mode("append").saveAsTable("employee_demo")
```

(4)spark jobs ran.

--

Now let's query and see whether new record got added or not.

```
display(spark.sql("select * from employee_demo"))
```

(2)spark jobs ran.

Table +

	emp_id	emp_name	gender	salary	Dept
1	100	Stephen	M	2000	IT
2	200	Philipp	M	8000	HR

2 rows | 3.04 seconds runtime

```
--
DATAFRAME INSERT INTO METHOD USING DIFFERENT SYNTAX
from pyspark.sql.types import IntegerType,StringType
employee_data =[(300,"Lara","F",6000,"SALES")]
employee_schema= StructType([
    StructField("emp_id",IntegerType(),False), \
    StructField("emp_name",StringType(),True), \
    StructField("gender",StringType(),True), \
    StructField("salary",IntegerType(),True), \
    StructField("dept",StringType(),True) \
    ])
df1= spark.createDataFrame(employee_data,employee_schema)
display(df1)
```

(3)spark jobs ran

df1: pyspark.sql.dataframe.DataFrame
- emp_id: integer
- emp_name: string
- gender: string
- salary: integer
- dept: string

Table +

	emp_id	emp_name	gender	salary	dept
1	300	Lara	F	6000	SALES

1 row | 0.88 seconds runtime

```
--
df1.write.insertInto('employee_demo',overwrite=False)
```

If overwrite true means it will truncate existing data on top of that it will insert only data from the data frame. When we are giving overwrite = false which means it will retain old records. Already in this table we are having 2 records both would be retained. On top of that emp_id=300 would be inserted.

```
%sql
select* from employee_demo
```

(2) Spark jobs

Table +

	emp_id	emp_name	gender	salary	Dept
1	100	Stephen	M	2000	IT
2	200	Philipp	M	8000	HR
3	300	Lara	F	6000	SALES

3 rows | 2.30 seconds runtime

INSERT USING TEMP VIEW

We will convert data frame as a temp view then after we can use sql command to insert data based on temp view.

#converting data frame to temp view

df1.createOrReplaceTempView("delta_data") → Now this data frame got converted to temp view. delta_data view got created

```
%sql
select * from delta_data
```

(3) Spark jobs ran

Table +

	emp_id	emp_name	gender	salary	dept
1	300	Lara	F	6000	SALES

1 row | 0.84 seconds runtime

Now we will insert content of temp view to main table.

```
%sql
insert into employee_demo
select * from delta_data
```

(4) Spark jobs ran

	num_affected_rows	num_inserted_rows
1	1	1

Showing all 1 rows.

```
%sql
select * from employee_demo
```

(2) Spark jobs ran

Table +

	emp_id	emp_name	gender	salary	Dept
1	100	Stephen	M	2000	IT
2	200	Philipp	M	8000	HR
3	300	Lara	F	6000	SALES
4	300	Lara	F	6000	SALES

4 rows | 2.47 seconds runtime

SPARK SQL INSERT METHOD

```
spark.sql("insert into employee_demo select * from delta_data")
```

▸ (4) Spark Jobs

```
Out[9]: DataFrame[num_affected_rows: bigint, num_inserted_rows: bigint]
```

--

```
%sql
select * from employee_demo
```

▸ (2) Spark Jobs

	emp_id	emp_name	gender	salary	Dept
1	100	Stephen	M	2000	IT
2	200	Philipp	M	8000	HR
3	300	Lara	F	6000	SALES
4	300	Lara	F	6000	SALES
5	300	Lara	F	6000	SALES

--

DIFFERENT APPROACHES TO DELETE DATA FROM DELTA TABLE

METHOD-1

```
%sql
delete from employee_demo where emp_id=100
```

(6)spark jobs ran

METHOD-2 SQL USING DELTA LOCATION

```
%sql
DELETE FROM delta.`/FileStore/tables/delta/path_employee_demoo` WHERE emp_id = 100;
```

(6)spark jobs ran

Output : Number affected rows=1

METHOD-3 SPARK SQL

```
spark.sql("delete from employee_demo where emp_id=100")
```

(8) spark jobs ran

```
Out[2]: DataFrame[num_affected_rows:bigint]
```

METHOD-4 PYSPARK DELTA TABLE INSTANCE

```
from delta.tables import *
from pyspark.sql.functions import *
deltaTable =DeltaTable.forName(spark,'employee_demo') → we can create one table
instance for our delta table.
#Declare the predicate by using a SQL-formatted string.
```

```
deltaTable.delete("emp_id=400") →we are deleting using table instance.
Output:
(7) spark jobs ran.
```

```
METHOD-5 MULTIPLE CONDITIONS USING SQL PREDICATE
deltaTable.delete("emp_id=500 and gender ='F' ")
(7) Spark Jobs ran
```

```
METHOD-6 PYSPARK DELTA TABLE INSTANCE – SPARK SQL PREDICATE
#Declare the predicate by using Spark SQL functions.
deltaTable.delete(col('emp_id')== 600)
(7) spark jobs ran
```

DELTA TABLE UPDATE

```
#CREATE TABLE

%sql
CREATE OR REPLACE TABLE employee_demo (
  emp_id INT,
  emp_name STRING,
  gender STRING,
  salary INT,
  Dept STRING)
USING DELTA
LOCATION '/FileStore/tables/delta/path_employee_demoo'
   (3)Spark jobs
OK
#POPULATE SAMPLE DATA
%sql
insert into employee_demo values(100,"Stephen","M",2000,"IT");
insert into employee_demo values(200,"Philipp","M",8000,"HR");
insert into employee_demo values(300,"Lara","F",6000,"SALES");
insert into employee_demo values(400,"Mike","M",4000,"IT");
insert into employee_demo values(500,"Sarah","F",9000,"HR");
insert into employee_demo values(600,"Serena","F",5000,"SALES");
insert into employee_demo values(700,"Mark","M",7000,"SALES");
(28) spark jobs ran.
```

	num_affected_rows	num_inserted_rows
1	1	1

Showing all 1 rows.

```
---
%sql
select * from employee_demo
```

	emp_id	emp_name	gender	salary	Dept
1	600	Serena	F	5000	SALES
2	100	Stephen	M	2000	IT
3	200	Philipp	M	8000	HR
4	300	Lara	F	6000	SALES
5	700	Mark	M	7000	SALES
6	500	Sarah	F	9000	HR
7	400	Mike	M	4000	IT

Showing all 7 rows.

METHOD 1: SQL STANDARD USING TABLE NAME

```
%sql
update employee_demo set salary=10000 where emp_name='Mark';
```

(6)spark jobs ran

Table +

	num_affected_rows
1	1

1 row | 5.50 seconds runtime

METHOD 2: SQL STANDARD USING TABLE PATH

```
%sql
UPDATE delta.`/FileStore/tables/delta/path_employee_demoo` SET salary=12000 WHERE
emp_name ='Mark';
```

(6)spark jobs ran

_sqldf: pyspark.sql.dataframe.DataFrame
num_affected_rows: long

Table +

	num_affected_rows
1	1

1 row | 5.97 seconds runtime

```
%sql
select * from employee_demo
```

(3) Spark Jobs

	emp_id	emp_name	gender	salary	Dept
1	600	Serena	F	5000	SALES
2	100	Stephen	M	2000	IT
3	200	Philipp	M	8000	HR
4	300	Lara	F	6000	SALES
5	700	Mark	M	12000	SALES
6	500	Sarah	F	9000	HR
7	400	Mike	M	4000	IT

Showing all 7 rows.

```
METHOD 3: PYSPARK STANDARD USING TABLE INSTANCE
from delta.tables import *
from pyspark.sql.functions import *
deltaTable = DeltaTable.forPath(spark,'/FileStore/tables/delta/path_employee_demoo')
#Declare the predicate by using a SQL-formatted string
deltaTable.update(
    condition="emp_name = 'Mark' ",
    set={ "salary":"15000" }
)
```

(8)spark jobs ran

```
%sql
select * from employee_demo
```

Table +

	emp_id	emp_name	gender	salary	Dept
1	600	Serena	F	5000	SALES
2	300	Lara	F	6000	SALES
3	100	Stephen	M	2000	IT
4	200	Philipp	M	8000	HR
5	700	Mark	M	15000	SALES
6	500	Sarah	F	9000	HR
7	400	Mike	M	4000	IT

7 rows | 2.80 seconds runtime

```
METHOD 4-DECLARE PREDICATE BY USING SPARK SQL FUNCTIONS
deltaTable.update(
    condition=col('emp_name') =='Mark',
    set={'Dept':lit('IT')}
)
```

7 spark jobs ran.

```
%sql
select * from employee_demo
```

Table +

	emp_id	emp_name	gender	salary	Dept
1	600	Serena	F	5000	SALES
2	300	Lara	F	6000	SALES
3	100	Stephen	M	2000	IT
4	200	Philipp	M	8000	HR
5	500	Sarah	F	9000	HR
6	400	Mike	M	4000	IT
7	700	Mark	M	15000	IT

7 rows | 2.49 seconds runtime

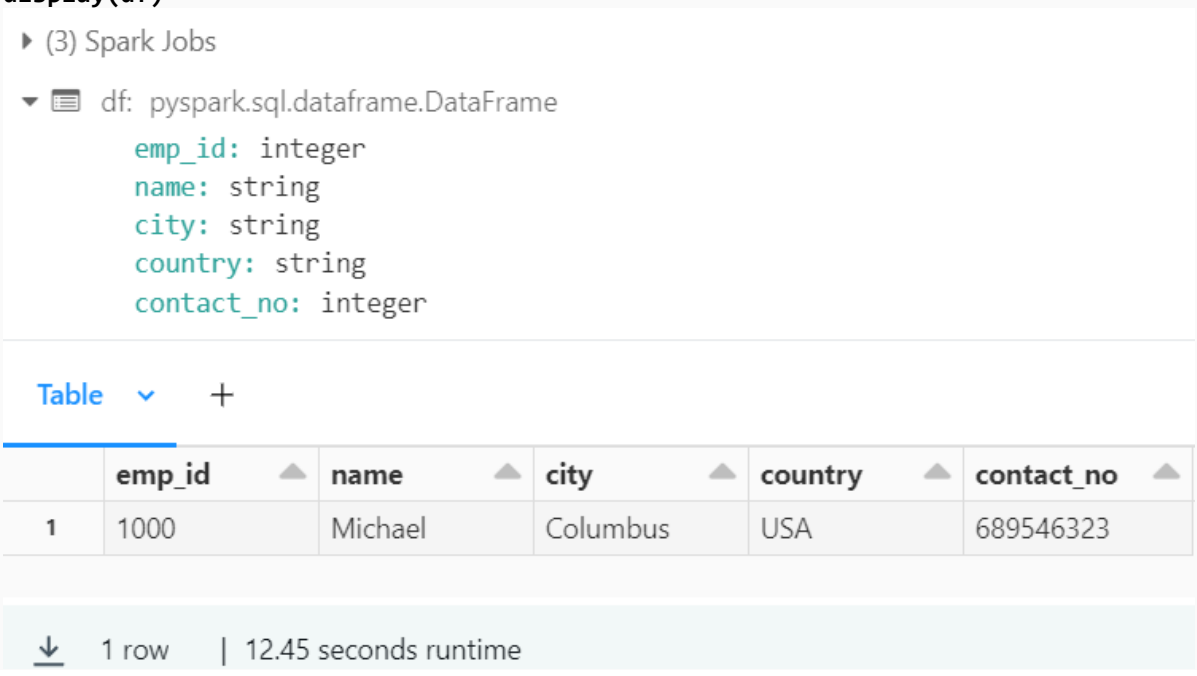
SCD TYPE 1 :MERGE/UPSERT OPERATION USING PYSPARK AND SPARK SQL

When we have to update target table based on latest incoming source data. We have to compare the data. If there is match found between source and target then target data would be updated based on the source. If there is no match found in the target table which means in source we received new data so new record would be inserted into target table. This operation is called upsert or merge operation.

When data lake got introduced in the bigdata development we were not able to perform merge operation. It was one of the shortcoming of data lake. Later when data bricks introduced delta lake, we are able to perform merge operation.

Merge statement can be performed via spark SQL but still it's not suitable for all the scenarios.(have to create UDF and using that UDF have to perform merge operation.in pyspark have to create UDF. So, some scenarios not suitable for spark SQL

```
from pyspark.sql.types import *
from pyspark.sql.functions import *
schema = StructType([
    StructField("emp_id",IntegerType(),True),
    StructField("name",StringType(),True),
    StructField("city",StringType(),True),
    StructField("country",StringType(),True),
    StructField("contact_no",IntegerType(),True)
])
--
data=[(1000,"Michael","Columbus","USA",689546323)]
df = spark.createDataFrame(data,schema)
display(df)
```



```
%sql
CREATE OR REPLACE TABLE dim_employee(
  emp_id INT,
  name STRING,
  city STRING,
  country STRING,
  contact_no INT
)
USING DELTA
LOCATION "/FileStore/tables/delta_merge"
```

(3)spark jobs ran.
Ok
--

```
%sql
select * from dim_employee
```

Output: Query returned no results.

METHOD-1 SPARK SQL

I need to convert df in to table , so here source is df and target is table. So for that we are creating temp view using below syntax.

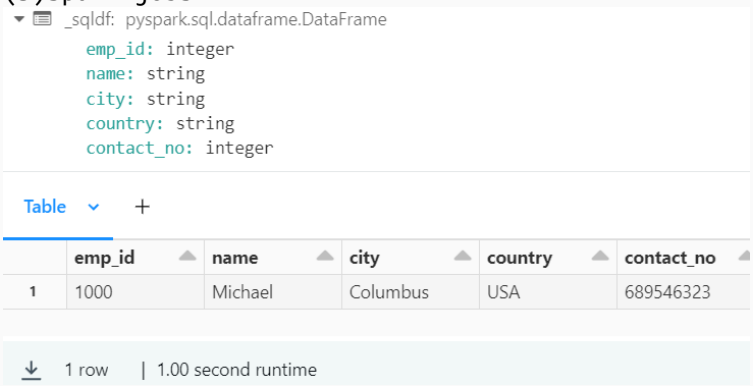
```
df.createOrReplaceTempView("source_view")
```

--

```
%sql
select * from source_view
```

→ This is our source data.

(3)spark jobs




```
%sql
select * from dim_employee
```

→ This is our target data. Currently no data in target table. So, during merge operation condition won't match so it will insert data into table.

```
%sql
MERGE INTO dim_employee as target
```

```
USING source_view as source
ON target.emp_id=source.emp_id
WHEN MATCHED
THEN UPDATE SET
target.name=source.name,
target.city=source.city,
target.country=source.country,
target.contact_no=source.contact_no
WHEN NOT MATCHED THEN
INSERT (emp_id,name,city,country,contact_no) VALUES
(emp_id,name,city,country,contact_no)
```

(5)spark jobs ran.

	num_affected_rows	num_updated_rows	num_deleted_rows	num_inserted_rows
1	1	0	0	1

```
%sql
select * from dim_employee
```

(1)spark jobs ran.

	emp_id	name	city	country	contact_no
1	1000	Michael	Columbus	USA	689546323

Creating another source of data.

```
data
=[(1000,"Michael","Chicago","USA",689546323),(2000,"Nancy","NewYork","USA",76345902)]
df= spark.createDataFrame(data,schema)
display(df)
```

(3)spark jobs ran.

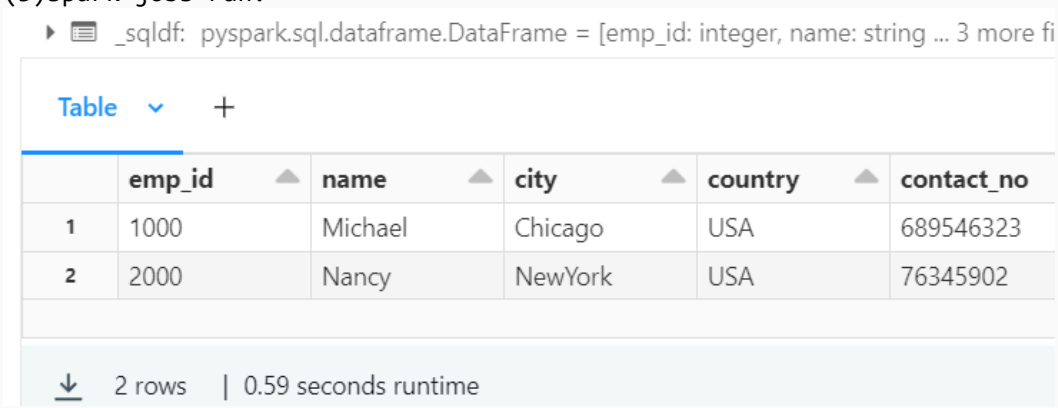
df: pyspark.sql.dataframe.DataFrame

```
emp_id: integer
name: string
city: string
country: string
contact_no: integer
```

Table +

	emp_id	name	city	country	contact_no
1	1000	Michael	Chicago	USA	689546323
2	2000	Nancy	NewYork	USA	76345902

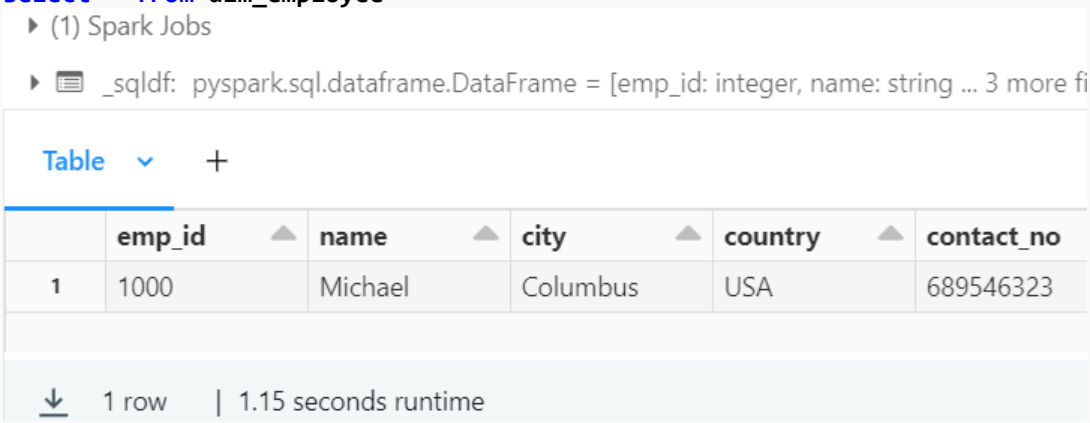
```
df.createOrReplaceTempView("source_view")
%sql
select * from source_view
(3)spark jobs ran.
```



	emp_id	name	city	country	contact_no
1	1000	Michael	Chicago	USA	689546323
2	2000	Nancy	NewYork	USA	76345902

2 rows | 0.59 seconds runtime

```
--
%sql
select * from dim_employee
```



	emp_id	name	city	country	contact_no
1	1000	Michael	Columbus	USA	689546323

1 row | 1.15 seconds runtime

```
--
%sql
MERGE INTO dim_employee as target
USING source_view as source
ON target.emp_id=source.emp_id
WHEN MATCHED
THEN UPDATE SET
target.name=source.name,
target.city=source.city,
target.country=source.country,
target.contact_no=source.contact_no
WHEN NOT MATCHED THEN
INSERT (emp_id,name,city,country,contact_no) VALUES
(emp_id,name,city,country,contact_no)
```

(11) spark jobs ran

	num_affected_rows	num_updated_rows	num_deleted_rows	num_inserted_rows
1	2	1	0	1

1 row | 11.67 seconds runtime

```
%sql
select * from dim_employee
```

(1) Spark jobs ran

_sqldf: pyspark.sql.dataframe.DataFrame = [emp_id: integer, name: string ... 3 more f

Table +

	emp_id	name	city	country	contact_no
1	1000	Michael	Chicago	USA	689546323
2	2000	Nancy	NewYork	USA	76345902

2 rows | 2.09 seconds runtime

METHOD 2 – PYSPARK

Here we don't need to create temporary view. Because we need to keep data in data frame format. We need to create delta table(delta_df). delta_df is a target.

```
data =[(2000,"Sarah","New
York","USA",76345902),(3000,"David","Atlanta","USA",563456787)]
df= spark.createDataFrame(data,schema)
display(df)
```

(3)spark jobs ran.

df: pyspark.sql.dataframe.DataFrame = [emp_id: integer, name: string ... 3 more fields]

Table +

	emp_id	name	city	country	contact_no
1	2000	Sarah	New York	USA	76345902
2	3000	David	Atlanta	USA	563456787

2 rows | 0.87 seconds runtime

Now we have created source data frame. Now we will create delta table.

```
from delta.tables import *
delta_df= DeltaTable.forPath(spark,"/FileStore/tables/delta_merge")
--
delta_df.alias("target").merge(
```

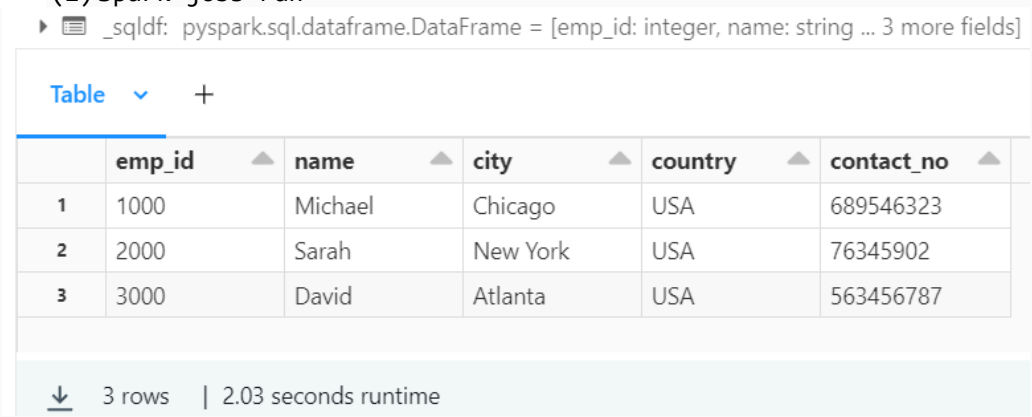
```
    source=df.alias("source"),
    condition="target.emp_id=source.emp_id"
).whenMatchedUpdate(set =
   {
    "name" : "source.name",
    "city": "source.city",
    "country":"source.country",
    "contact_no" : "source.contact_no"
   }
).whenNotMatchedInsert(values=
   {
    "emp_id": "source.emp_id",
    "name" :"source.name",
    "city":"source.city",
    "country":"source.country",
    "contact_no":"source.contact_no"
   }
).execute()
```

(10)spark jobs ran

--

```
%sql
select * from dim_employee
```

(2) Spark jobs ran



AUDIT LOG FOR DELTA LAKE TABLE OPERATIONS

We are not able to perform DML operations in data bricks tables. Later when delta lake got introduced, we can perform all DML operations such as update, delete and insert in delta lake tables. In DWH/data base tables, there are multiple log tables such as process log table, error log table or audit log table. Audit log table used to track the metrics performed on any fact/dimension tables. Basically, audit log table will contain metrics such as no of records inserted/updated/deleted by particular query or by particular process . In delta lake, data bricks is already providing some features,

there is inbuilt function history. Using history, we can obtain all these values but still it’s not in standard format because we can see product information in dictionary format. It’s not suitable for relational data base tables. In most of the projects we used to keep our own audit log table.

Code:

```
from pyspark.sql.types import *
from pyspark.sql.functions import *
schema = StructType([
    StructField("emp_id",IntegerType(),True),
    StructField("name",StringType(),True),
    StructField("city",StringType(),True),
    StructField("country",StringType(),True),
    StructField("contact_no",IntegerType(),True)
])
data =[(2000,"Kelvin","New York","USA",4567888),
       (3000,"David","Dallas","USA",34567867),
       (4000,"Peter","Columbus","USA",689546323),
       (5000,"Rosy","Columbus","USA",689546323)]
df= spark.createDataFrame(data,schema)
display(df)
(3)spark jobs ran
--
%sql
CREATE OR REPLACE TABLE dim_employee(
  emp_id INT,
  name STRING,
  city STRING,
  country STRING,
  contact_no INT
)
USING DELTA
LOCATION "/FileStore/tables/delta_merge"
(4)spark jobs ran
OK
--
%sql
select * from dim_employee
(2)spark jobs ran
```

	emp_id	name	city	country	contact_no
1	1000	Michael	Chicago	USA	689546323
2	2000	Sarah	New York	USA	76345902
3	3000	David	Atlanta	USA	563456787

3 rows | 9.42 seconds runtime

```
display(df)
```

(3)spark jobs ran.

Table +

	emp_id	name	city	country	contact_no
1	2000	Kelvin	New York	USA	4567888
2	3000	David	Dallas	USA	34567867
3	4000	Peter	Columbus	USA	689546323
4	5000	Rosy	Columbus	USA	689546323

4 rows | 0.48 seconds runtime

```
--
METHOD-1 SPARK SQL
df.createOrReplaceTempView("source_view")
%sql
select * from source_view
```

(3)spark jobs ran

_sqldf: pyspark.sql.dataframe.DataFrame = [emp_id: integer, name: string ... 3 more fields]

Table +

	emp_id	name	city	country	contact_no
1	2000	Kelvin	New York	USA	4567888
2	3000	David	Dallas	USA	34567867
3	4000	Peter	Columbus	USA	689546323
4	5000	Rosy	Columbus	USA	689546323

4 rows | 0.80 seconds runtime

```
%sql
MERGE INTO dim_employee as target
USING source_view as source
ON target.emp_id=source.emp_id
WHEN MATCHED
THEN UPDATE SET
    target.name=source.name,
    target.city = source.city,
```

```
    target.country=source.country,
    target.contact_no = source.contact_no
WHEN NOT MATCHED THEN
INSERT(emp_id,name,country,contact_no)VALUES(emp_id,name,country,contact_no)
```

(13)spark jobs ran

Table +

	num_affected_rows	num_updated_rows	num_deleted_rows	num_inserted_rows
1	4	2	0	2

1 row | 19.23 seconds runtime

```
%sql
create table audit_log(operation STRING,
                       updated_time timestamp,
                       user_name STRING,
                       notebook_name STRING,
                       numTargetRowsUpdated int,
                       numTargetRowsInserted int,
                       numTargetRowsDeleted int)

%sql
select * from audit_log
```

Query returned no results

--

In order to get all the operations of historical information we can use history command.

```
display(delta_df.history())
```

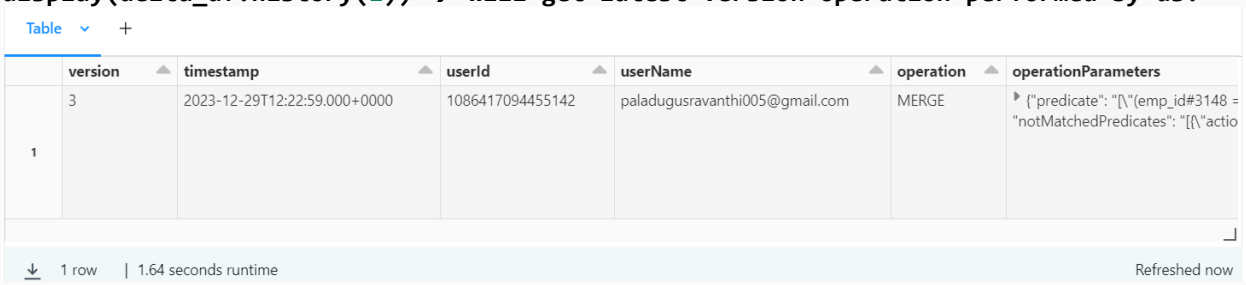
(2)spark jobs ran

	version	timestamp	userId	userName	operation	operationParame
1	3	2023-12-29T12:22:59.000+0000	1086417094455142	paladugusravanthi005@gmail.com	MERGE	{"predicate": "[\" "notMatchedPredi
2	2	2023-12-29T11:47:47.000+0000	1086417094455142	paladugusravanthi005@gmail.com	MERGE	{"predicate": "[\" "notMatchedPredi
3	1	2023-12-29T11:21:52.000+0000	1086417094455142	paladugusravanthi005@gmail.com	MERGE	{"predicate": "[\" [{\"actionType\":\"
4	0	2023-12-29T10:46:35.000+0000	1086417094455142	paladugusravanthi005@gmail.com	CREATE OR REPLACE TABLE	{"isManaged": "t

4 rows | 2.40 seconds runtime Refreshed 5 minutes ago

For audit log purpose if we want to get latest operations performed in the delta lake table for that we can give number 1,we will get latest 1 record.

display(delta_df.history(1)) → will get latest version operation performed by us.

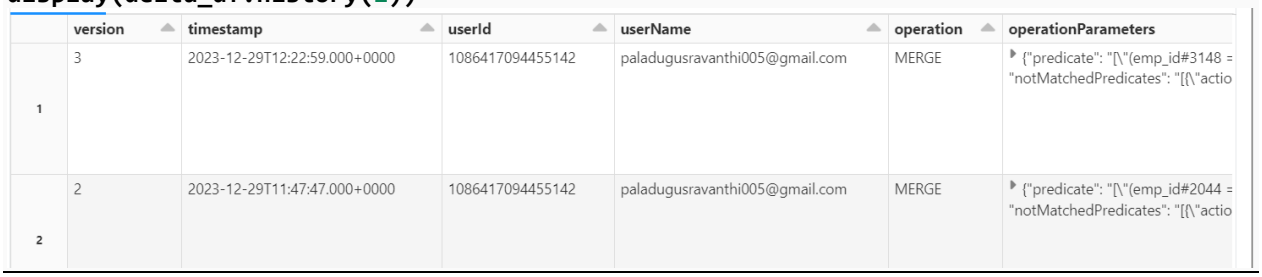


	version	timestamp	userId	userName	operation	operationParameters
1	3	2023-12-29T12:22:59.000+0000	1086417094455142	paladugusravanthi005@gmail.com	MERGE	{"predicate": "[\"(emp_id#3148 = "notMatchedPredicates": "[{\"actio

1 row | 1.64 seconds runtime — Refreshed now

If we want to get latest 2 version operations, we need to keep 2.

display(delta_df.history(2))



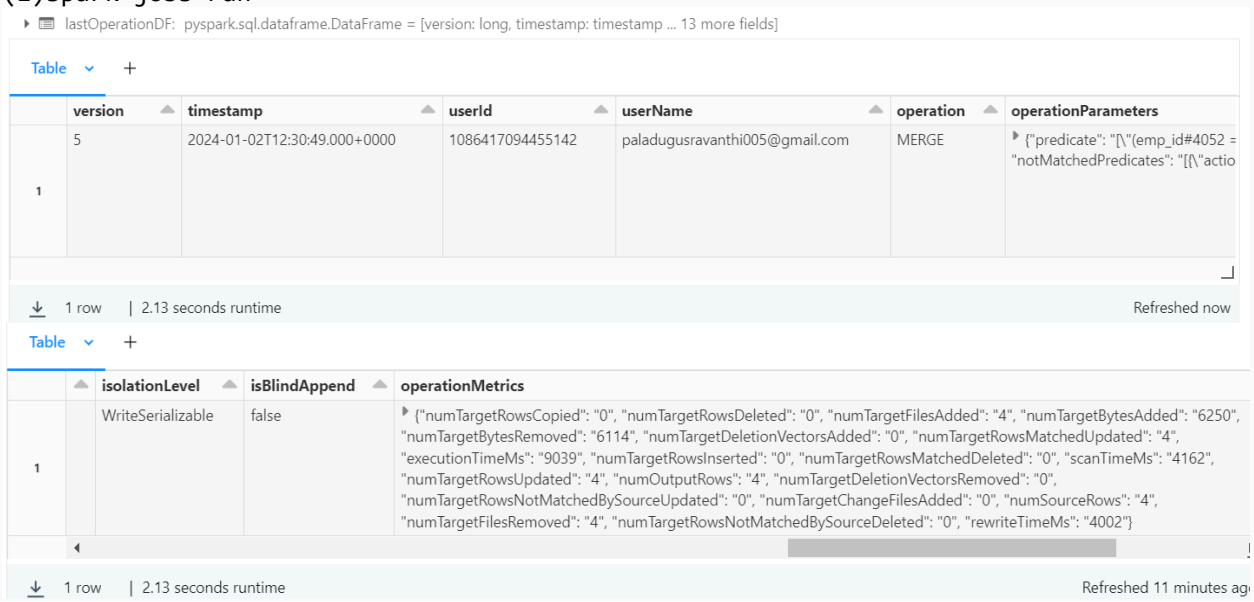
	version	timestamp	userId	userName	operation	operationParameters
1	3	2023-12-29T12:22:59.000+0000	1086417094455142	paladugusravanthi005@gmail.com	MERGE	{"predicate": "[\"(emp_id#3148 = "notMatchedPredicates": "[{\"actio
2	2	2023-12-29T11:47:47.000+0000	1086417094455142	paladugusravanthi005@gmail.com	MERGE	{"predicate": "[\"(emp_id#2044 = "notMatchedPredicates": "[{\"actio

---

CREATE DATA FRAME WITH LAST OPERATION IN DELTA TABLE

```
from delta.tables import *
delta_df = DeltaTable.forPath(spark,"/FileStore/tables/delta_merge")
lastOperationDF=delta_df.history(1) #get the last operation
display(lastOperationDF)
```

(1)spark jobs ran



lastOperationDF: pyspark.sql.dataframe.DataFrame = [version: long, timestamp: timestamp ... 13 more fields]

	version	timestamp	userId	userName	operation	operationParameters
1	5	2024-01-02T12:30:49.000+0000	1086417094455142	paladugusravanthi005@gmail.com	MERGE	{"predicate": "[\"(emp_id#4052 = "notMatchedPredicates": "[{\"actio

1 row | 2.13 seconds runtime — Refreshed now

	isolationLevel	isBlindAppend	operationMetrics
1	WriteSerializable	false	{"numTargetRowsCopied": "0", "numTargetRowsDeleted": "0", "numTargetFilesAdded": "4", "numTargetBytesAdded": "6250", "numTargetBytesRemoved": "6114", "numTargetDeletionVectorsAdded": "0", "numTargetRowsMatchedUpdated": "4", "executionTimeMs": "9039", "numTargetRowsInserted": "0", "numTargetRowsMatchedDeleted": "0", "scanTimeMs": "4162", "numTargetRowsUpdated": "4", "numOutputRows": "4", "numTargetDeletionVectorsRemoved": "0", "numTargetRowsNotMatchedBySourceUpdated": "0", "numTargetChangeFilesAdded": "0", "numSourceRows": "4", "numTargetFilesRemoved": "4", "numTargetRowsNotMatchedBySourceDeleted": "0", "rewriteTimeMs": "4002"}

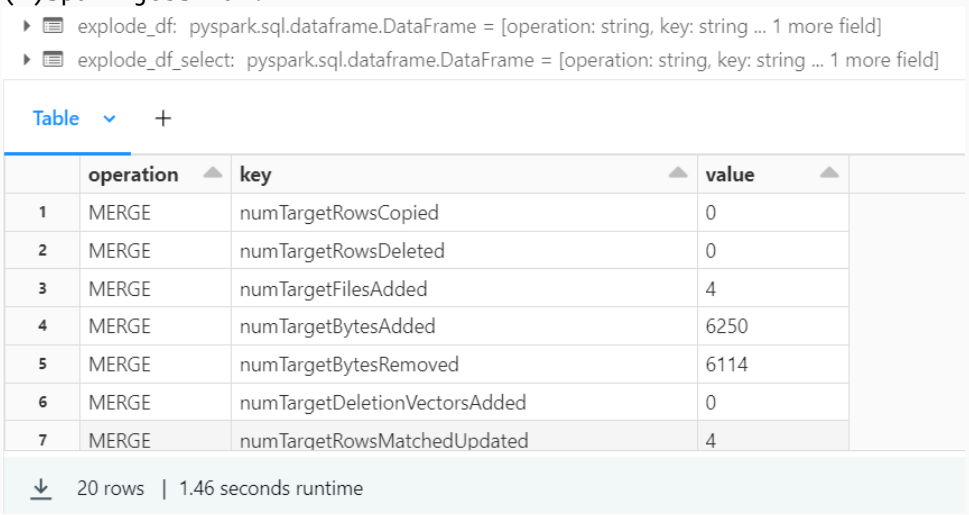
1 row | 2.13 seconds runtime — Refreshed 11 minutes ag

Here after merge statement , version got increased from 4 to 5. And we can see what operation got performed at latest.

EXPLODE OPERATION METRICS COLUMN

```
explode_df = lastOperationDF.select(lastOperationDF.operation,
explode(lastOperationDF.operationMetrics))
explode_df_select =
explode_df.select(explode_df.operation,explode_df.key,explode_df.value.cast('int'))
display(explode_df_select)
```

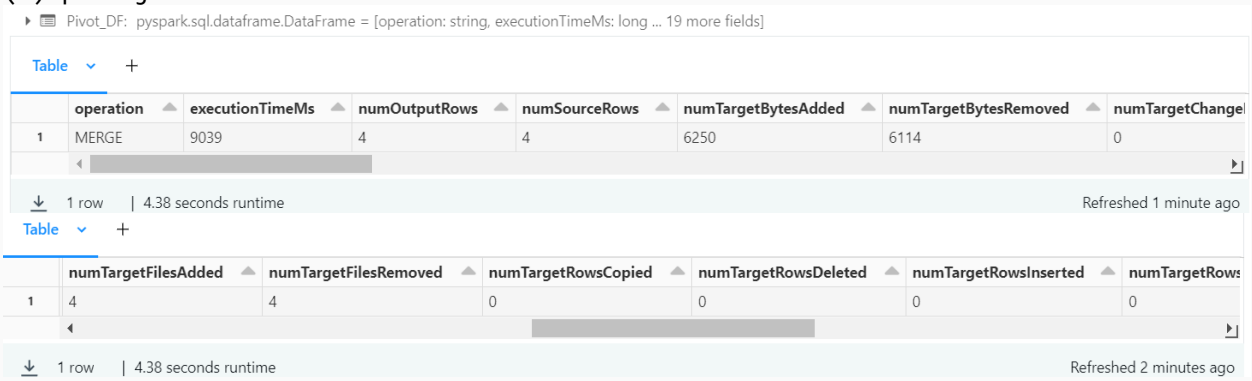
(2)spark jobs ran.



#PIVOT OPERATION TO CONVERT ROWS TO COLUMNS

```
Pivot_DF = explode_df_select.groupBy("operation").pivot("key").sum("value")
display(Pivot_DF)
```

(8)spark jobs ran.



#SELECTING ONLY COLUMNS NEEDED FOR OUR AUDIT LOG TABLE

```
Pivot_DF_select =
Pivot_DF.select(Pivot_DF.operation,Pivot_DF.numTargetRowsUpdated,Pivot_DF.numTargetRo
wsInserted,Pivot_DF.numTargetRowsDeleted)

display(Pivot_DF_select)
```

(4) Spark Jobs

Pivot_DF_select: pyspark.sql.dataframe.DataFrame = [operation: string, numTargetRowsUpdated: long ... 2 more fields]

Table +

	operation	numTargetRowsUpdated	numTargetRowsInserted	numTargetRowsDeleted
1	MERGE	4	0	0

1 row | 2.38 seconds runtime

```
#ADDING EXTRA INFORMATION ADD NOTEBOOK PARAMETERS SUCH AS USERNAME,NOTEBOOK PATH ETC
auditDF=Pivot_DF_select.withColumn("user_name",lit(dbutils.notebook.entry_point.getDb
utils().notebook().getContext().userName().get())).withColumn("notebook_name",lit(dbu
tils.notebook.entry_point.getDbutils().notebook().getContext().notebookPath().get()))
.withColumn("updated_time",lit(current_timestamp()))
display(auditDF)
```

(4)spark jobs ran

Table +

	operation	numTargetRowsUpdated	numTargetRowsInserted	numTargetRowsDeleted	user_name	notebook_name
1	MERGE	4	0	0	paladugusravanthi005@gmail.com	/Users/paladugusra

1 row | 2.56 seconds runtime Refreshed now

```
#REARRANGING COLUMNS IN DATAFRAME TO MATCH IT WITH AUDIT LOG TABLE
auditDF_select =
auditDF.select(auditDF.operation,auditDF.updated_time,auditDF.user_name,auditDF.noteb
ook_name,auditDF.numTargetRowsUpdated,auditDF.numTargetRowsInserted,auditDF.numTarget
RowsDeleted)

display(auditDF_select)
```

(4)spark jobs ran.

Table +

	operation	updated_time	user_name	notebook_name
1	MERGE	2024-01-02T13:14:59.542+0000	paladugusravanthi005@gmail.com	/Users/paladugusravanthi005@gmail.com/Untitled Notebook 2024-01-02 12:4

1 row | 1.62 seconds runtime Refreshed now

```
#CREATE TEMP VIEW ON DATAFRAME
auditDF_select.createOrReplaceTempView("audit")
--
%sql
select * from audit
```

(4)spark jobs ran

_sqldf: pyspark.sql.dataframe.DataFrame = [operation: string, updated_time: timestamp ... 5 more fields]

Table +

	operation	updated_time	user_name	notebook_name
1	MERGE	2024-01-02T13:18:09.124+0000	paladugusravanthi005@gmail.com	/Users/paladugusravanthi005@gmail.com/Untitled Notebook 2024-01-02 12:4

1 row | 1.99 seconds runtime Refreshed now

```
%sql
select * from audit_log
```

(1) Spark Jobs

	operation	updated_time	user_name	notebook_name	numTargetRowsUpdated
1	MERGE	2022-01-07T11:44:48.514+0000	audaciousazure@gmail.com	/Users/audaciousazure@gmail.com/Channel/Raja's Data Engineering/Pyspark Merge Statement	1

```
#INSERT AUDIT DATA INTO AUDIT LOG TABLE
%sql

insert into audit_log select * from audit
```

(10)spark jobs ran.

	num_affected_rows	num_inserted_rows
1	1	1

Showing all 1 rows.

```
%sql
select * from audit_log
```

(2)spark jobs ran

	operation	updated_time	user_name	notebook_name	numTargetRowsUpdated
1	MERGE	2022-01-07T11:44:48.514+0000	audaciousazure@gmail.com	/Users/audaciousazure@gmail.com/Channel/Raja's Data Engineering/Pyspark Merge Statement	1
2	MERGE	2022-01-07T14:55:50.164+0000	audaciousazure@gmail.com	/Users/audaciousazure@gmail.com/Channel/Raja's Data Engineering/Pyspark Merge Statement	2

Showing all 2 rows.

		notebook_name	numTargetRowsUpdated	numTargetRowsInserted	numTargetRowsDeleted
1	zure@gmail.com	/Users/audaciousazure@gmail.com/Channel/Raja's Data Engineering/Pyspark Merge Statement	1	1	0
2	zure@gmail.com	/Users/audaciousazure@gmail.com/Channel/Raja's Data Engineering/Pyspark Merge Statement	2	2	0

Showing all 2 rows.

SLOWLY CHANGING DIMENSION

Slowly Changing dimension is nothing but change of attribute or value of entities over a period of time. Let's say we are having customer entity, customer table and having an attribute address. Let's say customer is staying in one address today and two years down the line, he is changing his address. As a result, we have to update the address in our data ware housing solution also. So, within customer entity we have to update address field with latest address. This is called Slowly Changing Dimension. In order to handle Slowly Changing Dimension, there are various methods in data warehousing solution. SCD Type 1 , SCD type 2 and SCD type 3.

In SCD Type 1 let's say address got changed, we are getting new address and that will be overwritten. That is called SCD Type 1. In this SCD Type 1 , we are going to lose the history. Let's say after certain period of time, we want to retrieve previous address which can't be done by SCD Type 1 because we are not maintaining history here, address got overwritten. Coming to SCD Type-2, we are getting new address, we will make previous version of record inactive and we will insert new record with new address. This is SCD Type 2. Basically, we are introducing new record for each change of attribute. In this method we are maintaining history but at the same time we are creating duplicate records of primary keys. Coming to SCD Type-3 , instead of overwriting or introducing new record, we are going to create new column for the updated value. Let's say we are going to get new address then previous address we are going to make a column as a previous address and the new address we will make a new column as current address. In this way we are introducing new column for each updated value . So, these are the 3 different ways to handle slowly changing dimension in data warehousing solution.

CODE:

```
%sql
CREATE OR REPLACE TABLE scd2Demo (
                                    pk1 INT,
                                    pk2 STRING,
                                    dim1 INT,
                                    dim2 INT,
                                    dim3 INT,
                                    dim4 INT,
                                    active_status STRING,
                                    start_date TIMESTAMP,
                                    end_date TIMESTAMP)
                        USING DELTA
                        LOCATION '/FileStore/tables/scd2Demo'
(3)spark jobs ran
Ok
--
%sql
insert into scd2Demo
values(111,'Unit1',200,500,800,400,'Y',current_timestamp(),'9999-12-31');
insert into scd2Demo
values(222,'Unit2',900,Null,700,100,'Y',current_timestamp(),'9999-12-31');
insert into scd2Demo
values(333,'Unit3',300,900,250,650,'Y',current_timestamp(),'9999-12-31');

(12)spark jobs ran
```

	num_affected_rows	num_inserted_rows
1	1	1

Showing all 1 rows.

```
--
Creating delta table instance

from delta import *
targetTable = DeltaTable.forPath(spark,"/FileStore/tables/scd2Demo")
targetDF=targetTable.toDF()
display(targetDF)

2 spark jobs ran
```

Table +

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2024-01-04T13:46:01.418+0000	9999-12-31T00:00:00.000+0000
2	333	Unit3	300	900	250	650	Y	2024-01-04T13:46:18.872+0000	9999-12-31T00:00:00.000+0000
3	222	Unit2	900	null	700	100	Y	2024-01-04T13:46:13.626+0000	9999-12-31T00:00:00.000+0000

3 rows | 9.19 seconds runtime Refreshed now

```
--

from pyspark.sql.types import *
from pyspark.sql.functions import *
schema = StructType([StructField("pk1",StringType(),True), \
                     StructField("pk2",StringType(),True), \
                     StructField("dim1",IntegerType(),True), \
                     StructField("dim2",IntegerType(),True), \
                     StructField("dim3",IntegerType(),True), \
                     StructField("dim4",IntegerType(),True)])
--

data = [(111,'Unit1',200,500,800,400),
        (222,"Unit2",800,1300,800,500),
        (444,"Unit4",100,None,700,300)]
sourceDF= spark.createDataFrame(data,schema)
display(sourceDF)
3 spark jobs ran
```

sourceDF: pyspark.sql.dataframe.DataFrame

```
pk1: string
pk2: string
dim1: integer
dim2: integer
dim3: integer
dim4: integer
```

Table +

	pk1	pk2	dim1	dim2	dim3	dim4
1	111	Unit1	200	500	800	400
2	222	Unit2	800	1300	800	500
3	444	Unit4	100	null	700	300

3 rows | 3.59 seconds runtime

--

```
joinDF = sourceDF.join(targetDF,(sourceDF.pk1==targetDF.pk1) & \
                         (sourceDF.pk2 == targetDF.pk2) & \
                         (targetDF.active_status=="Y"),"leftouter") \
                  .select(sourceDF["*"], \
                          targetDF.pk1.alias("target_pk1"), \
                          targetDF.pk2.alias("target_pk2"), \
                          targetDF.dim1.alias("target_dim1"), \
                          targetDF.dim2.alias("target_dim2"), \
                          targetDF.dim3.alias("target_dim3"), \
                          targetDF.dim4.alias("target_dim4"))
display(joinDF)
```

used alias in above command because source and target columns are same so used alias.

(3)spark jobs

joinDF: pyspark.sql.dataframe.DataFrame

```
pk1: string
pk2: string
dim1: integer
dim2: integer
dim3: integer
dim4: integer
target_pk1: integer
target_pk2: string
target_dim1: integer
target_dim2: integer
target_dim3: integer
target_dim4: integer
```

Table +

	pk1	pk2	dim1	dim2	dim3	dim4	target_pk1	target_pk2	target_dim1	target_dim2	target_dim3	target_dim4
1	111	Unit1	200	500	800	400	111	Unit1	200	500	800	400
2	222	Unit2	800	1300	800	500	222	Unit2	900	null	700	100
3	444	Unit4	100	null	700	300	null	null	null	null	null	null

3 rows | 3.96 seconds runtime Refreshed 3 minute

--

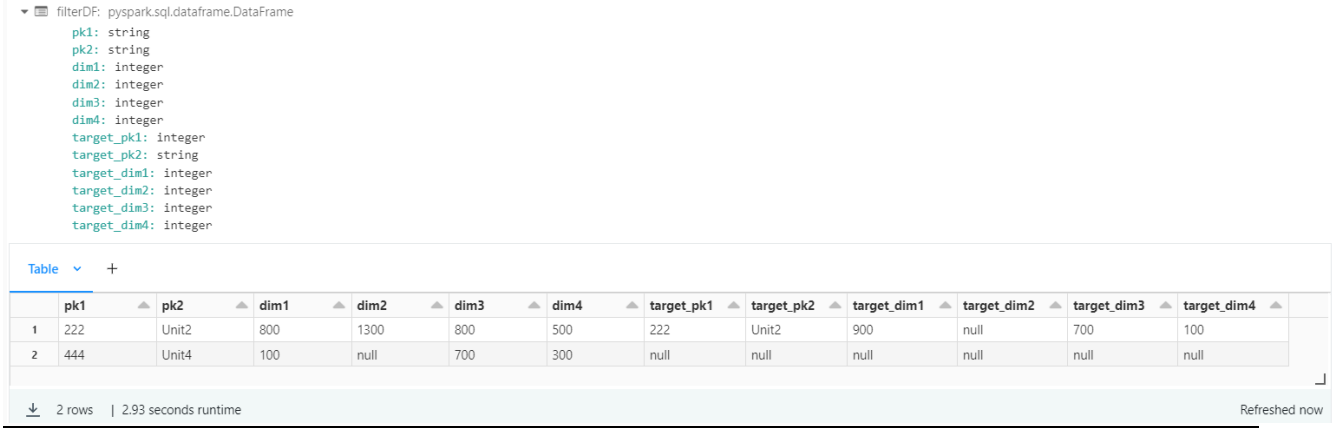
xxhash64 is used to concatenate all columns.

```
filterDF= joinDF.filter(xxhash64(joinDF.dim1,joinDF.dim2,joinDF.dim3,joinDF.dim4)
```

```
!=xxhash64(joinDF.target_dim1,joinDF.target_dim2,joinDF.target_dim3,joinDF.target_dim4))

display(filterDF)
```

(3)spark jobs ran

There is no change for 111 pk1 so ignore that row. We are getting records here only when there is a change.

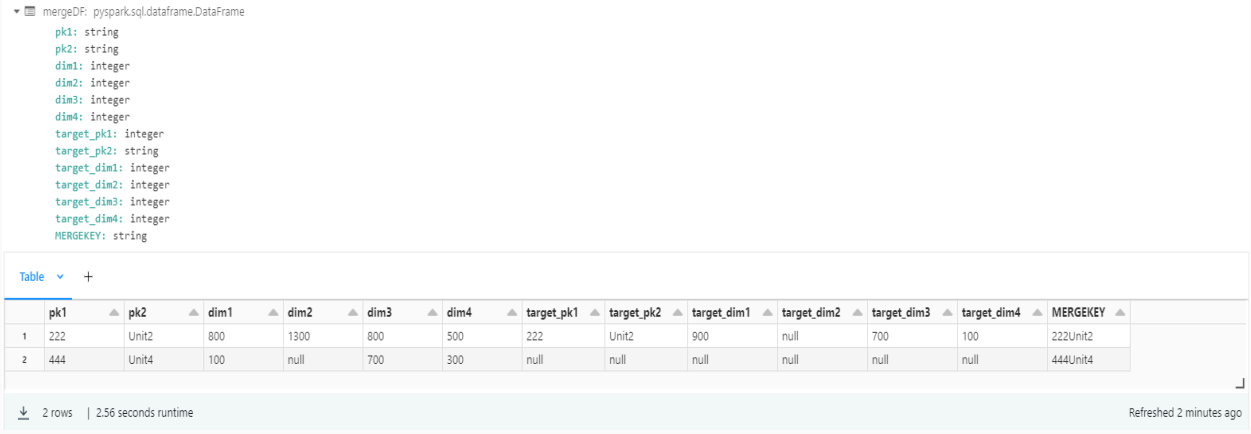


	pk1	pk2	dim1	dim2	dim3	dim4	target_pk1	target_pk2	target_dim1	target_dim2	target_dim3	target_dim4
1	222	Unit2	800	1300	800	500	222	Unit2	900	null	700	100
2	444	Unit4	100	null	700	300	null	null	null	null	null	null

```
mergeDF = filterDF.withColumn("MERGEKEY",concat(filterDF.pk1,filterDF.pk2))
display(mergeDF)
```

here we are creating merge key by combining all my key columns pk1 and pk2

(3)spark jobs ran.



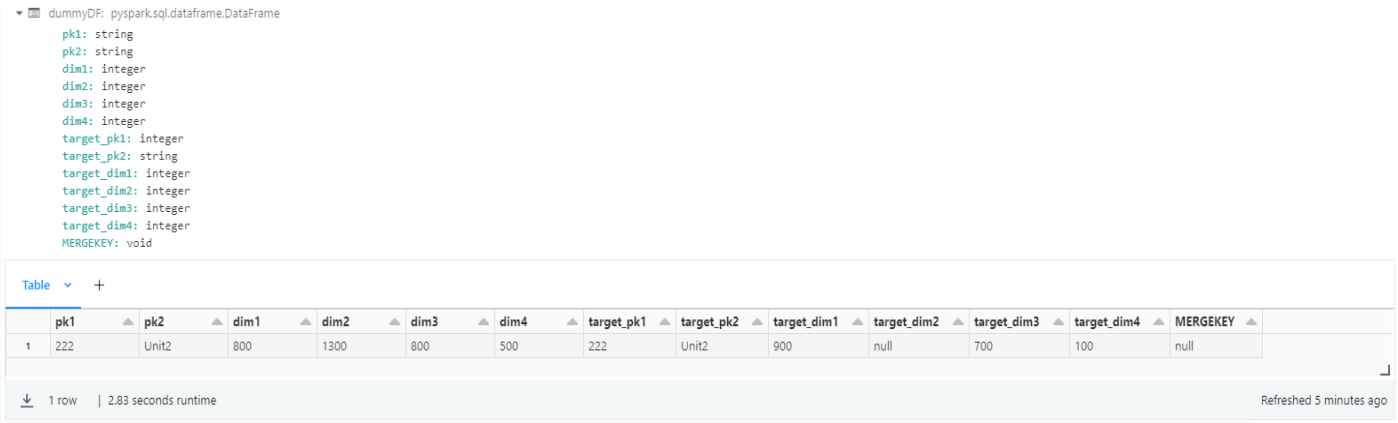
	pk1	pk2	dim1	dim2	dim3	dim4	target_pk1	target_pk2	target_dim1	target_dim2	target_dim3	target_dim4	MERGEKEY
1	222	Unit2	800	1300	800	500	222	Unit2	900	null	700	100	222Unit2
2	444	Unit4	100	null	700	300	null	null	null	null	null	null	444Unit4

--

Now creating one more merge key one more dummy record only for matched records which is 222Unit2 where target_pk1 is not null

```
dummyDF = filterDF.filter("target_pk1 is not null").withColumn("MERGEKEY",lit(None))
display(dummyDF)
```

(3)spark jobs ran

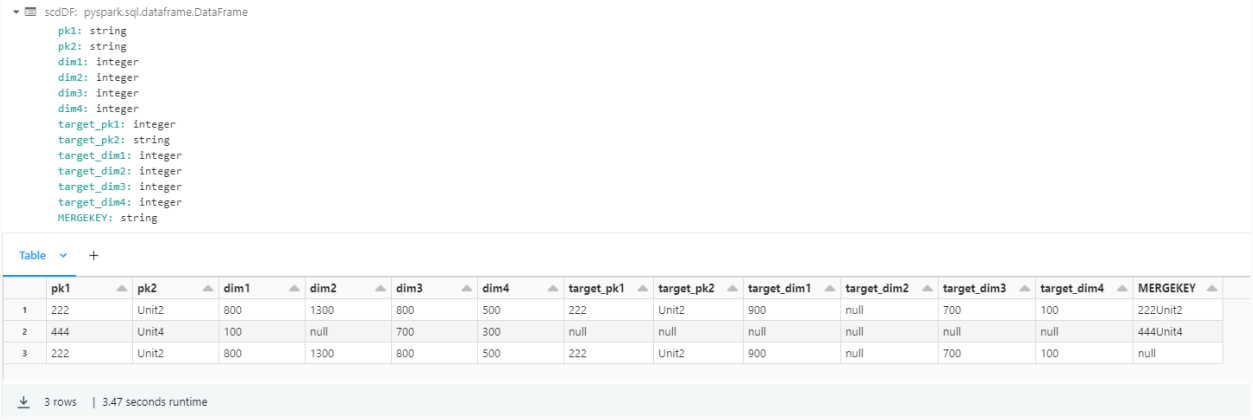


--

Now we are performing union operation between the 2 data frames mergeDF and dummyDF

```
scdDF= mergeDF.union(dummyDF)
display(scdDF)
```

(3) Spark jobs ran



--

```
targetTable.alias("target").merge(
    source = scdDF.alias("source"),
    condition = "concat(target.pk1,target.pk2) = source.MERGEKEY and
target.active_status ='Y' "
   ).whenMatchedUpdate(set=
      {
        "active_status" : " 'N' ",
         "end_date" : "current_date"
      }
   ).whenNotMatchedInsert(values =
      {
        "pk1" : "source.pk1",
        "pk2" : "source.pk2",
        "dim1" : "source.dim1",
        "dim2" : "source.dim2",
```

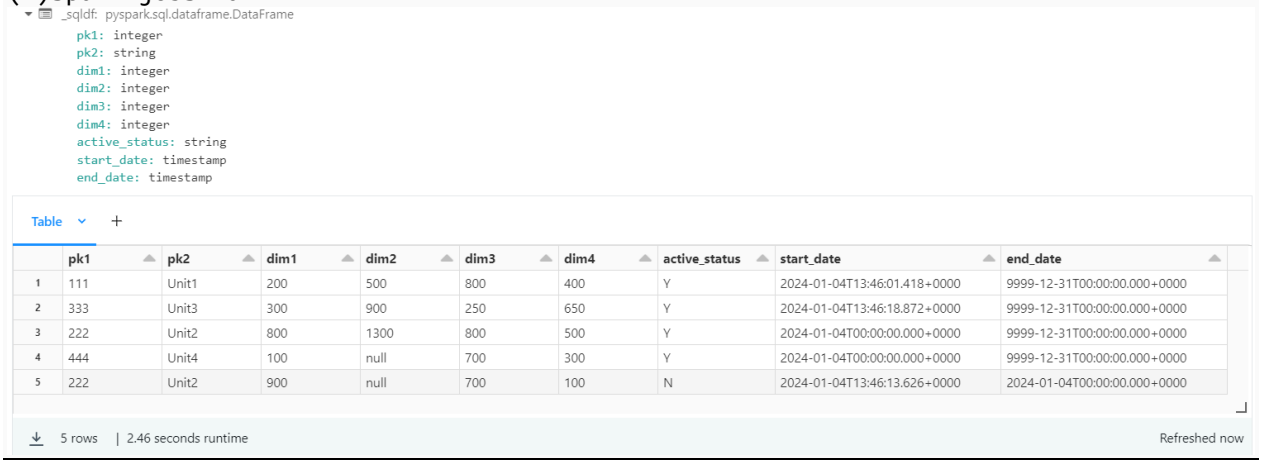
```
        "dim3" : "source.dim3",
        "dim4" : "source.dim4" ,
        "active_status" : " 'Y' ",
        "start_date" : "current_date",
        "end_date" : """to_date('9999-12-31','yyyy-MM-dd')"""
      }
      ).execute()
```

7 spark jobs ran

--

```
%sql
select * from scd2demo
```

(2)spark jobs ran



DELTA TABLE : TIME TRAVEL

Time travel is traversing or travelling back and forth through snapshots of delta lake table. Snapshot is complete state of a table during a particular time. Let's say we are having 100 records in a delta lake table today. Later we are adding 10 records. Now the table is going to contain 110 records. So, there are 2 versions, in previous version it is containing 100 records that was one snapshot and later we added 10 records. So, the table is containing 110 records , it is another snapshot of the table. So here we are having two versions we can go back and forth. For example, I want to capture the previous version of the table or previous snapshot of the table then I can go back using some methodology that is called time travel in data bricks.

Code:

#TABLE HISTORY

In order to perform time travel, we need to get list of versions or snapshots so that can be achieved using below command

```
%sql
describe history scd2Demo
```

(1)spark jobs ran

	version	timestamp	userId	userName	operation	operationParameters
1	4	2022-05-15T10:02:45.000+0000	1383370256722896	audaciousazure@gmail.com	MERGE	{"predicate": "((concat(CAS "matchedPredicates": "[{\"ac
2	3	2022-05-15T09:55:58.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "parti
3	2	2022-05-15T09:55:53.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "parti
4	1	2022-05-15T09:55:47.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "parti
5	0	2022-05-15T09:55:04.000+0000	1383370256722896	audaciousazure@gmail.com	CREATE OR REPLACE TABLE	{"isManaged": "false", "des

Showing all 5 rows.

This is the history of particular table, there are 5 different versions starting from 0 till 4 and each and every version provides audit related information such as what kind of operation performed and how many records got impacted. It will give complete information .

If I want to see state of version-1,then we can do time travel. Time travel we can do in 2 approaches using SQL and 2nd is Pyspark.

PYSPARK APPROACHES

METHOD-1 : PYSPARK TIMESTAMP + TABLE

Time travel we can achieve using either table name or location of the table. Time travel we can do using either version or timestamp.

```
df= spark.read \
    .format("delta") \
    .option("timestampAsOf", "2022-05-15T09:55:47.000+0000") \
    .table("scd2Demo")

display(df)
```

(2)spark jobs ran

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2022-05-15T09:55:42.211+0000	9999-12-31T0(
2	222	Unit2	900	null	700	100	Y	2022-05-15T09:55:51.669+0000	9999-12-31T0(

Showing all 2 rows.

METHOD-2 : PYSPARK- TIMESTAMP + PATH

```
df= spark.read \
    .format("delta") \
    .option("timestampAsOf", "2022-05-15T09:55:53.000+0000") \
    .load("/FileStore/tables/scd2Demo")

display(df)
```

(2)spark jobs ran

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2022-05-15T09:55:42.211+0000	9999-12-31T0(
2	222	Unit2	900	null	700	100	Y	2022-05-15T09:55:51.669+0000	9999-12-31T0(

Showing all 2 rows.

METHOD-3 : PYSPARK- VERSION + PATH

```
df= spark.read \
    .format("delta") \
```

```
    .option("versionAsOf",3) \          → if we want to see version 3 data then 3
    .load("/FileStore/tables/scd2Demo")

display(df)
```

2 spark jobs ran

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2022-05-15T09:55:42.211+0000	9999-12-31T0(
2	333	Unit3	300	900	250	650	Y	2022-05-15T09:55:55.936+0000	9999-12-31T0(
3	222	Unit2	900	null	700	100	Y	2022-05-15T09:55:51.669+0000	9999-12-31T0(

Showing all 3 rows.

This is the snapshot during version 3.

METHOD-4 : PYSPARK- VERSION + TABLE

```
df= spark.read \
    .format("delta") \
    .option("versionAsOf",2) \
    .table("scd2Demo")

display(df)
```

2 spark jobs ran

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2022-05-15T09:55:42.211+0000	9999-12-31T0(
2	222	Unit2	900	null	700	100	Y	2022-05-15T09:55:51.669+0000	9999-12-31T0(

Showing all 2 rows.

SQL APPROACHES

METHOD-5 : SQL-VERSION + TABLE

```
%sql
SELECT * FROM scd2demo VERSION AS OF 2
```

2 spark jobs ran

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2022-05-15T09:55:42.211+0000	9999-12-31T0(
2	222	Unit2	900	null	700	100	Y	2022-05-15T09:55:51.669+0000	9999-12-31T0(

Showing all 2 rows.

METHOD-6 : SQL-VERSION + PATH

```
%sql
select * from delta.`/FileStore/tables/scd2Demo` VERSION AS OF 2
```

2 spark jobs ran.

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2022-05-15T09:55:42.211+0000	9999-12-31T0(
2	222	Unit2	900	null	700	100	Y	2022-05-15T09:55:51.669+0000	9999-12-31T0(

Showing all 2 rows.

METHOD-7 : SQL-TIMESTAMP + TABLE

```
%sql
SELECT * FROM scd2Demo TIMESTAMP AS OF "2022-05-15T09:55:53.000+0000"
```

2 spark jobs ran

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2022-05-15T09:55:42.211+0000	9999-12-31T0(
2	222	Unit2	900	null	700	100	Y	2022-05-15T09:55:51.669+0000	9999-12-31T0(

Showing all 2 rows.

METHOD-8 : SQL-TIMESTAMP + PATH

```
%sql
SELECT * FROM delta.`/FileStore/tables/scd2Demo` TIMESTAMP AS OF "2022-05-
15T09:55:53.000+0000"
```

2 spark jobs ran

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2022-05-15T09:55:42.211+0000	9999-12-31T0(
2	222	Unit2	900	null	700	100	Y	2022-05-15T09:55:51.669+0000	9999-12-31T0(

Showing all 2 rows.

DELTA TABLE- RESTORE COMMAND

In traditional data bases or data warehouses , when we are performing series of DML operations and later we realized there is a mistake and something went wrong with the DML operation then we can go back to the previous state using rollback command. even in some other data bases or data warehouses, we have concepts called flashback or snapshot. Restore is one of such command. Using restore, we can go back to previous version of the delta table. We can make the delta table permanently to previous version or previous timestamp that is called restore command.

In time travel we can go back and forth across multiple versions of the table as one time activity but coming to restore when we are applying restore command to delta table to certain previous version, it will make table permanent to that previous version so that is the difference between time travel and vacuum.

SQL CREATION TABLE:

```
%sql
CREATE OR REPLACE TABLE scd2Demo (
                                    pk1 INT,
                                    pk2 STRING,
                                    dim1 INT,
                                    dim2 INT,
                                    dim3 INT,
                                    dim4 INT,
                                    active_status STRING,
                                    start_date TIMESTAMP,
                                    end_date TIMESTAMP)
                        USING DELTA
                        LOCATION '/FileStore/tables/scd2Demo'
```

PYSPARK APPROACH FOR CREATING DELTA TABLE

```
from delta.tables import *

DeltaTable.create(spark) \
    .tableName("scd2Demo") \
    .addColumn("pk1","INT") \
    .addColumn("pk2","STRING") \
    .addColumn("dim1","INT") \
    .addColumn("dim2","INT") \
    .addColumn("dim3","INT") \
    .addColumn("dim4","INT") \
    .addColumn("active_status","STRING") \
    .addColumn("start_date","TIMESTAMP") \
    .addColumn("end_date","TIMESTAMP") \
    .location("/FileStore/tables/scd2Demo") \
    .execute()
```

```
%sql
select * from scd2demo  → data already inserted from above example.
```

2 spark jobs ran

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2022-05-16T07:34:38.309+0000	9999-12-31T00:0
2	333	Unit3	300	900	250	650	Y	2022-05-16T07:34:44.933+0000	9999-12-31T00:0
3	888	Unit8	300	900	250	650	Y	2022-05-16T07:34:55.280+0000	9999-12-31T00:0
4	666	Unit6	100	500	800	400	Y	2022-05-16T07:34:48.109+0000	9999-12-31T00:0
5	222	Unit2	900	null	700	100	Y	2022-05-16T07:34:41.679+0000	9999-12-31T00:0

Showing all 5 rows.

```
--
```

CREATE DELTA TABLE INSTANCE

```
from delta import *
targetTable = DeltaTable.forPath(spark,"/FileStore/tables/scd2Demo")
```

Delta Table Instance we will call as target table.

Note: For Pyspark approach, we are creating delta table instance in order to capture version history of table. Whereas in SQL approach no need of instance, directly we can use describe history command.

LIST DOWN VERSION HISTORY

```
display(targetTable.history())
```

1 spark job ran

	version	timestamp	userId	userName	operation	operationParameters
1	8	2022-05-16T07:35:07.000+0000	1383370256722896	audaciousazure@gmail.com	UPDATE	{"predicate": "(pk1#13298 =
2	7	2022-05-16T07:35:02.000+0000	1383370256722896	audaciousazure@gmail.com	DELETE	{"predicate": "[\"(spark_catal
3	6	2022-05-16T07:34:57.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitio
4	5	2022-05-16T07:34:53.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitio
5	4	2022-05-16T07:34:50.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitio
6	3	2022-05-16T07:34:46.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitio
7	2	2022-05-16T07:34:43.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitio

Showing all 9 rows.

RESTORE USING VERSION NUMBER

If I want to restore version from 8 to 6, If I want to make table to previous version 6 permanently, so for that we need to use below command

```
targetTable.restoreToVersion(6)  #restore table to oldest version
```

15 spark jobs ran

```
Out[3]: DataFrame[table_size_after_restore: bigint, num_of_files_after_restore: bigint, num_removed_files: bigint, num_restored_files: bigint, removed_files_size: bigint, restored_files_size: bigint]

Command took 42.46 seconds -- by audaciousazure@gmail.com at 5/22/2022, 5:21:09 PM on demo
```

--

```
%sql
select * from scd2demo
```

4 spark jobs ran

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	666	Unit6	200	500	800	400	Y	2022-05-16T07:34:48.109+0000	9999-12-31T00:0
2	111	Unit1	200	500	800	400	Y	2022-05-16T07:34:38.309+0000	9999-12-31T00:0
3	333	Unit3	300	900	250	650	Y	2022-05-16T07:34:44.933+0000	9999-12-31T00:0
4	888	Unit8	300	900	250	650	Y	2022-05-16T07:34:55.280+0000	9999-12-31T00:0
5	777	Unit7	900	null	700	100	Y	2022-05-16T07:34:51.079+0000	9999-12-31T00:0
6	222	Unit2	900	null	700	100	Y	2022-05-16T07:34:41.679+0000	9999-12-31T00:0

Showing all 6 rows.

It restored permanently to previous version 6.

--

RESTORE USING TIME STAMP

If I want to restore back to one particular timestamp, we can restore using below command.

```
targetTable.restoreToTimestamp('2022-05-16T07:34:50.000+000') #restore to a specific timestamp
```

```
▸ (18) Spark Jobs
Out[4]: DataFrame[table_size_after_restore: bigint, num_of_files_after_restore: bigint, num_removed_files: bigint, num_restored_files: bigint, removed_files_size: bigint, restored_files_size: bigint]

Command took 14.76 seconds -- by audaciousazure@gmail.com at 5/22/2022, 5:23:13 PM on demo
```

```
%sql
select * from scd2demo
```

3 spark jobs ran

	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	666	Unit6	200	500	800	400	Y	2022-05-16T07:34:48.109+0000	9999-12-31T00:0
2	111	Unit1	200	500	800	400	Y	2022-05-16T07:34:38.309+0000	9999-12-31T00:0
3	333	Unit3	300	900	250	650	Y	2022-05-16T07:34:44.933+0000	9999-12-31T00:0
4	222	Unit2	900	null	700	100	Y	2022-05-16T07:34:41.679+0000	9999-12-31T00:0

Showing all 4 rows.

It restored permanently to 4th version timestamp.

--

display(targetTable.history())

1 spark job

	version	timestamp	userId	userName	operation	operationParameters
1	10	2022-05-22T11:53:23.000+0000	1383370256722896	audaciousazure@gmail.com	RESTORE	{"version": null, "timestamp":
2	9	2022-05-22T11:51:46.000+0000	1383370256722896	audaciousazure@gmail.com	RESTORE	{"version": "6", "timestamp":
3	8	2022-05-16T07:35:07.000+0000	1383370256722896	audaciousazure@gmail.com	UPDATE	{"predicate": "(pk1#13298 =
4	7	2022-05-16T07:35:02.000+0000	1383370256722896	audaciousazure@gmail.com	DELETE	{"predicate": "[\"(spark_catal
5	6	2022-05-16T07:34:57.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitio
6	5	2022-05-16T07:34:53.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitio
7	4	2022-05-16T07:34:50.000+0000	1383370256722896	audaciousazure@gmail.com	WRITE	{"mode": "Append", "partitio

Showing all 11 rows.

Previously it is only till version 8, now 2 more versions got added. Even if we are performing restore operation, it will be recorded as a separate version(restore 9th version for version and restore 10th version is with timestamp)

DELTA LAKE – OPTIMIZE COMMAND – FILE COMPACTION

Optimize command used to compact smaller size files in delta lake. In any big data processing, keeping too many smaller size files or too few big size files are not suitable. It will hit the performance. We should always go for optimal sized partitions or files. In spark by default partition block size is 128MB but that is not magic number. Depending on use case we can increase or decrease . we can even have 32MB,64MB,128MB,256MB or 512 MB. We can keep any size according to our use case. But by default, it is 128MB. When we are working with delta lake development, we used to create many number of smaller size files. In order to compact them to optimal size, we can go for optimize command.

```
%sql
CREATE OR REPLACE TABLE scd2Demo (
                                    pk1 INT,
                                    pk2 STRING,
                                    dim1 INT,
                                    dim2 INT,
                                    dim3 INT,
                                    dim4 INT,
                                    active_status STRING,
                                    start_date TIMESTAMP,
                                    end_date TIMESTAMP)
```

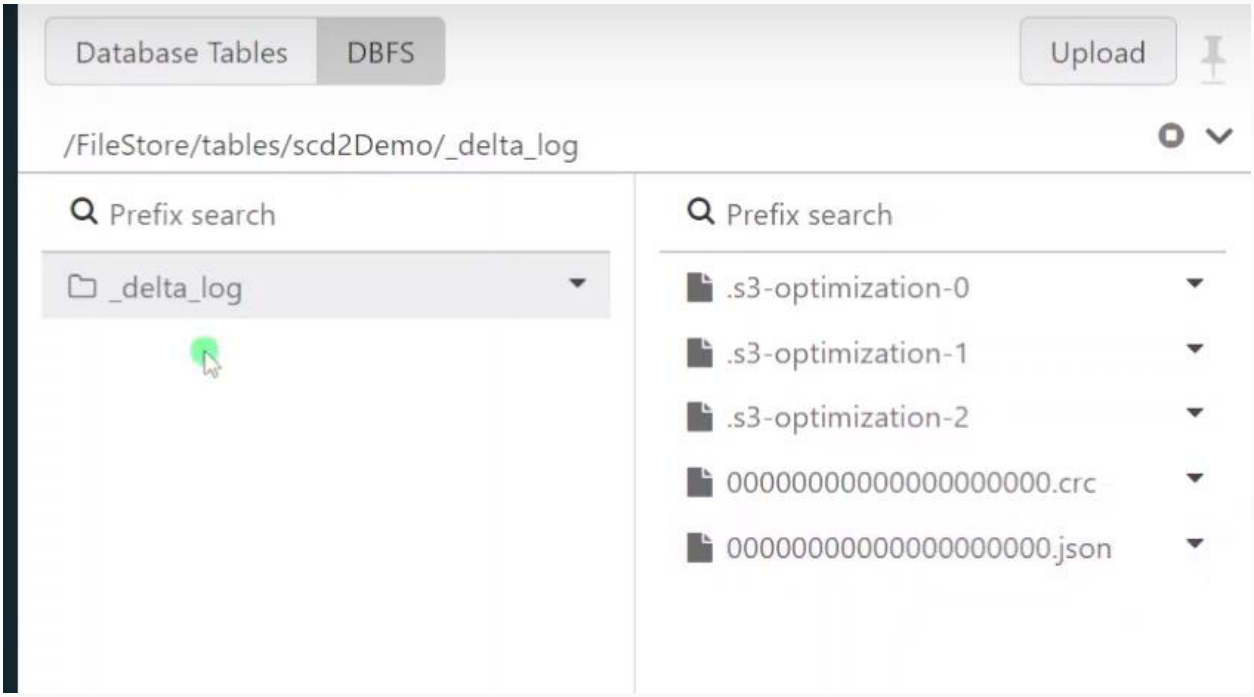
```
                    USING DELTA
                    LOCATION '/FileStore/tables/scd2Demo'
```

3 spark jobs ran

Ok

--

In DATA → DBFS→ FILESTORE/TABLES/scd2Demo → only log files was created with 0 version i.e.; only table definition data will be there. We haven't inserted any data so no data files are there in scd2Demo.

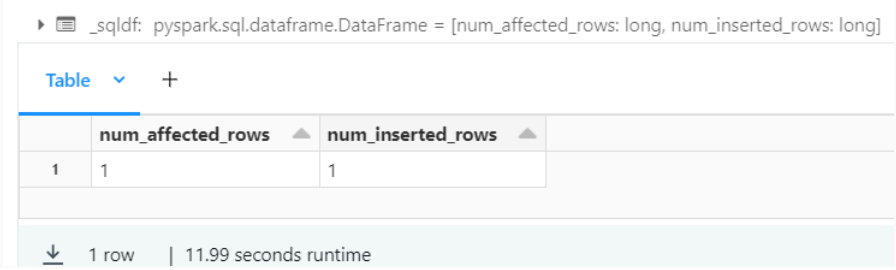


--

For each insert operation, it will create separate file which means it will create 3 different data files for this delta table

```
%sql
insert into scd2Demo
values(111,'Unit1',200,500,800,400,'Y',current_timestamp(),'9999-12-31');
insert into scd2Demo
values(222,'Unit2',900,Null,700,100,'Y',current_timestamp(),'9999-12-31');
insert into scd2Demo
values(333,'Unit3',300,900,250,650,'Y',current_timestamp(),'9999-12-31');
```

12 spark jobs ran



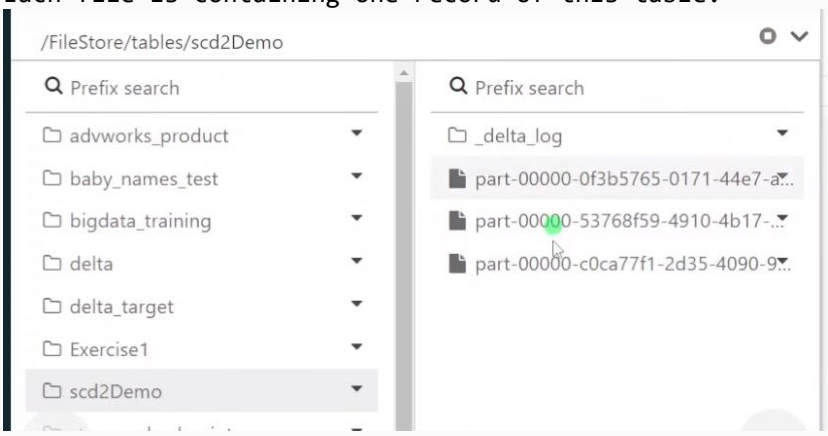
```
%sql
select * from scd2demo
```

2 spark jobs ran

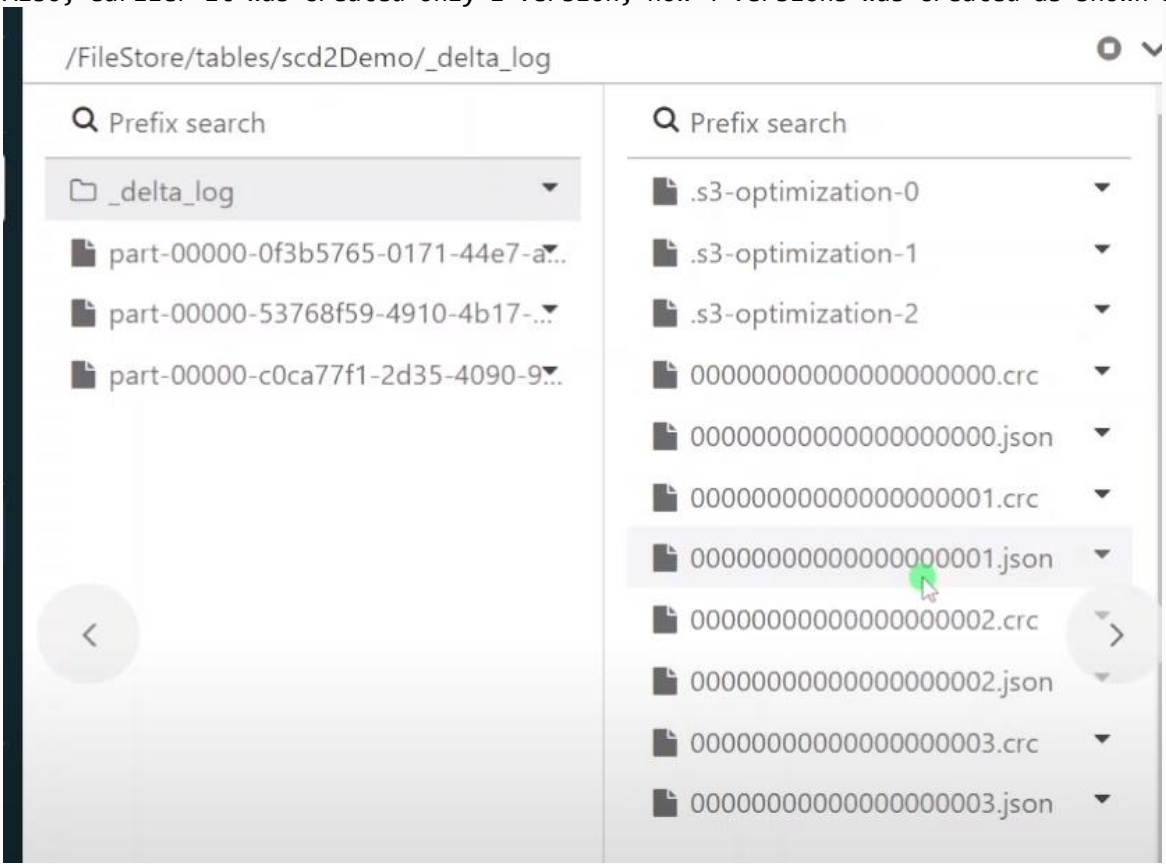
	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2022-06-01T07:18:19.537+0000	9999-12-31T00:0
2	333	Unit3	300	900	250	650	Y	2022-06-01T07:18:26.704+0000	9999-12-31T00:0
3	222	Unit2	900	null	700	100	Y	2022-06-01T07:18:22.952+0000	9999-12-31T00:0

Showing all 3 rows.

After inserting 3 rows in a table, 3 part files got created in path as shown below. Each file is containing one record of this table.



Also, earlier it was created only 1 version, now 4 versions was created as shown below,



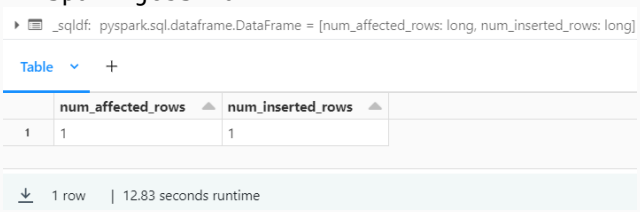
When we are querying using select statement, what happens is spark engine delta engine goes to latest json,this is having pointer to all these tables then it will give snapshot latest snapshot of delta table.

--

Inserting 3 more records

```
%sql
insert into scd2Demo
values(666,'Unit6',200,500,800,400,'Y',current_timestamp(),'9999-12-31');
insert into scd2Demo
values(777,'Unit7',900,Null,700,100,'Y',current_timestamp(),'9999-12-31');
insert into scd2Demo
values(888,'Unit8',300,900,250,650,'Y',current_timestamp(),'9999-12-31');
```

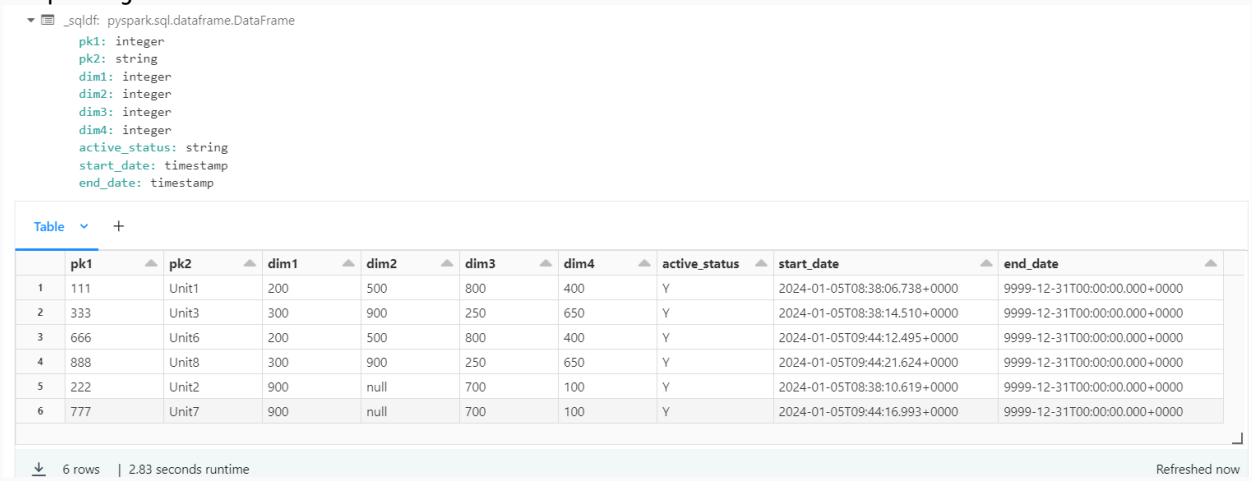
12 spark jobs ran



--

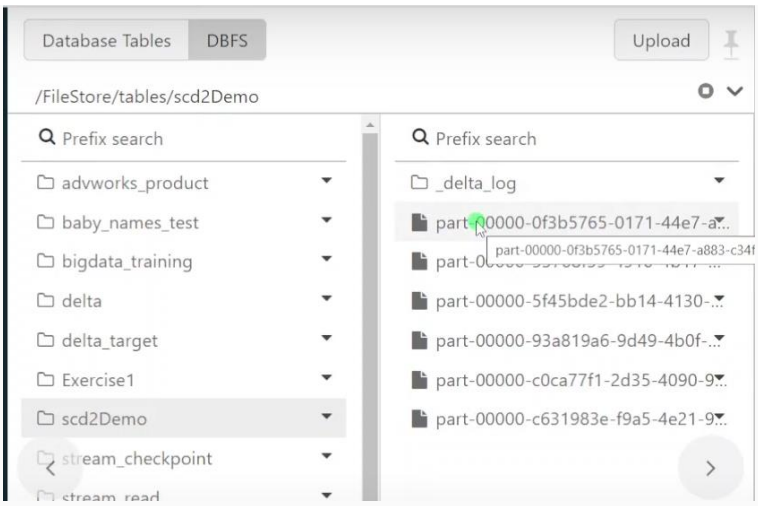
```
%sql
select * from scd2demo
```

3 spark jobs ran.

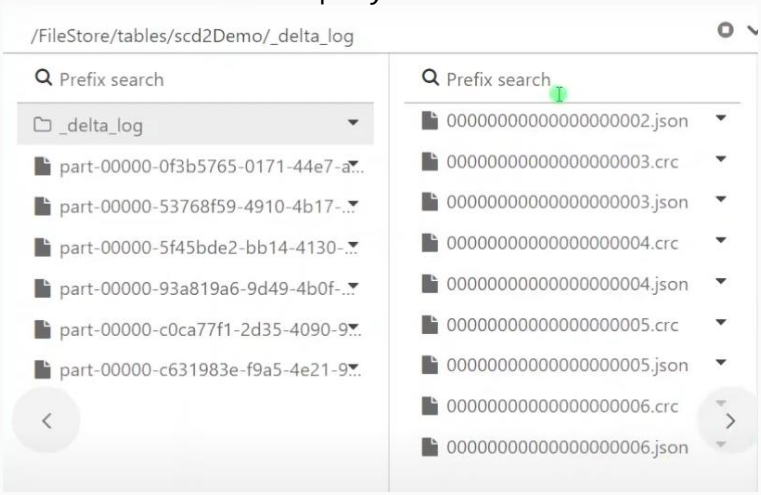


--

There are 6 data files got created, each file is containing 1 record.



If we go to _delta_log ,we can see 7 versions starting from 0 till 6 and 6 is the latest json log file which is having reference of all these files. When we are querying the table, so this particular json log will be used and based on that it will return the result in select query.



--

```
%sql
delete from scd2Demo where pk1 = 777
```

6 spark jobs

```
%sql
select * from scd2demo
```

2 spark jobs

_sqldf: pyspark.sql.dataframe.DataFrame
pk1: integer
pk2: string
dim1: integer
dim2: integer
dim3: integer
dim4: integer
active_status: string
start_date: timestamp
end_date: timestamp

Table +

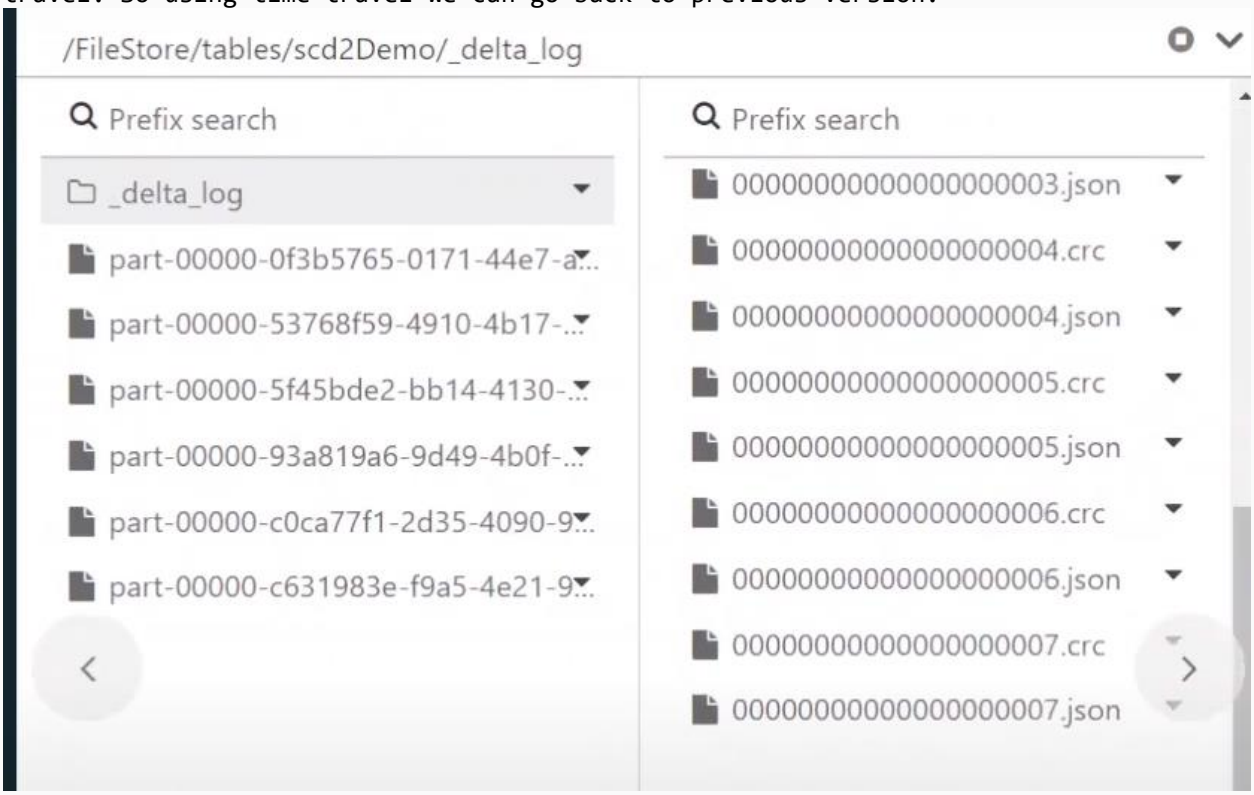
	pk1	pk2	dim1	dim2	dim3	dim4	active_status	start_date	end_date
1	111	Unit1	200	500	800	400	Y	2024-01-05T08:38:06.738+0000	9999-12-31T00:00:00.000+0000
2	333	Unit3	300	900	250	650	Y	2024-01-05T08:38:14.510+0000	9999-12-31T00:00:00.000+0000
3	666	Unit6	200	500	800	400	Y	2024-01-05T09:44:12.495+0000	9999-12-31T00:00:00.000+0000
4	888	Unit8	300	900	250	650	Y	2024-01-05T09:44:21.624+0000	9999-12-31T00:00:00.000+0000
5	222	Unit2	900	null	700	100	Y	2024-01-05T08:38:10.619+0000	9999-12-31T00:00:00.000+0000

5 rows | 2.79 seconds runtime Refreshed now

--

Still, we are able to see 6 part data files, we have deleted 1 record which means corresponding file, file which is containing that particular file should be eliminated but still we could see part file as shown below,

For delete operation it has added new log entry but at the same time it has not removed data file because in delta whenever we are updating or deleting , the older version of the file will not be removed immediately because in delta lake , we can perform time travel. So using time travel we can go back to previous version.



--

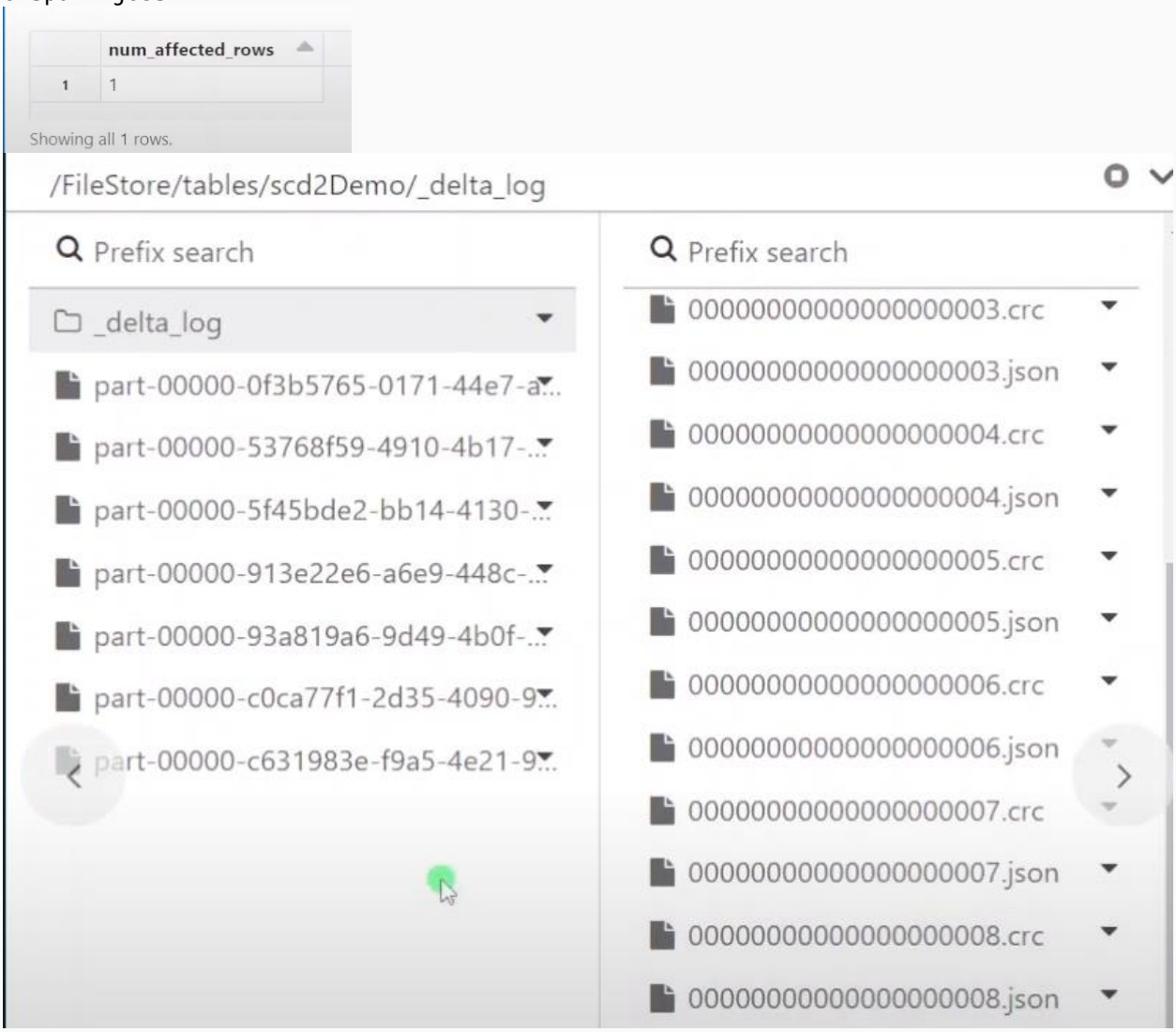
If we want to go to previous version using time travel, in previous version it should give deleted record 777 also, how it will come is ? because it will take from available data only. So that's the reason whenever in delta we are performing update/delete ,it

is not going to remove that data file but instead of that in log file, it is going to give information saying that we have deleted the record so this particular one of the file which is containing that pk1 =777 it is going to tell this particular file is not needed any more so this is not considered for the latest snapshot of the delta table so don't consider that and consider only remaining five , that is what it will write into json log file. Whenever we are exploring delta table then delta engine will come to json log and it will retrieve data only from the active five files . That is the reason we are able to see five records .

--

```
%sql
update scd2Demo set dim1=100 where pk1=666
```

6 spark jobs



One more part file got added as shown above. And one more json log file also got created as shown above.In latest json file, it will tell out of these 7 files one is deleted don't consider that and another one is older version of updated record so don't

consider that file as well, it will tell to consider only 5 active files. When we are exploring we are selecting data from delta table so engine will retrieve data only from active 5 files. If we have to perform time travel then engine will use updated or deleted files as well.

Now coming to optimize command, if we use in above screenshot 7 part files got created. Out of that 5 files are active. These 5 files actually they are holding only 1 record , it could be few KB not even 1 MB but in real time use case, it can be few MB because each and every file might contain one or more than 1 records. So, depending on number of records, number of columns, it might be in few KB's but that is not optimal. For e.g., here we have created 5 different files for latest delta snapshot but all are very smaller. So, in real time scenario, when we are going to create millions or billions of smaller files and as a result delta engine has to maintain huge amount of metadata also. As a result, keeping huge amount of metadata is a process overhead. As a result, it is going to hit the performance.so its not good idea to keep many number of smaller files. Instead of that we can combine smaller files to optimal size so for that we can go with optimize command in delta lake. So as per optimize default size is 1 GB. So, if any of file is having lesser than 1 GB then it will combine, merge the files to 1GB. That is the concept of optimize. In this case we are having few files i.e.; in the latest snapshot , we are having only five files each will be few KB in it, it is not optimal size.so when we apply optimize command on this delta table, then it will combine all those 5 files and it will make several KB or few Mega Bytes. So even though it is not 1 GB,but it is still going to combine all those files into one single file. so that is the concept of optimize.

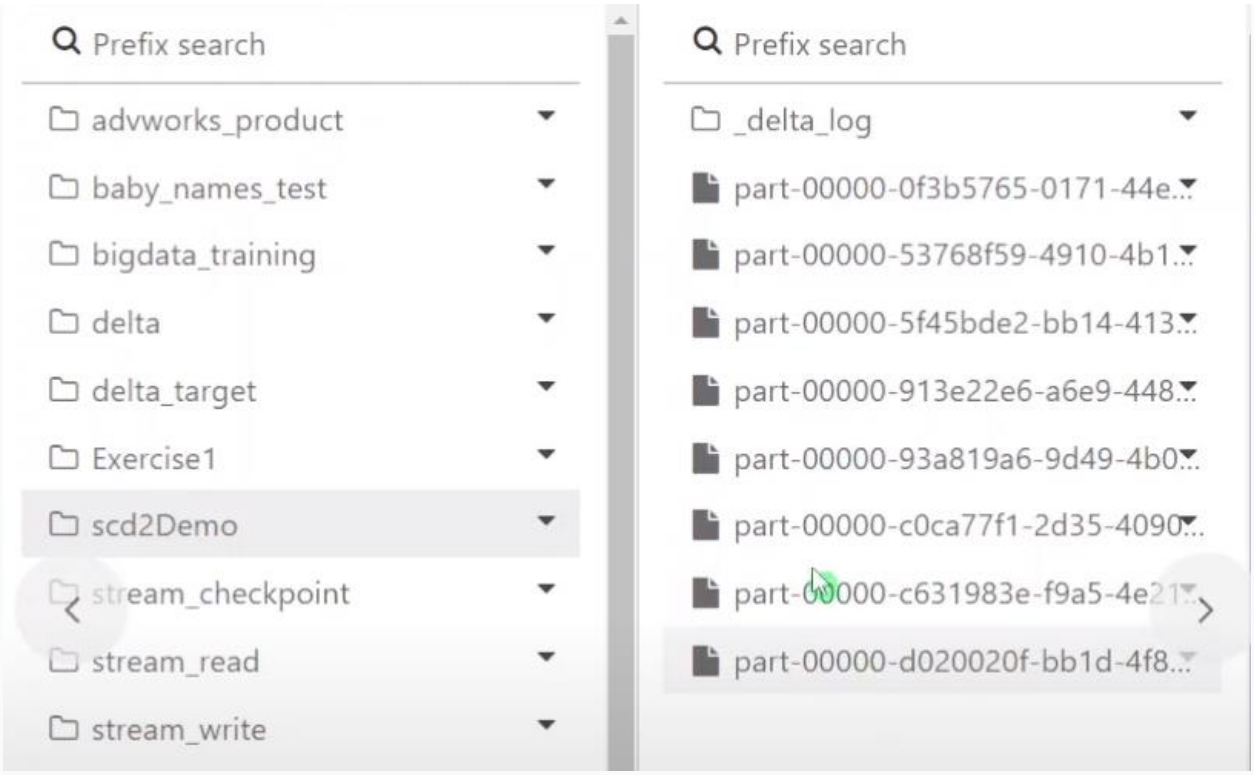
```
%sql
OPTIMIZE scd2Demo
```

9 spark jobs ran.

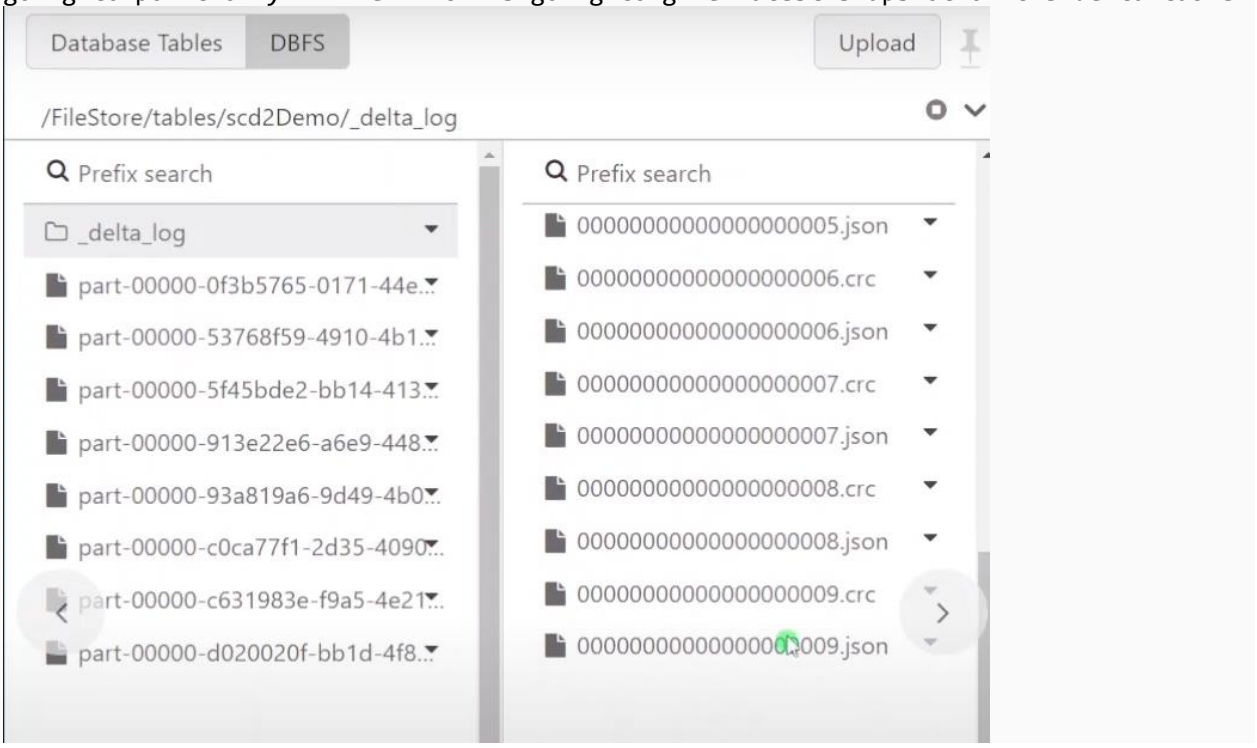
	path	metrics
1	dbfs:/FileStore/tables/scd2Demo	{"numFilesAdded": 1, "numFilesRemoved": 5, "filesAdded": {"min": 2706, "max": 2706, "avg": 2706, "totalFiles": 1, "totalSize": 2706}, "filesRemoved": {"min": 2489, "max": 2525, "avg": 2517.6, "totalFiles": 5, "totalSize": 12588}, "partitionsOptimized": 0, "zOrderStats": null, "numBatches": 1, "totalConsideredFiles": 5, "totalFilesSkipped": 0, "preserveInsertionOrder": true}

Showing all 1 rows.

In above screenshot, we can say that 5 files removed and inserted into single file, removed means actually it is not removed still folder files are available because it is used for time travel so it is not removed immediately. It will be removed later. So, it is not removed actually but instead of that it is combined and it has created new file. so going forward delta engine will refer only the latest combined one single file. so, it is going to improve the performance.



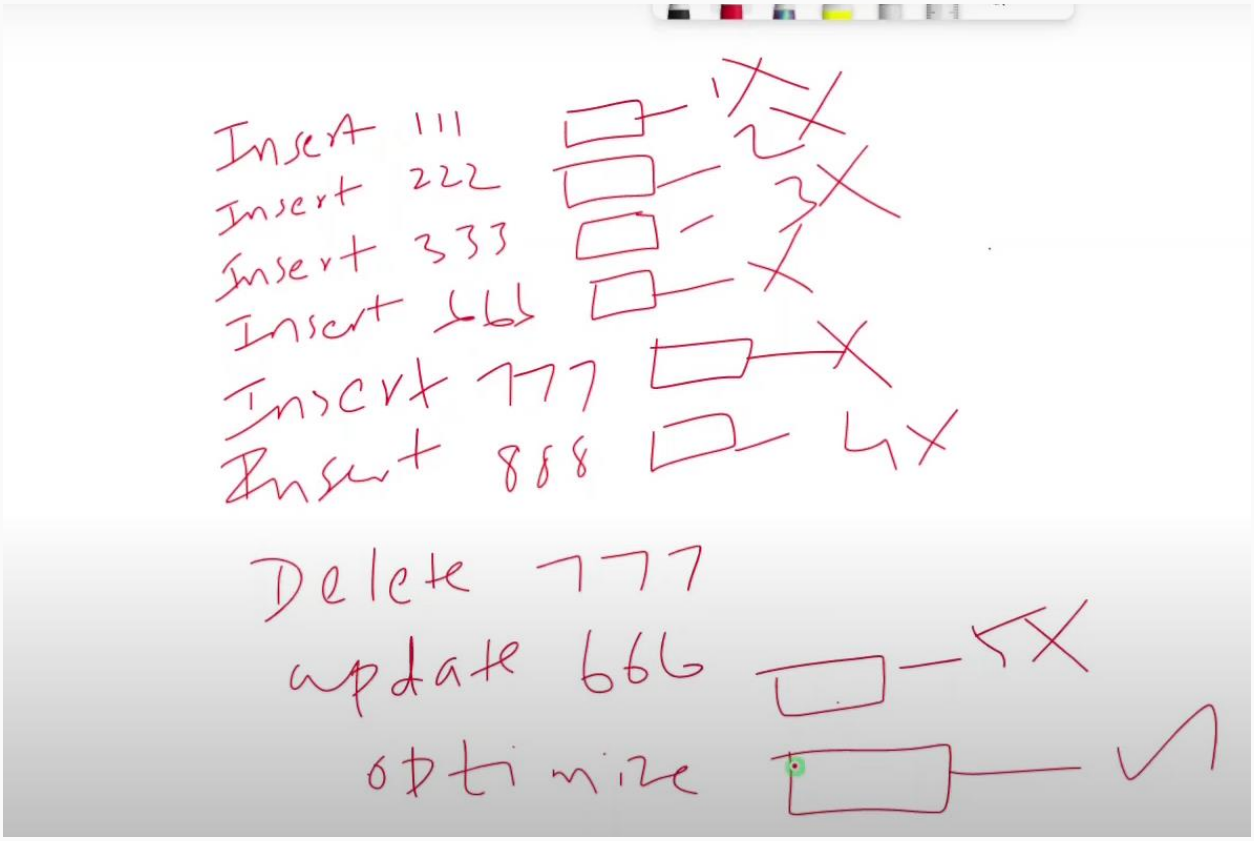
If we see above screenshot, earlier we have only 7 files, now we have 8 files because active 5 files got combined and it has created new file by combining all those 5 files. Coming to log files, it has created one more extra entry of json log file which is going to point only one single file now by removing those 5 files from the log. It is going to point only 1 file which is going to give latest snapshot of the delta table



DELTA LAKE TABLE -VACCUM COMMAND

While performing optimize command or delete or update command, delta lake engine will not remove older files. Let's say we are deleting data from one file which means that file is not needed anymore for delta table but at the same time it is not going to be removed immediately. Instead of that if the new version of data file will be created in the same location but at the same time it is going to maintain the old version of file. Also using this older version of the files, we can perform time travel in delta lake but at the same time over the time over years , we are not going to clean up the data its going to pile up huge amount of data so that is not good from maintenance perspective and storage perspective.so we have to clean up the data periodically so for that we can go with vacuum command.

Vacuum command is used to clean up absolute files which are not part of latest version of delta lake.



Now only 1 active file will be there after writing optimize command. whenever we are performing select * from delta table ,it's going to refer only optimized file will be referred.

Now from above screenshot we can see that we have produced invalid files shown above. In real time scenario its going to produce thousands/millions of invalid files so we need to clean up those files periodically so for that we can go with vacuum command.

```
#LIST OF FILES TO VIEW AT THIS LOCATION
%fs
ls /FileStore/tables/scd2Demo
```

	path	name
1	dbfs:/FileStore/tables/scd2Demo/_delta_log/	_delta_log/
2	dbfs:/FileStore/tables/scd2Demo/part-00000-0f3b5765-0171-44e7-a883-c34f77bbaeea-c000.snappy.parquet	part-00000-0f3b5765-0171-44e7-a883-c34f77bbaeea-c0
3	dbfs:/FileStore/tables/scd2Demo/part-00000-53768f59-4910-4b17-87d2-4789ae6360a7-c000.snappy.parquet	part-00000-53768f59-4910-4b17-87d2-4789ae6360a7-c
4	dbfs:/FileStore/tables/scd2Demo/part-00000-5f45bde2-bb14-4130-8a92-57ba4d6b4ab6-c000.snappy.parquet	part-00000-5f45bde2-bb14-4130-8a92-57ba4d6b4ab6-c
5	dbfs:/FileStore/tables/scd2Demo/part-00000-913e22e6-a6e9-448c-8a84-ccff2ab44f2e-c000.snappy.parquet	part-00000-913e22e6-a6e9-448c-8a84-ccff2ab44f2e-c0
6	dbfs:/FileStore/tables/scd2Demo/part-00000-93a819a6-9d49-4b0f-b5ae-2345c00b3dca-c000.snappy.parquet	part-00000-93a819a6-9d49-4b0f-b5ae-2345c00b3dca-c
7	dbfs:/FileStore/tables/scd2Demo/part-00000-c0ca77f1-2d35-4090-9386-3fc31dde8af7-c000.snappy.parquet	part-00000-c0ca77f1-2d35-4090-9386-3fc31dde8af7-c0

Showing all 9 rows.

--

For vacuum command, default retain duration is 7 days which means 168 hrs. If file is invalidated 7 days before then only those files will be considered.so in order to clean up any data, data file from delta folder there are 2 conditions, 1 is that particular data file should not be part of latest version of delta table. 2nd is particular file should have been invalidated atleast 7 days before by default. This 7 days value is configurable. We can change it.

```
VACCUM COMMAND VERSION 1:
%sql
VACUUM scd2Demo DRY RUN
```

In the first version, using dry run which means it is not going to clean up the file immediately instead of that it is going to list down the files which will be deleted as part of vacuum operation. For that purpose, we can use dry run.

10 spark jobs ran

	path
1	dbfs:/FileStore/tables/scd2Demo/part-00000-0f3b5765-0171-44e7-a883-c34f77bbaeea-c000.snappy.parquet
2	dbfs:/FileStore/tables/scd2Demo/part-00000-53768f59-4910-4b17-87d2-4789ae6360a7-c000.snappy.parquet
3	dbfs:/FileStore/tables/scd2Demo/part-00000-5f45bde2-bb14-4130-8a92-57ba4d6b4ab6-c000.snappy.parquet
4	dbfs:/FileStore/tables/scd2Demo/part-00000-913e22e6-a6e9-448c-8a84-ccff2ab44f2e-c000.snappy.parquet
5	dbfs:/FileStore/tables/scd2Demo/part-00000-93a819a6-9d49-4b0f-b5ae-2345c00b3dca-c000.snappy.parquet
6	dbfs:/FileStore/tables/scd2Demo/part-00000-c0ca77f1-2d35-4090-9386-3fc31dde8af7-c000.snappy.parquet
7	dbfs:/FileStore/tables/scd2Demo/part-00000-c631983e-f9a5-4e21-9554-e507c7402c30-c000.snappy.parquet

Showing all 7 rows.

In above screenshot we are having 7 rows which means 7 files will be deleted.

--

Latest version of delta table is referring only 1 optimized file which means 7 files are invalid so it will remove 7 files and those 7 files actually it has been validated before 7 days because we run optimize command 10 days back.

Or else we can execute vacuum command directly without dry run,

```
%sql
VACUUM scd2Demo
```

```
%sql
```

```
VACUUM scd2Demo RETAIN 720 HOURS DRY RUN
```
8 spark jobs ran
Query returned no results.

Using above command, we can change parameter, by default it will be 168 hrs but this is highly configurable.
For example, we have given condition of 30 days(720 hrs) which means we are going to clean up only files which are not part of latest version.
--we can also give 0 hrs in command which means starting from this minute if we have any invalid or absolute file that should be removed. But by vacuum command, it won't give 0 hrs because in prod env its not a good practice.
--
```
%sql
set spark.databricks.delta.retentionDurationCheck.enabled = False
```
If we enable this , we can run vacuum command even with 0 hrs.
```
%sql
VACUUM scd2Demo RETAIN 0 HOURS DRY RUN
```
10 spark jobs ran

	path
1	dbfs:/FileStore/tables/scd2Demo/part-00000-0f3b5765-0171-44e7-a883-c34f77bbaeea-c000.snappy.parquet
2	dbfs:/FileStore/tables/scd2Demo/part-00000-53768f59-4910-4b17-87d2-4789ae6360a7-c000.snappy.parquet
3	dbfs:/FileStore/tables/scd2Demo/part-00000-5f45bde2-bb14-4130-8a92-57ba4d6b4ab6-c000.snappy.parquet
4	dbfs:/FileStore/tables/scd2Demo/part-00000-913e22e6-a6e9-448c-8a84-ccff2ab44f2e-c000.snappy.parquet
5	dbfs:/FileStore/tables/scd2Demo/part-00000-93a819a6-9d49-4b0f-b5ae-2345c00b3dca-c000.snappy.parquet
6	dbfs:/FileStore/tables/scd2Demo/part-00000-c0ca77f1-2d35-4090-9386-3fc31dde8af7-c000.snappy.parquet
7	dbfs:/FileStore/tables/scd2Demo/part-00000-c631983e-f9a5-4e21-9554-e507c7402c30-c000.snappy.parquet

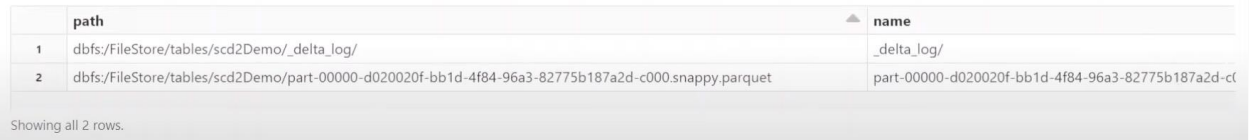
Showing all 7 rows.

--
```
%sql
VACCUM scd2Demo
```
This command will physically clean up the data so for that no need to use any other parameters simply we can give VACCUM scd2Demo. It is going to remove any absolute file which are older than 7 days.
5 spark jobs ran

	path
1	dbfs:/FileStore/tables/scd2Demo

Showing all 1 rows.

--
```
%fs
ls /FileStore/tables/scd2Demo
```



	path	name
1	dbfs:/FileStore/tables/scd2Demo/_delta_log/	_delta_log/
2	dbfs:/FileStore/tables/scd2Demo/part-00000-d020020f-bb1d-4f84-96a3-82775b187a2d-c000.snappy.parquet	part-00000-d020020f-bb1d-4f84-96a3-82775b187a2d-c(

Showing all 2 rows.

Which means delta log and optimized parquet file only present. We have cleaned up previously invalidated files.

DELTA TABLE : Z-ORDERING

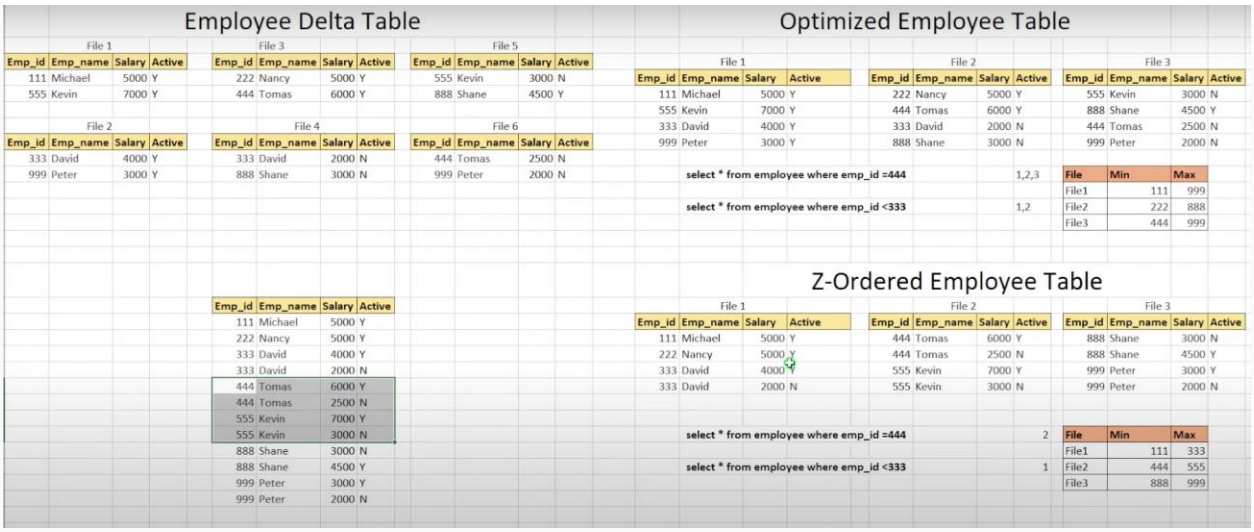
Z-ordering is one of the performance optimization technique used in delta lake development.

WHAT IS DELTA LAKE ?Delta Lake is one additional layer sitting on top of data lake which provides features like ACID transactions. Actually, data is stored in the form of files and it does not allow as a transaction. It was one of the shortcomings with data lake. So as a result , data bricks introduced delta lake which provides additional layer sitting on top of the data lake which provides many features such as performance improvement and one of the important feature is ACID transaction.

WHAT IS DELTA TABLE ? Delta Table is combination of data files in the form of parquet plus transaction log files in the form of JSON and Parquet. Delta Table would contain one specific name and stored in a specific location. So, this combination is called delta table. It is similar to any relational database table but coming to data structure now this is a file format and we can perform ACID transaction directly on the file using some mechanism that is provided by delta lake.

WHAT IS OPTIMIZE ? In order to improve performance of delta table, we can use optimize command. While working with delta table over time, we used to create large number of small datafiles. Whenever we are creating large number of small files that is not good for performance because engine has to maintain huge amount of metadata as well and as a result it will impact lot of performance optimization techniques such as data skipping. In order to avoid that we can go with optimize, so smaller files will be combined into optimal size. The default size for optimize command is 1 GB. Let’s say we are having 100 mega bytes of files then 10 files will be combined into one single file using optimize command.

WHAT IS Z-ORDERING? Z-ordering is an extension of optimize. Z-ordering is used along with optimize. Optimize command is used to combine many small files into larger one but at the same time it does not care about data ordering. It would randomly combine the file data and it will create optimal size of the file but at the same time if we are going to add Z-Ordering also along with optimize then it will combine smaller files into larger one but at the same time it will also reorder the data which will be helpful in performance improvement.



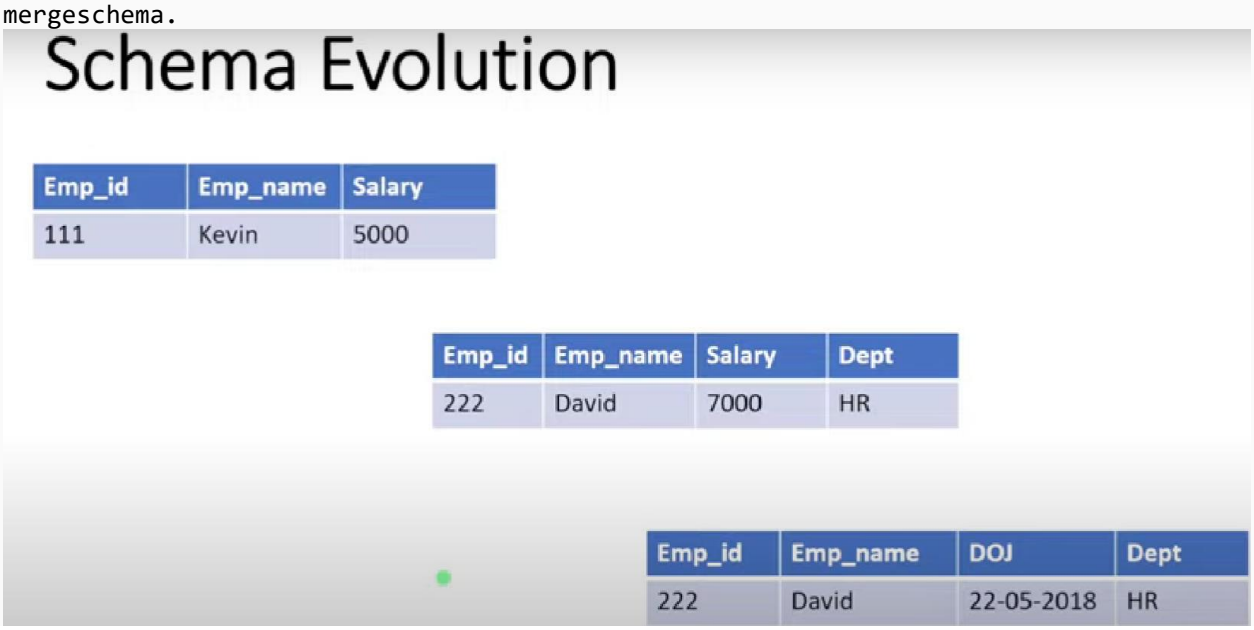

```
%sql
OPTIMIZE scd2Demo
ZORDER PK1
```

	path	metrics
1	dbfs:/FileStore/tables/scd2Demo	{"numFilesAdded": 1, "numFilesRemoved": 5, "filesAdded": {"min": 2706, "max": 2706, "avg": 2706, "totalFiles": 1, "totalSize": 2706}, "filesRemoved": {"min": 2489, "max": 2525, "avg": 2517.6, "totalFiles": 5, "totalSize": 12588}, "partitionsOptimized": 0, "zOrderStats": null, "numBatches": 1, "totalConsideredFiles": 5, "totalFilesSkipped": 0, "preserveInsertionOrder": true}

Showing all 1 rows.

DELTA TABLE : SCHEMA EVOLUTION : MergeSchema

When we are working with bigdata files, structure of the file keeps on changing. If we don't put proper mechanism to handle schema mismatch, the entire pipeline would be failed. In order to handle this situation, data bricks has provided feature like mergeschema.



Let's say we have developed a pipeline which is handling data processing for structure like above screenshot. We keep on receiving csv files and file contains data with structure similar to above , Emp_id, Emp_name, Salary. And for this data file we have constructed delta table with same schema structure but after certain time we started receiving different structure for the data file. So, it's like ,for example, we started receiving department column as additional column. So earlier we built delta table only for this structure as below . now we started receiving different column as well department. As a result, entire pipeline will fail because this new record cannot be inserted into delta table because it cannot accommodate new column department and not only this time, may be later we used to get some other schema as well. For example, after some days we are getting some other csv file which is having date of joining in the place of salary. So, this is also new column which is not available with our original delta table. We are receiving data file with change of schema. As a result, there would be schema mismatch so entire pipeline will be failed. So, for that we have to put proper mechanism.

```
from delta.tables import *
DeltaTable.create(spark) \
    .tableName("employee_demo") \
    .addColumn("emp_id","INT") \
    .addColumn("emp_name","STRING") \
    .addColumn("gender","STRING") \
    .addColumn("salary","INT") \
    .addColumn("Dept","STRING") \
    .property("description","table created for demo purpose") \
    .location("/FileStore/tables/delta/path_employee_demo") \
    .execute()
```

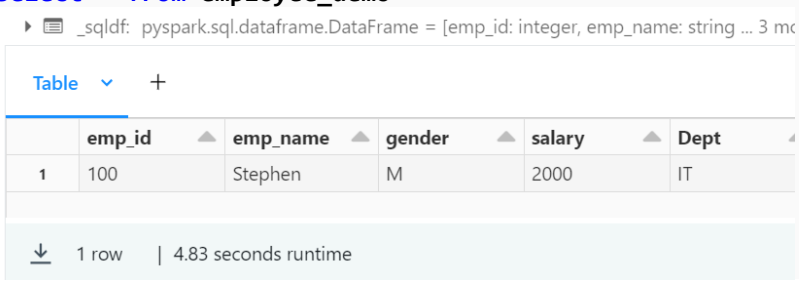
4 spark jobs ran

```
%sql
select * from employee_demo
```

Query Returned no results

```
%sql
insert into employee_demo values(100,"Stephen","M",2000,"IT");
```

```
%sql
select * from employee_demo
```



DATA → click on table name → we will get schema details of the table(its column and its data types) and history of table operations(its versions and operations performed on table whether CREATE TABLE/WRITE/UPDATE/DELETE/MERGE)

SCHEMA EVOLUTION

Now let us assume we are receiving csv file which is having extra column.

```
from pyspark.sql.types import IntegerType,StringType
employee_data = [(200,"Philipp","M",8000,"HR","test data")]

employee_schema = StructType([ \
    StructField("emp_id",IntegerType(),False) , \
    StructField("emp_name",StringType(),False) , \
    StructField("gender",StringType(),True) , \
    StructField("salary",IntegerType(),True) , \
    StructField("dept",StringType(),True) , \
    StructField("additional_column1",StringType(),True) , \
    ])
df = spark.createDataFrame(employee_data,employee_schema)
display(df)
```

If we want to insert this data frame into my delta table it is going to fail because our delta table schema does not contain this extra column as shown below.

```
df.write.format("delta").mode("append").saveAsTable("employee_demo")
```

If we run above command schema mismatch detected as shown in below screenshot when writing to delta table and table schema is not containing any additional column but coming to data schema its having one additional column. So as a result, it could not handle the scenario so the pipeline will fail with this error message as shown in below screenshot.

```
AnalysisException: A schema mismatch detected when writing to the Delta table (Table ID: 9385c005-28af-4990-82ce-9cbe844075b0).
To enable schema migration using DataFrameWriter or DataStreamWriter, please set:
'.option("mergeSchema", "true")'.
For other operations, set the session configuration
spark.databricks.delta.schema.autoMerge.enabled to "true". See the documentation
specific to the operation for details.

Table schema:
root
-- emp_id: integer (nullable = true)
-- emp_name: string (nullable = true)
-- gender: string (nullable = true)
-- salary: integer (nullable = true)
-- Dept: string (nullable = true)


Data schema:
root
-- emp_id: integer (nullable = true)
-- emp_name: string (nullable = true)
-- gender: string (nullable = true)
-- salary: integer (nullable = true)
-- dept: string (nullable = true)
-- additional_column1: string (nullable = true)
```

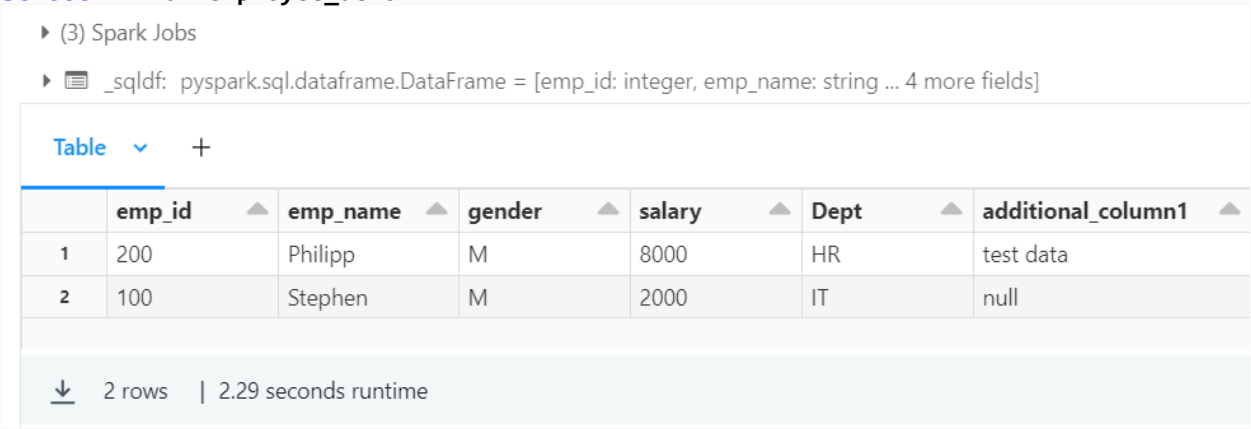
In order to avoid those error, data bricks has provided mechanism **mergeSchema** to handle schema mismatch issue.

```
    df.write.option("mergeSchema","true").format("delta").mode("append").saveAsTable("employee_demo")
```

once we run above command,emp_id=200 row data will be inserted into table.

By default, mergeSchema will be false, now we are enabling mergeSchema to be true.

```
%sql
select * from employee_demo
```



	emp_id	emp_name	gender	salary	Dept	additional_column1
1	200	Philipp	M	8000	HR	test data
2	100	Stephen	M	2000	IT	null

additional_column1 and its data was added to the table as shown above.

For the latest record it has populated value that was given in the data frame. But at the same time, previously inserted record will populate just null value because we don't have any corresponding respective value for that column.

Now if we look at schema in hive metastore,

default.employee_demo | Refresh

My Cluster

Details History

Description:
Created at: 2024-01-17 07:20:58
Last modified: 2024-01-18 05:41:34
Partition columns:
Number of files: 2
Size: 3.35 kB

Schema:

	col_name	data_type	comment
1	emp_id	int	null
2	emp_name	string	null
3	gender	string	null
4	salary	int	null
5	Dept	string	null
6	additional_column1	string	null

Sample Data:

	emp_id	emp_name	gender	salary	Dept	additional_column1
1	200	Philipp	M	8000	HR	test data
2	100	Stephen	M	2000	IT	null

```
--
Now creating another column as shown below.
from pyspark.sql.types import IntegerType,StringType
employee_data = [(300,"David","M",8000,"HR","dummy data")]

employee_schema = StructType([ \
    StructField("emp_id",IntegerType(),False) , \
    StructField("emp_name",StringType(),False) , \
    StructField("gender",StringType(),True) , \
    StructField("salary",IntegerType(),True) , \
    StructField("dept",StringType(),True) , \
    StructField("additional_column2",StringType(),True) , \
    ])
df = spark.createDataFrame(employee_data,employee_schema)
display(df)
```

(3) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [emp_id: integer, emp_name: string ... 4 more fields]

Table +

	emp_id	emp_name	gender	salary	dept	additional_column2
1	300	David	M	8000	HR	dummy data

1 row | 0.90 seconds runtime

```
df.write.option("mergeSchema","true").format("delta").mode("append").saveAsTable("employee_demo")
%sql
select * from employee_demo
```

(3) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [emp_id: integer, emp_name: string ... 5 more fields]

Table +

	emp_id	emp_name	gender	salary	Dept	additional_column1	additional_column2
1	300	David	M	8000	HR	null	dummy data
2	200	Philipp	M	8000	HR	test data	null
3	100	Stephen	M	2000	IT	null	null

3 rows | 2.37 seconds runtime

HOW TO INSERT DATAFRAME DATA INTO DELTA TABLE

Sample Data Frame :

```
data = [("ABC pvt ltd","Q1",2000),("XYZ pvt ltd","Q1",5000),("KLM pvt ltd","Q1",2000)]
column= ["Company","Quarter","Revenue"]
df= spark.createDataFrame(data=data, schema=column)
display(df)
```

	Company	Quarter	Revenue
1	ABC pvt ltd	Q1	2000
2	XYZ pvt ltd	Q1	5000
3	KLM pvt ltd	Q1	2000

Now we need to create delta table. In order to create delta table from df, we need to use saveAsTable. We are creating managed table so we are not providing any path.

```
df.write.saveAsTable("default.fact_revenue")
```

Now delta table got created. Now we want to see data from delta table. In order to retrieve data from delta table, there are multiple approaches.

```
display(spark.table("default.fact_revenue"))
```

	Company	Quarter	Revenue
1	XYZ pvt ltd	Q1	5000
2	KLM pvt ltd	Q1	2000
3	ABC pvt ltd	Q1	2000

2nd approach:

```
display(spark.sql("select * from default.fact_revenue"))
```

3rd approach

```
%sql
select * from default.fact_revenue
```

Now we need to insert data into delta table based on data frame data. We can do it via below approach.

```
data = [("RST pvt ltd","Q4",7000)]
column =["Company","Quarter","Revenue"]
df1= spark.createDataFrame(data=data,schema=column)
display(df1)
```

	Company	Quarter	Revenue
1	RST pvt ltd	Q4	7000

We need to insert above 1 record into delta table. While inserting data into delta table, there are multiple approaches as shown below.

```
df1.write.insertInto("default.fact_revenue",overwrite=False)
```

overwrite as false means this df1 data will not be overwritten. overwrite as true means it will remove 3 existing records and then top of that it will insert this new record but when we give overwrite = false which means it will append the data .

```
display(spark.table("default.fact_revenue"))
```

	Company	Quarter	Revenue
1	XYZ pvt ltd	Q1	5000
2	KLM pvt ltd	Q1	2000
3	RST pvt ltd	Q4	7000
4	ABC pvt ltd	Q1	2000

```
data = [("QRT pvt ltd","Q3",3000)]
column =["Company","Quarter","Revenue"]
df2= spark.createDataFrame(data=data,schema=column)
display(df2)
```

	Company	Quarter	Revenue
1	QRT pvt ltd	Q3	3000

```
df2.write.insertInto("default.fact_revenue",overwrite=True)
display(spark.table("default.fact_revenue"))
```

	Company	Quarter	Revenue
1	QRT pvt ltd	Q3	3000

We can use %sql magic command , for that we have to convert data frame into table or view to insert data into delta table from sql .

```
data = [("ABC pvt ltd","Q3",4000)]
column =["Company","Quarter","Revenue"]
df3= spark.createDataFrame(data=data,schema=column)
display(df3)
```

	Company	Quarter	Revenue
1	ABC pvt ltd	Q3	4000

```
df3.createOrReplaceTempView("v_insert_data")
%sql
insert into default.fact_revenue
select * from v_insert_data
display(spark.table("default.fact_revenue"))
```

	Company	Quarter	Revenue
1	QRT pvt ltd	Q3	3000
2	ABC pvt ltd	Q3	4000

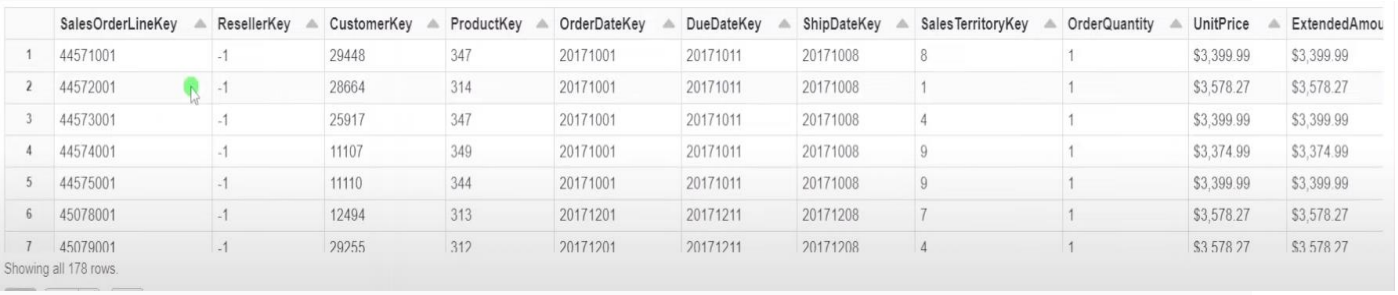
SPARK_PARTITION_ID | DATA SKEWNESS

SPARK_PARTITION_ID function is used to identify partition_id for each record in the data frame but what is the use of knowing partition_id for each record ? SPARK_PARTITION_ID function is used to identify the data skewness in the dataframe. Data Skewness will keep the performance in spark programming. Generally, in order to identify data skewness, normally we will go with spark UI. With in spark UI, we can check no of tasks and how much time each task has taken to complete the execution. based on that , if any particular task takes more time compared to other tasks, then we can conclude there is a data skewness for that particular task or partition. The other method is we can look at 25^{th} percentile, 50^{th} percentile or 75^{th} percentile for the matrix. We can check execution time at these different intervals to identify the data skewness. Once we identify data skewness we can go with mechanisms such as salting

/repartition/bucketby etc. There are many methods to handle data skewness, we can go with one of the method.

Code:
Creating DataFrame:

```
from pyspark.sql import functions as F
root_path = "/FileStore/tables/Exercise1/"
df =spark.read.option("sep","|").option("header","true").csv(root_path + "/*/*/*")
display(df)
```

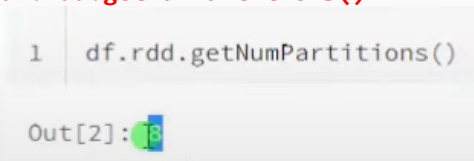


	SalesOrderLineKey	ResellerKey	CustomerKey	ProductKey	OrderDateKey	DueDateKey	ShipDateKey	SalesTerritoryKey	OrderQuantity	UnitPrice	ExtendedAmou
1	44571001	-1	29448	347	20171001	20171011	20171008	8	1	$3,399.99	$3,399.99
2	44572001	-1	28664	314	20171001	20171011	20171008	1	1	$3,578.27	$3,578.27
3	44573001	-1	25917	347	20171001	20171011	20171008	4	1	$3,399.99	$3,399.99
4	44574001	-1	11107	349	20171001	20171011	20171008	9	1	$3,374.99	$3,374.99
5	44575001	-1	11110	344	20171001	20171011	20171008	9	1	$3,399.99	$3,399.99
6	45078001	-1	12494	313	20171201	20171211	20171208	7	1	$3,578.27	$3,578.27
7	45079001	-1	29255	312	20171201	20171211	20171208	4	1	$3,578.27	$3,578.27

Showing all 178 rows.

Normally default parallelism for spark is 8 which means when it reads data from external csv file it would create 8 partitions by default.
So, if we want to get no of partitions for our data frame , we can go with below syntax.

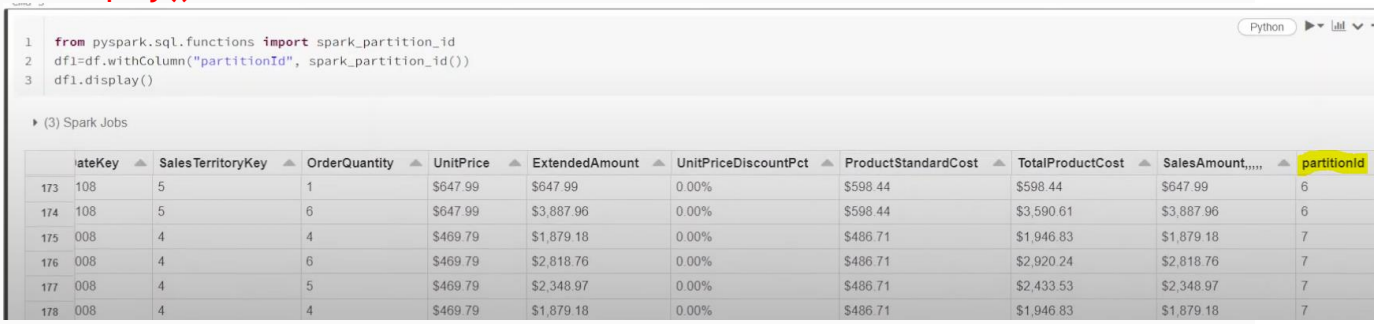
```
df.rdd.getNumPartitions()
```



```
1  df.rdd.getNumPartitions()

Out[2]: 8
```

We can see here 8 partitions are created and partition id will range from 0 to 7. Now we want to see partition id for each record in the data frame. For that we will use spark_partition_id function

```
from pyspark.sql.functions import spark_partition_id
df1 =df.withColumn("partitionId",spark_partition_id())
df1.display()
```



```
1  from pyspark.sql.functions import spark_partition_id
2  df1=df.withColumn("partitionId", spark_partition_id())
3  df1.display()
```

Python

(3) Spark Jobs

	ateKey	SalesTerritoryKey	OrderQuantity	UnitPrice	ExtendedAmount	UnitPriceDiscountPct	ProductStandardCost	TotalProductCost	SalesAmount,,,,,	partitionId
173	108	5	1	$647.99	$647.99	0.00%	$598.44	$598.44	$647.99	6
174	108	5	6	$647.99	$3,887.96	0.00%	$598.44	$3,590.61	$3,887.96	6
175	008	4	4	$469.79	$1,879.18	0.00%	$486.71	$1,946.83	$1,879.18	7
176	008	4	6	$469.79	$2,818.76	0.00%	$486.71	$2,920.24	$2,818.76	7
177	008	4	5	$469.79	$2,348.97	0.00%	$486.71	$2,433.53	$2,348.97	7
178	008	4	4	$469.79	$1,879.18	0.00%	$486.71	$1,946.83	$1,879.18	7

Programatically if I want to change no of partitions to 20, using repartition we can do it.

```
df2=df.repartition(20).withColumn("partitionId", spark_partition_id())
df2.display()
```

(4) Spark Jobs

	ateKey	SalesTerritoryKey	OrderQuantity	UnitPrice	ExtendedAmount	UnitPriceDiscountPct	ProductStandardCost	TotalProductCost	SalesAmount,,,,,	partitionId
173	308	10	2	$38.10	$76.20	0.00%	$23.75	$47.50	$76.20	19
174	308	4	1	$202.33	$202.33	0.00%	$187.16	$187.16	$202.33	19
175	708	5	1	$338.99	$338.99	0.00%	$308.22	$308.22	$338.99	19
176	109	6	2	$5.70	$11.40	0.00%	$3.40	$6.79	$11.40	19
177	108	5	6	$647.99	$3,887.96	0.00%	$598.44	$3,590.61	$3,887.96	19
178	008	4	4	$469.79	$1,879.18	0.00%	$486.71	$1,946.83	$1,879.18	19

Showing all 178 rows.

```
df2.rdd.getNumPartitions()
```

op: 20

since we have used repartition function and created 20 partitions .

Now I want to identify dataskewness for the data frame, so for that we have to perform aggregation based on partition id . we have to count no of rows per partition id so that we will get some idea how my data is distributed across partition as shown below,

```
from pyspark.sql.functions import spark_partition_id,asc,desc
df3 =df2\
    .withColumn("partitionId",spark_partition_id())\
    .groupBy("partitionId")\
    .count()\
    .orderBy(desc("count"))

df3.display()
```

Basically we are doing orderBy , so I will get partition where we have data skew , that's the reason we are giving desc order which means partition which is having more no of records will come at the top.

	partitionId	count
1	9	10
2	10	10
3	12	10
4	3	9
5	5	9
6	19	9
7	6	9

Showing all 20 rows.

	partitionId	count
15	17	9
16	1	8
17	14	8
18	16	8
19	2	8
20	0	8

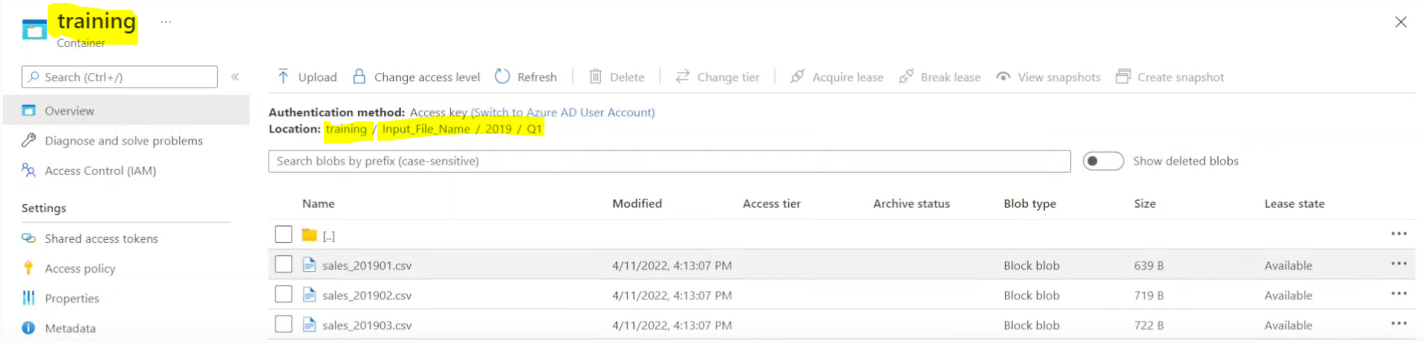
Showing all 20 rows.

Partition id 9 10 12 having 10 records. Similarly it is ranging 10 records per partition to 8 records per partition. So, it is uniformly distributed. So, we don't need to worry about data skewness in this particular data frame because it is evenly distributed.

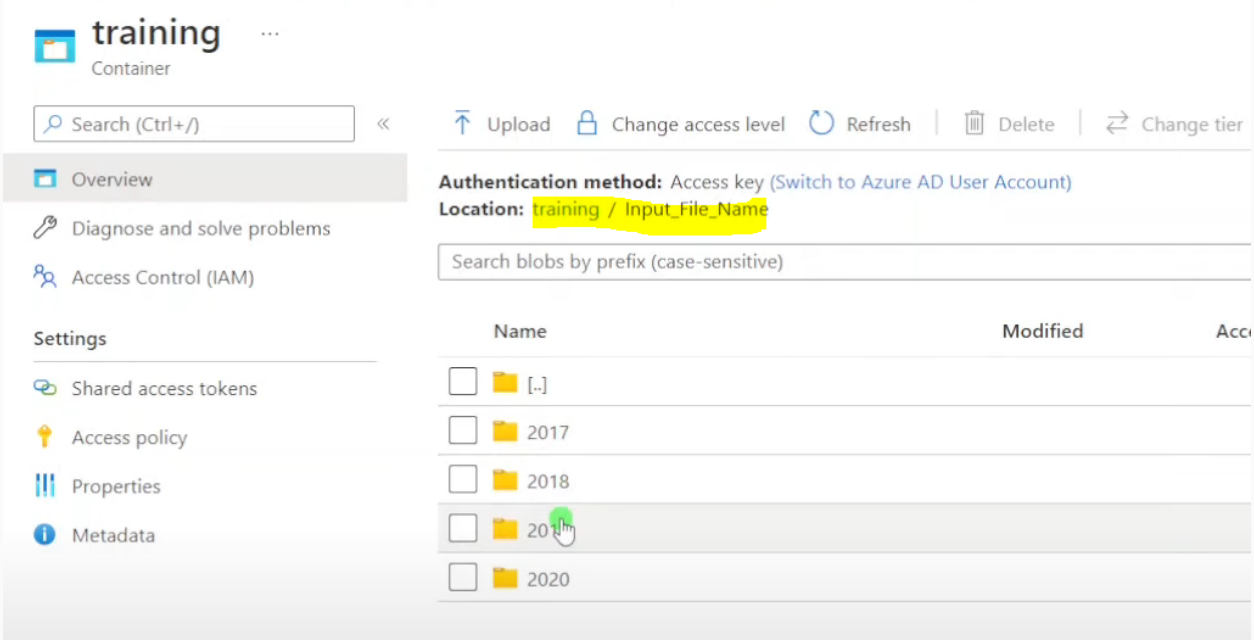
Input_File_Name

This function Input_File_Name is used to identify input file name for each record that got created in the data frame. Let's say we are creating data frame by reading a folder. within a folder, we are having 100's and 1000s of files and later while exploring the data frame we come to know there is a problem with 1 particular record.Now we have to trace it back through which file this particular corrupt record got produced. So manually we cannot go back through several folders and several files, we cant open files manually and check.Instead of that we can use input file name. So, while creating data frame along with actual data , it will capture input file name also. This function is mainly used for debugging and troubleshooting purpose.

Firstly, we created ADLS Storage in azure portal and also we created container named training and Input_File_Name folder and within that, years folder created, inside years quarter folders and its having monthly folders within each quarter as shown below,

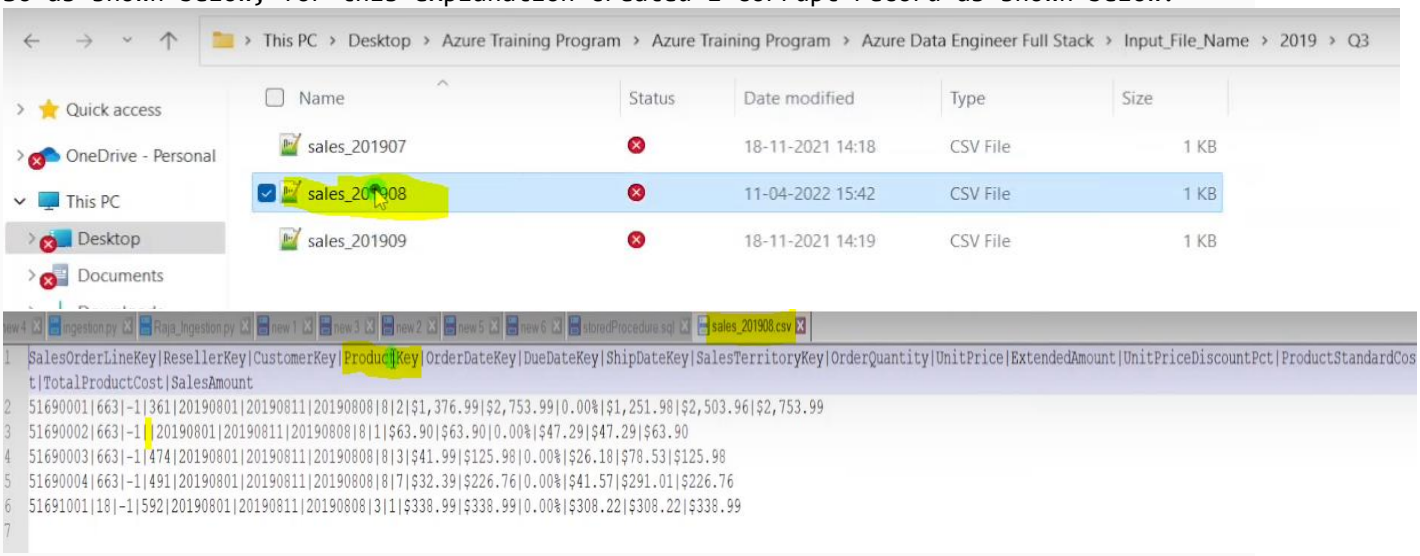


Let's say there are around 36 files located under 16 different folders. Let's say we are creating 1 data frame by reading entire data, entire files under input file name.



In one of the input file there is some problem. Later through sanity check in our program we come to know . But how can we identify that particular file for that record. So manually we can't go through all these files and we cant check.So instead of that we can use Input_File_Name function to determine the file name through which that particular record got created.

So as shown below, for this explanation created 1 corrupt record as shown below.



Here in attached csv file as shown above, there is one column ProductKey located in this file. This is one of the mandatory column. It should not accept any null value but this column ProductKey has null value. This is one of the corrupt record that we have to capture.

So, till now we have created Azure Blob Storage and uploaded files under different folders.
Now we will go to ADB notebook, we will create mount point to read data from ADLS Storage in databricks.

```
dbutils.fs.mount(
      source = "wasbs://training@cloudshell267238421.blob.core.windows.net",
      mount_point = "/mnt/adls",
      extra_configs =
{"fs.azure.account.key.cloudshell267238421.blob.core.windows.net":"fzannoxpglPVhBgRpxzVJBGWr3MtFdJXq3SapArPW2f63e299jCcefEcIB0f7DJslArRtiu9Z3fRgN5mxNU3OQ=="})
```

```
Out[3]: True
```

Now mount point is created for all files located under Input_File_Name. So first we are creating variable with root_path.

root_path = "/mnt/adls/Input_File_Name/"

If we want to see all folders under Input File Name

%fs

ls /mnt/adls/Input_File_Name/

This will list down files located under input file name folder,Its not recursive, Basically it will give only 4 folders as shown below.

	path	name	size	modificationTime
1	dbfs:/mnt/adls/Input_File_Name/2017/	2017/	0	0
2	dbfs:/mnt/adls/Input_File_Name/2018/	2018/	0	0
3	dbfs:/mnt/adls/Input_File_Name/2019/	2019/	0	0
4	dbfs:/mnt/adls/Input_File_Name/2020/	2020/	0	0

Now we want to read all files under root_path folder, so using wild card as /*/*/* in below syntax.

```
from pyspark.sql import functions as F
df= spark.read.option("sep","|").option("header","true").csv(root_path + "/*/*/*")
display(df)
```

so, this containing all columns from all csv files.

	SalesOrderLineKey	ResellerKey	CustomerKey	ProductKey	OrderDateKey	DueDateKey	ShipDateKey
1	44571001	-1	29448	347	20171001	20171011	20171008
2	44572001	-1	28664	314	20171001	20171011	20171008
3	44573001	-1	25917	347	20171001	20171011	20171008
4	44574001	-1	11107	349	20171001	20171011	20171008
5	44575001	-1	11110	344	20171001	20171011	20171008
6	45078001	-1	12494	313	20171201	20171211	20171208
7							

Showing all 178 rows.

Let's say in our project, we are having some sanity check after reading the files from folder . SO, for example let's say we should not have any null value for ProductKey. This is one of the important column for my business requirement. So we cannot accept null value for Product Key.So we are doing some sanity check shown below.

```
dfNull= df.filter("ProductKey is null")
dfNull.display()
```

	SalesOrderLineKey	ResellerKey	CustomerKey	ProductKey	OrderDateKey	DueDateKey	ShipDateKey
1	51690002	663	-1	null	20190801	20190811	20190808

we can see only 1 one record as shown above.

Now we have to understand from which file this record is getting created, then based on that we have to locate and we have to write a mail to our source team, we have to ask for a reason why productkey is null for this record or we have to get updated file. we need to identify now which file has issue.Totally there are 36 files in Input File Name folder .we cant open 36 files manually and check. So in this case input_file_name function we can use.

```
dfInputFileName = df.withColumn("input_file_name",F.input_file_name())
dfInputFileName.display()
```

	ProductStandardCost	TotalProductCost	SalesAmount,,,,,	input_file_name
1	$1,912.15	$1,912.15	$3,399.99	dbfs:/mnt/adls/Input_File_Name/2017/Q4/sales_201710.csv
2	$2,171.29	$2,171.29	$3,578.27	dbfs:/mnt/adls/Input_File_Name/2017/Q4/sales_201710.csv
3	$1,912.15	$1,912.15	$3,399.99	dbfs:/mnt/adls/Input_File_Name/2017/Q4/sales_201710.csv
4	$1,898.09	$1,898.09	$3,374.99	dbfs:/mnt/adls/Input_File_Name/2017/Q4/sales_201710.csv
5	$1,912.15	$1,912.15	$3,399.99	dbfs:/mnt/adls/Input_File_Name/2017/Q4/sales_201710.csv
6	$2,171.29	$2,171.29	$3,578.27	dbfs:/mnt/adls/Input_File_Name/2017/Q4/sales_201712.csv
7				

Showing all 178 rows.

So here input_file_name column got created for each record as shown above with corresponding file name for each record. We are passing this data to some downstream application but later we can rectify if we find there is a problem.

Now we can do sanity check by filtering data which is of null,as shown below.

```
dfInputFileName=df.withColumn("input_file_name",F.input_file_name()).
filter("ProductKey is null")
dfInputFileName.display()
```

	ProductStandardCost	TotalProductCost	SalesAmount,,,,,	input_file_name
1	$47.29	$47.29	$63.90	dbfs:/mnt/adls/Input_File_Name/2019/Q3/sales_201908.csv

Showing all 1 rows.

Now we don't need to manually check which file is getting problem ,instead we can easily identify file name using this function input_file_name().

Now we can go through this input_file_name path , Input_File_Name Folder →2019 →Q3→ sales_201908.csv file. So, from this file we are getting ProductKey as null. So, like this we can easily identify and we can write drop a mail to source team with this file name.

This input_file_name() function mainly used for trouble shooting and debugging.Also it's a wise decision to add date ingested column(current_timestamp()) along with input file name. Why timestamp column along with input file name is immediately we are not capturing corrupt record and we are passing data to downstream applications but 1 week later or 1 month later we come to know there is a problem with one of the record. So, we have to understand what the file name is. If we are going to push date ingested also along with input file name , it will help us to speed up the process on which particular day that file arrived.and on which particular day we processed, i.e; ingested that particular record.

```
df1= dfInputFileName.withColumn("date_ingested", F.current_timestamp())
display(df1)
```

	roductCost	SalesAmount,,,,,	input_file_name	date_ingested
1		$63.90	dbfs:/mnt/adls/Input_File_Name/2019/Q3/sales_201908.csv	2022-04-11T11:10:11.780+0000

Showing all 1 rows.

So, we are pushing this data to data warehouse or data base . later after a month we are exploring we found there is a problem , then easily we can trace it back using date ingested and input file name.

WINDOW FUNCTIONS LEAD AND LAG

Window functions is very commonly used transformation in all the projects. It's very rare to see any development with out using window function. Window function has many transformations such as rownumber, rank,denserank,first,ntile,lead and lag etc.
Window function is nothing but splitting the data into windows/partitions/bucket. Splitting large data into windows based on a particular key then applying the transformation for each window, that is called window function.

Let's say we have a dataframe containing 3 columns name,department and salary. For sales department there are 5 records. For finance there are 3 records and for marketing there are 2 records. Now I want to split this data based on particular key that is department. Now I want to create partition based on department then it will create 3 partitions.Now I want to sort the data based on salary

Dataframe

name	department	salary
James	Sales	3000
Michael	Sales	4600
Robert	Sales	4100
David	Sales	3000
Saif	Sales	4100
Maria	Finance	3000
Scott	Finance	3300
Jen	Finance	3900
Jeff	Marketing	3000
Kumar	Marketing	2000

Window Partitioned

name	department	salary
James	Sales	3000
Michael	Sales	4600
Robert	Sales	4100
David	Sales	3000
Saif	Sales	4100
Maria	Finance	3000
Scott	Finance	3300
Jen	Finance	3900
Jeff	Marketing	3000
Kumar	Marketing	2000

Window.partitionBy("department")

Window Partitioned Ascending by sales

name	department	salary
James	Sales	3000
David	Sales	3000
Robert	Sales	4100
Saif	Sales	4100
Michael	Sales	4600
Maria	Finance	3000
Scott	Finance	3300
Jen	Finance	3900
Kumar	Marketing	2000
Jeff	Marketing	3000

Window.partitionBy("department").orderBy("salary")

Window Partitioned Descending by name

name	department	salary
Saif	Sales	4100
Robert	Sales	4100
Michael	Sales	4600
James	Sales	3000
David	Sales	3000
Scott	Finance	3300
Maria	Finance	3000
Jen	Finance	3900
Kumar	Marketing	2000
Jeff	Marketing	3000

Window.partitionBy("department").orderBy("name").desc()

On top of this splitted data, we can apply any logic such as rownumber, rank,denserank, lead,lag,ntile.
If we apply lead , lag on above data , we will get result like as below.

LEAD

name	department	salary	Lead
James	Sales	3000	3000
David	Sales	3000	4100
Robert	Sales	4100	4100
Saif	Sales	4100	4600
Michael	Sales	4600	NULL
Maria	Finance	3000	3300
Scott	Finance	3300	3900
Jen	Finance	3900	NULL
Kumar	Marketing	2000	3000
Jeff	Marketing	3000	NULL

LAG

name	department	salary	Lag
James	Sales	3000	NULL
David	Sales	3000	3000
Robert	Sales	4100	3000
Saif	Sales	4100	4100
Michael	Sales	4600	4100
Maria	Finance	3000	NULL
Scott	Finance	3300	3000
Jen	Finance	3900	3300
Kumar	Marketing	2000	NULL
Jeff	Marketing	3000	2000

lead function applied on top of salary column ,so it will create another column based on salary column, we will get lead value which means what is the corresponding leading record, next record. If we want to get leading value of 2 records after current record, we will write lead(column,2). Similarly, if we want to get previously available value , we will get using lag.

Code: Sample DataFrame

```
simpleData = (("James", "Sales", 3000), \
    ("Michael", "Sales", 4600),  \
    ("Robert", "Sales", 4100),   \
    ("James", "Sales", 3000),  \
    ("Saif", "Sales", 4100), \
    ("Maria", "Finance", 3000),  \
    ("Scott", "Finance", 3300),  \
    ("Jen", "Finance", 3900),    \
    ("Jeff", "Marketing", 3000), \
    ("Kumar", "Marketing", 2000)
  )

columns= ["employee_name", "department", "salary"]
df = spark.createDataFrame(data = simpleData, schema = columns)

df.show()
```

```
+-------------+----------+------+
|employee_name|department|salary|
+-------------+----------+------+
|        James|     Sales|  3000|
|      Michael|     Sales|  4600|
|       Robert|     Sales|  4100|
|        James|     Sales|  3000|
|         Saif|     Sales|  4100|
|        Maria|   Finance|  3000|
|        Scott|   Finance|  3300|
|          Jen|   Finance|  3900|
|         Jeff| Marketing|  3000|
|        Kumar| Marketing|  2000|
+-------------+----------+------+
```

```
#Window Definition
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number
windowSpec = Window.partitionBy("department").orderBy("salary")

#Lag Window Function
from pyspark.sql.functions import lag
df.withColumn("lag",lag("salary",1).over(windowSpec)) \
   .show()
```

Here lag("salary",1) means it will take previous salary only without skipping rows.

```
|employee_name|department|salary| lag
+-------------+----------+------+----
|        Maria|   Finance|  3000|null
|        Scott|   Finance|  3300|3000
|          Jen|   Finance|  3900|3300
|        Kumar| Marketing|  2000|null
|         Jeff| Marketing|  3000|2000
|        James|     Sales|  3000|null
|        James|     Sales|  3000|3000
|       Robert|     Sales|  4100|3000
|         Saif|     Sales|  4100|4100
|      Michael|     Sales|  4600|4100
```

If we use 2 here in lag function, output will be like below.

```
from pyspark.sql.functions import lag
df.withColumn("lag",lag("salary",2).over(windowSpec)) \
   .show()
```

```
|employee_name|department|salary| lag|
+-------------+----------+------+----+
|        Maria|   Finance|  3000|null|
|        Scott|   Finance|  3300|null|
|          Jen|   Finance|  3900|3000|
|        Kumar| Marketing|  2000|null|
|         Jeff| Marketing|  3000|null|
|        James|     Sales|  3000|null|
|        James|     Sales|  3000|null|
|       Robert|     Sales|  4100|3000|
|         Saif|     Sales|  4100|3000|
|      Michael|     Sales|  4600|4100|
```

```
#Lead Window Function
from pyspark.sql.functions import lead
df.withColumn("lead",lead("salary",1).over(windowSpec)) \
   .show()
```

employee_name	department	salary	lead
Maria	Finance	3000	3300
Scott	Finance	3300	3900
Jen	Finance	3900	null
Kumar	Marketing	2000	3000
Jeff	Marketing	3000	null
James	Sales	3000	3000
James	Sales	3000	4100
Robert	Sales	4100	4100
Saif	Sales	4100	4600
Michael	Sales	4600	null

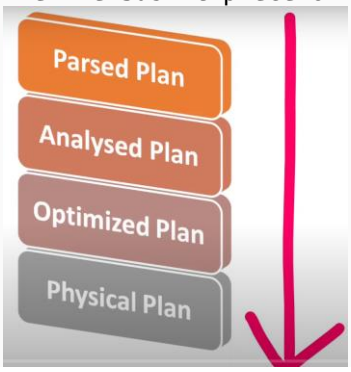
```
from pyspark.sql.functions import lead
df.withColumn("lead",lead("salary",2).over(windowSpec)) \
   .show()
```

employee_name	department	salary	lead
Maria	Finance	3000	3900
Scott	Finance	3300	null
Jen	Finance	3900	null
Kumar	Marketing	2000	null
Jeff	Marketing	3000	null
James	Sales	3000	4100
James	Sales	3000	4100
Robert	Sales	4100	4600
Saif	Sales	4100	null
Michael	Sales	4600	null

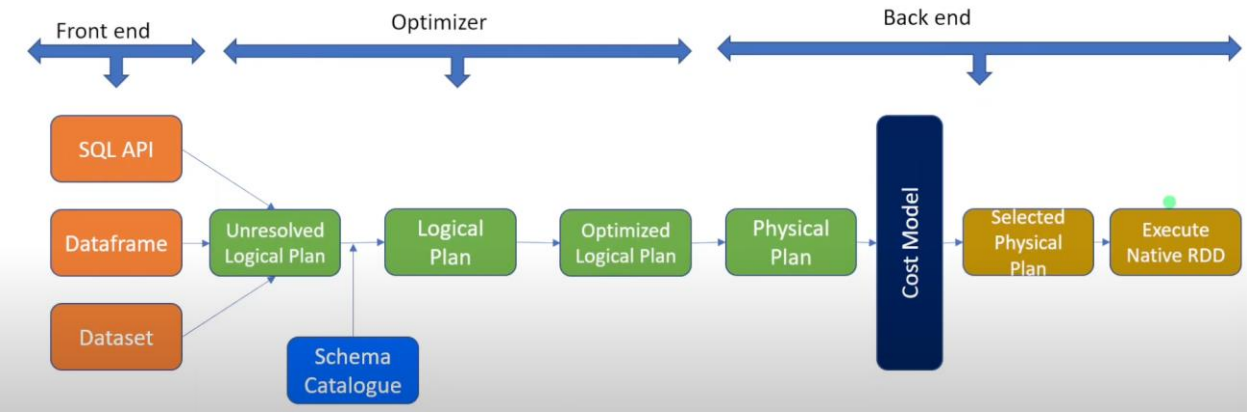
EXPLAIN PLAN

Explain Plan is used to choose or build efficient execution plan while submitting some code. Let's say we are submitting particular application called code. When we are submitting piece of code for execution into spark environment then that piece of code goes through many phases and at the end it is going to choose best and efficient execution plan for execution. So, for that execution plan concept is used.

When we submit piece of code, it goes through many phases.



Initially it will generate Parsed Plan, then Analysed Plan, Then Optimized Plan then Physical Plan. **Note: look into catalyst optimizer content.** Parsed Logic Plan also call as unresolved logical plan. Parsing means checking syntax and semantic validation. When we are writing any SQL statement, we need to check proper syntax. If we give wrong syntax like adding column name in pyspark, in withcolumn c should be in caps . if c is in small case, then it will show as syntax error. This is called syntax validation.Once syntax validation is done, then system will go to semantic validation which means it is object validation, we are referring one delta table in the SQL query but delta table is not available in the system.We are referring one column but that column is not associated with the delta table. This is called semantic check, we are referring some object but which is not present. This is called semantic check.



So, in parsed/unresolved logical plan, syntax/semantic checks will be verified. Apart from syntax and semantic checks , there could be many calculated columns in the code but how to determine data type or name of those columns, now those will be unresolved. In end of this stage, it's going to create unresolved/parsed logical plan. Then the next stage is logical/Analysed plan. Once logical plan is created then the spark engine will refer the schema catalog which will hold the metadata about objects. For example, in one of delta table, what is the number of columns and name of the columns .what is the data type of each column, those information will be kept in schema catalog. So whatever objects we are using in our query everything will be validated and accordingly data type will be assigned into logical plan and in our code we might have number of calculated fields and for that proper name and proper data type would be assigned in the logical plan. Coming to logical plan, it follows rule-based optimizer.In rule-based optimizer, there are certain prewritten rules. Based on execution engine,it will generate execution plan. If we give select * from table where col= . As per prewritten rule, first it has to filter the data and then based on that prewritten book that is actually in the binary tree mode. So based on that prewritten rules, it will create logical plan. Once logical plan is created, then that is the input for next stage that is optimized logical plan.

In the optimized logical plan, engine will look into logical plan and will try to optimize as much as possible. For example, in our code we are having 2 filter conditions, the first filter is going to filter only 10% of data and 2nd filter is

going to limit 70% of data. In this case, what we have to do is first we have to apply right filter which will eliminate most of the data. So as a result, for the second stage we can carry forward only limited data. So, this is called rearranging the filter or predicate push down, projection push down. These kind of optimization will be applied to the logical plan. At the end of this process, optimized logical plan will be created.

Once optimized logical plan is created, engine control goes to the next plan i.e; physical plan. In physical plan, for optimized logical plan it will create n number of physical plan. Its an internal detail step how it can be executed.so for this step engine will use cost-based optimizer. In cost-based optimizer basically it will refer statistics. Statistics is nothing but whenever we are having delta table or any object, it will collect statistics metadata about the table. It will keep all the columns along with its minimum value, max value, no of rows count. It will keep lot of statistics.By using that statistics, physical plan will be created. These are the 4 different stages for creating the plan. Once n number of physical plans are created then optimizer will go to cost model. Cost model for generated n number of physical plans it will start calculating the cost. Cost is nothing but how much time each internal step will take and how much resource its going to utilize. Based on that, it will create cost.it will assign cost for each model. Let's say we are having 1000's of model and cost will be created for each physical plan. Then once cost is done, then finally engine will choose best efficient plan which means which plan is going to take less time and also its going to use less computing resources. That plan will be chosen for the execution. Then once plan is chosen, for the chosen plan it will convert the code into JAR files.Then the system is ready to submit the code to executors in the form of jar files. So, for each application, its going to execute the code in the form of jobs ,stages and tasks. Then code will be created in the form of jar file. That will be submitted to executor by driver. This is the complete life cycle of catalyst optimizer. Based on Catalyst Optimizer, explain plan will be created.

CODE-

CREATE EMPLOYEE DATA FRAME:

```
employee_schema=["employee_id","name","doj","dept_id","gender","salary"]
employee_data =[(10,"Raj","1999","100","M",2000),
                (20,"Rahul","2002","200","M",8000),
                (30,"Raghav","2010","100","",6000),
                (40,"Raja","2004","100","F",7000),
                (50,"Rama","2008","400","F",1000),
                (60,"Rasul","2014","500","M",5000)]
employeeDF= spark.createDataFrame(data=employee_data,schema=employee_schema)
display(employeeDF)
```

	employee_id	name	doj	dept_id	gender	salary
1	10	Raj	1999	100	M	2000
2	20	Rahul	2002	200	M	8000
3	30	Raghav	2010	100		6000
4	40	Raja	2004	100	F	7000
5	50	Rama	2008	400	F	1000
6	60	Rasul	2014	500	M	5000

```
#Create Department Dataframe
department_data= [("HR",100),("Supply",200),("Sales",300),("Stock",400)]
department_schema =["dept_name","dept_id"]
departmentDF= spark.createDataFrame(data=department_data,schema=department_schema)
display(departmentDF)
```

	dept_name	dept_id
1	HR	100
2	Supply	200
3	Sales	300
4	Stock	400

```
#INNER JOIN
from pyspark.sql.functions import col
dfJoin = employeeDF
.join(departmentDF,employeeDF.dept_id==departmentDF.dept_id,"inner") \
.withColumn("bonus",col("salary")*0.1) \
.groupBy("dept_name").sum("salary")

display(dfJoin)
```

	dept_name	sum(salary)
1	Stock	1000
2	HR	15000
3	Supply	8000

Internally if I want to see how above particular code is getting executed, we can see and understand via explain plan for this query.

`dfJoin.explain()` using this query we will get explain plan

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[dept_name#53], functions=[finalmerge_sum(merge sum#104L) AS sum(salary#18L)#97L])
   +- Exchange hashpartitioning(dept_name#53, 200), ENSURE_REQUIREMENTS, [id=#410]
      +- HashAggregate(keys=[dept_name#53], functions=[partial_sum(salary#18L) AS sum#104L])
         +- Project [salary#18L, dept_name#53]
            +- SortMergeJoin [cast(dept_id#16 as bigint)], [dept_id#54L], Inner
               :- Sort [cast(dept_id#16 as bigint) ASC NULLS FIRST], false, 0
               :  +- Exchange hashpartitioning(cast(dept_id#16 as bigint), 200), ENSURE_REQUIREMENTS, [id=#402]
               :     +- Project [dept_id#16, salary#18L]
               :        +- Filter isnotnull(dept_id#16)
               :           +- Scan ExistingRDD[employee_id#13L,name#14,doj#15,dept_id#16,gender#17,salary#18L]
               +- Sort [dept_id#54L ASC NULLS FIRST], false, 0
                  +- Exchange hashpartitioning(dept_id#54L, 200), ENSURE_REQUIREMENTS, [id=#403]
                     +- Filter isnotnull(dept_id#54L)
                        +- Scan ExistingRDD[dept_name#53,dept_id#54L]
```

Here in explain plan, it has produced only physical plan, but catalyst optimizer will produce multiple plans but we can see only physical plan here because whenever we are giving explain function, if we don't pass any parameter it is going to consider only physical plan.

Now how to read physical plan of above? Whenever we have to read any explain plan,we have to start from bottom, so it is actually bottom-up approach. In spark , we have DAG-Directed Acyclic Graph -it is top-down approach, we have to start reading from top. But coming to explain plan,we will read it from bottom. Physical plan is the final stage. This is the chosen best efficient plan by our spark engine.

1st step: We have to start from bottom so first what happens is it is scanning (Scan Existing RDD[dept_name#53,dept_id#54L]. Basically, in our query, we are doing joining between two data frames and then doing aggregation process and we need to get sum of salary for each department. So, first join will happen and then aggregation will happen. So, for join , first it has to scan the data , it has to produce the data frame of a department data frame so for that it is scanning then it is containing department name and department id.So in our department data frame we are having dept_name and dept_id. So basically, it is scanning the data for department data frame.

2nd step: Once that is done, next step is planning to perform inner join which means id based on dept_id so we cant accept any null value because at the end after performing inner join if there is any null record that will be eliminated.so as a first step, it is going to filter not null records based on dept_id .so that is happening in 2nd step(Filter isnotnull(dept_id#54L)) .

3rd step: Now we are planning to join , so in order to perform join, join is one of the wide transformation.so as part of wide transformation, there will be data shuffle. So first of all data should be ordered properly for a department dataframe . So here

partition exchange happen. based on that we have to join based on dept_id. So that's the reason hash partitioning exchange there based on dept_id.

4th step: Once hash partitioning exchange done, then finally sorting data based on dept_id and nulls first. When we join later then we can join based on sort merge join. So that's the reason, it is exchanging the data. Based on that, it is rearranging the data then finally it is sorting. So, this is done for one data frame that is departmentDF.

5th step: Coming to next data frame , it is starting from Scan ExistingRDD- the other data frame employeeDF. employeeDF containing fields employee_id,name,doj,dept_id,gender and salary. Internally engine will give assign some id for each column because it's used for its own reference so if we can see some number that is assigning unique id (like 13L,14,15,16,17,18L) for each object or column.

```
   +- Scan ExistingRDD[employee_id#13L,name#14,doj#15,dept_id#16,gender#17,salary#18L]
```

So once scanning is done, again we are planning to join this employeeDF with department df so using dept_id it has to generate all the non-null records.so it is applying the Filter isnotnull based on dept_id. Finally, in employeeDF, we are having many fields(employee_id,name,doj,dept_id,gender,salary) but if we look at our join query, after joining mostly we are using only dept_id and salary for this one. And for another data frame we are using only dept_id and dept_name for joining.so that's the reason even though we are having many other cols in employeeDF they are not used in the calculation.Thats the reason our explain plan go with projection. Projection means choosing the right columns which columns we are choosing that is called projection.

```
+- SortMergeJoin [cast(dept_id#16 as bigint)], [dept_id#54L], Inner
   :- Sort [cast(dept_id#16 as bigint) ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(cast(dept_id#16 as bigint), 200), ENSURE_REQUIREMENTS, [id=#402]
   :     +- Project [dept_id#16, salary#18L]
   :        +- Filter isnotnull(dept_id#16)
   :           +- Scan ExistingRDD[employee_id#13L,name#14,doj#15,dept_id#16,gender#17,salary#18L]
```

So, output of this employeeDF its going to choose only dept_id and salary because we are not interested in any other columns. Any other cols are not used for any calculation. Once that is done, again its going to exchange because we have to perform join so that's the reason its doing exchange same like departmentDF, then finally it is doing sort. Once that is done, based on sort merge join, join process is happening. So, in data bricks in spark, we have various joining methods like broadcast join, sort merge join, shuffle join . So, by default if we don't use broadcast join, sort merge join will be applied to generate output for joining multiple dataframes. So that's the reason we are getting output of join till here as shown below. Basically, join happened and at the end of join , we are interested in only salary and dept_name. so that's the reason after joining we will get dept_id from both the dataframes but we are not interested in that. Even dept_id is getting discarded in Project step . at the end of join its going to project only 2 columns ,salary and dept_name.

```
+- Project [salary#18L, dept_name#53]
   +- SortMergeJoin [cast(dept_id#16 as bigint)], [dept_id#54L], Inner
      :- Sort [cast(dept_id#16 as bigint) ASC NULLS FIRST], false, 0
      :  +- Exchange hashpartitioning(cast(dept_id#16 as bigint), 200), ENSURE_REQUIREMENTS, [id=#402]
      :     +- Project [dept_id#16, salary#18L]
      :        +- Filter isnotnull(dept_id#16)
      :           +- Scan ExistingRDD[employee_id#13L,name#14,doj#15,dept_id#16,gender#17,salary#18L]
      +- Sort [dept_id#54L ASC NULLS FIRST], false, 0
         +- Exchange hashpartitioning(dept_id#54L, 200), ENSURE_REQUIREMENTS, [id=#403]
            +- Filter isnotnull(dept_id#54L)
               +- Scan ExistingRDD[dept_name#53,dept_id#54L]
```

6th step: Once join is done and we got the output , now we have to perform groupBy because in our code we have given groupBy also. First, we are doing join and then we are doing groupBy. So, for that groupBy, one second it will start exchanging the data based on dept_name because groupBy we are doing in query on dept_name.Thats the reason it starts exchanging the data. Once that is done it is performing aggregate operation dept_name and sum(salary).

```
+- HashAggregate(keys=[dept_name#53], functions=[finalmerge_sum(merge sum#104L) AS sum(salary#18L)#97L])
   +- Exchange hashpartitioning(dept_name#53, 200), ENSURE_REQUIREMENTS, [id=#410]
```

so this is the final output and end of the process and it will start giving the output.

This is the entire life cycle of explain plan. Basically, it is detailing out what are the internal steps how our code will be executed internally.

One more thing in the explain plan, AdaptiveSparkPlan isFinalPlan=false. Basically, AQE plan is nothing but one of the performance optimization technique provided by data bricks. Using that engine will automatically perform certain performance optimization such as handling data skew, applying broadcast join, applying coalesce. These kind of techniques will be applied by AQE. So in the query in the all physical plans we can see AdaptiveSparkPlan isFinalPlan=false which means we are not using AQE plan at the moment because Adaptive query plan is mainly based on dynamic statistics. So even though it is showing false in the explain plan but whenever we are executing the code and if we go to spark UI, we can see based on live or dynamic statistics , some place it will start using adaptive spark plan. There we can see it is set to true. But in the physical plan,in the explain plan we can always see AdaptiveSparkPlan isFinalPlan=false to false.

```
dfJoin = employeeDF
.join(departmentDF,employeeDF.dept_id==departmentDF.dept_id,”inner”) \
.withColumn(“bonus”,col(“salary”)*0.1) \
.groupBy(“dept_name”).sum(“salary”)

display(dfJoin)
```

In this query,we added one calculated column bonus but in our physical plan, we could not see any reference because we calculated this bonus column but still that is not used anywhere.so internally in between it got calculated and it got dropped.So it is not useful for our final output.so that's the reason our engine will come to conclusion that there is no point in calculating this unnecessarily. We are wasting our time and also resources in calculating. So that's the reason in our final chosen or selected plan we could not see the reference but at the same time if we are looking at analyzed or parsed plan, we will see the references.
Now we can see different variations of explain plan

dfJoin.explain(extended=True)

```
== Parsed Logical Plan ==
'Aggregate ['dept_name], ['dept_name, sum(salary#18L) AS sum(salary)#98L]
+- Project [employee_id#13L, name#14, doj#15, dept_id#16, gender#17, salary#18L, dept_name#53, dept_id#54L, (cast(salary#18L as double) *
0.1) AS bonus#78]
   +- Join Inner, (cast(dept_id#16 as bigint) = dept_id#54L)
      :- LogicalRDD [employee_id#13L, name#14, doj#15, dept_id#16, gender#17, salary#18L], false
      +- LogicalRDD [dept_name#53, dept_id#54L], false

== Analyzed Logical Plan ==
dept_name: string, sum(salary): bigint
Aggregate [dept_name#53], [dept_name#53, sum(salary#18L) AS sum(salary)#98L]
+- Project [employee_id#13L, name#14, doj#15, dept_id#16, gender#17, salary#18L, dept_name#53, dept_id#54L, (cast(salary#18L as double) *
0.1) AS bonus#78]
   +- Join Inner, (cast(dept_id#16 as bigint) = dept_id#54L)
      :- LogicalRDD [employee_id#13L, name#14, doj#15, dept_id#16, gender#17, salary#18L], false
      +- LogicalRDD [dept_name#53, dept_id#54L], false
```

Here we can see all the plans now. If we want to see all the plans, we need to use the parameter extended=True.

Now coming to parsed logical plan in the explain plan, first it is deciding we have to create departmentDF(dept_name and dept_id)logical RDD and 2nd data frame is employee data frame containing employee_id,name,doj,dept_id,gender and salary then perform inner join based on dept_id on both sides i.e; on both data frames. Now we need to project, we need to choose the column. During projection along with all the columns from both the data frame, it has to add one more extra column that is bonus(salary*0.1). After projection, we have to groupBy aggregation. groupBy dept_name then sum(salary).This is the final output. In parsed logical plan, engine has verified only semantic and syntax checks. Here in parsed logical plan, we could not see any keyword like unresolved but when we are having some complex query and many calculated fields are created then we can see keyword like unresolved in many places in this parsed logical plan.

Once this is done, Analyzed logical plan is created by referring schema catalog. Then in each column, type would be determined.For example , in parsed logical plan, it has not given final output and what is the data type.it was showing only column names.But coming to Analyzed logical plan, it is giving dept_name string and sum(salary) as bigint. Basically, it has referred schema catalog. Now from the catalog it has

understood dept_name is string type and salary field is bigint. This is how in determining data type.

```
== Optimized Logical Plan ==
Aggregate [dept_name#53], [dept_name#53, sum(salary#18L) AS sum(salary)#98L]
+- Project [salary#18L, dept_name#53]
   +- Join Inner, (cast(dept_id#16 as bigint) = dept_id#54L)
      :- Project [dept_id#16, salary#18L]
      :  +- Filter isnotnull(dept_id#16)
      :     +- LogicalRDD [employee_id#13L, name#14, doj#15, dept_id#16, gender#17, salary#18L], false
      +- Filter isnotnull(dept_id#54L)
         +- LogicalRDD [dept_name#53, dept_id#54L], false
```

Coming to optimized logical plan,we will do any predicate push down if logical plan can be optimized in any way.so those steps will be carried out in optimized logical plan. Apart from that steps is same in optimized logical plan.

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[dept_name#53], functions=[finalmerge_sum(merge sum#104L) AS sum(salary#18L)#97L], output=[dept_name#53, sum(sala
ry)#98L])
   +- Exchange hashpartitioning(dept_name#53, 200), ENSURE_REQUIREMENTS, [id=#410]
      +- HashAggregate(keys=[dept_name#53], functions=[partial_sum(salary#18L) AS sum#104L], output=[dept_name#53, sum#104L])
         +- Project [salary#18L, dept_name#53]
            +- SortMergeJoin [cast(dept_id#16 as bigint)], [dept_id#54L], Inner
               :- Sort [cast(dept_id#16 as bigint) ASC NULLS FIRST], false, 0
               :  +- Exchange hashpartitioning(cast(dept_id#16 as bigint), 200), ENSURE_REQUIREMENTS, [id=#402]
               :     +- Project [dept_id#16, salary#18L]
               :        +- Filter isnotnull(dept_id#16)
               :           +- Scan ExistingRDD[employee_id#13L,name#14,doj#15,dept_id#16,gender#17,salary#18L]
```

---

dfJoin.explain(mode="simple")
This query is same as dfJoin.explain().
When we are calling explain without any parameter that is going to be equivalent of simple mode.

dfJoin.explain(mode="extended")
This will give us 4 different plans.

dfJoin.explain(mode="formatted")
This will give plan in the formatted way which means it will give in more readable format.

```
== Physical Plan ==
AdaptiveSparkPlan (15)
+- HashAggregate (14)
   +- Exchange (13)
      +- HashAggregate (12)
         +- Project (11)
            +- SortMergeJoin Inner (10)
               :- Sort (5)
               :  +- Exchange (4)
               :     +- Project (3)
               :        +- Filter (2)
               :           +- Scan ExistingRDD (1)
               +- Sort (9)
                  +- Exchange (8)
                     +- Filter (7)
                        +- Scan ExistingRDD (6)
```

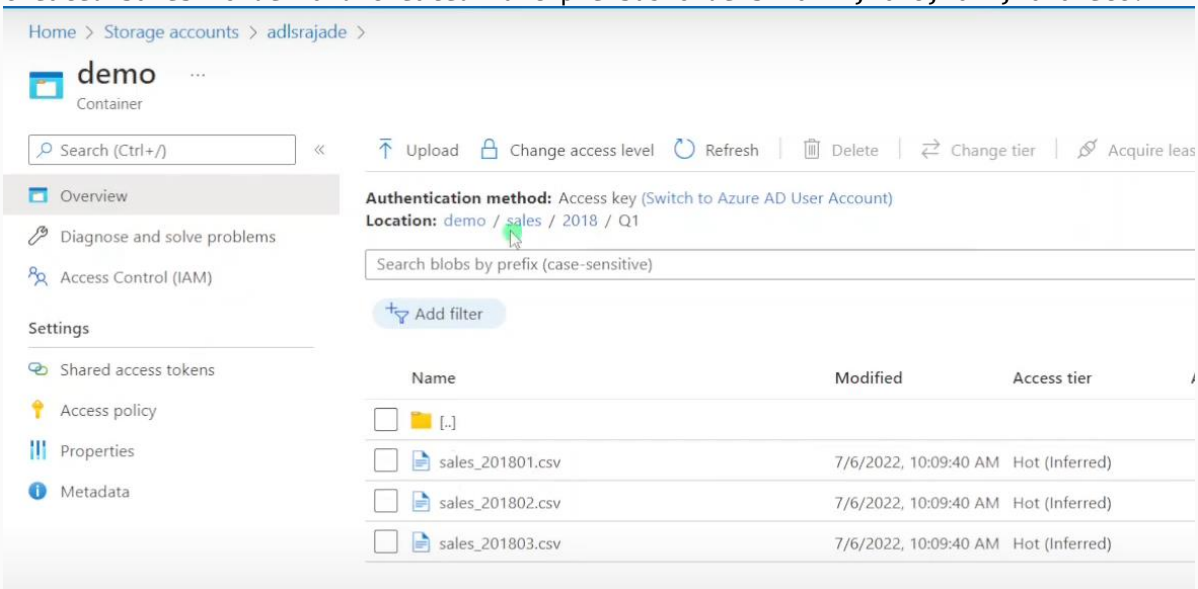
Here nos 1,2,3,4,5,6 will be explained in more detail as shown below.

```
(1) Scan ExistingRDD
Output [6]: [employee_id#13L, name#14, doj#15, dept_id#16, gender#17, salary#18L]
Arguments: [employee_id#13L, name#14, doj#15, dept_id#16, gender#17, salary#18L], MapPartitionsRDD[4] at applySchemaToPythonRDD at Nati
veMethodAccessorImpl.java:0, ExistingRDD, UnknownPartitioning(0)

(2) Filter
Input [6]: [employee_id#13L, name#14, doj#15, dept_id#16, gender#17, salary#18L]
Condition : isnotnull(dept_id#16)

(3) Project
Output [2]: [dept_id#16, salary#18L]
Input [6]: [employee_id#13L, name#14, doj#15, dept_id#16, gender#17, salary#18L]

(4) Exchange
Input [2]: [dept_id#16, salary#18L]
Arguments: hashpartitioning(cast(dept_id#16 as bigint), 200), ENSURE_REQUIREMENTS, [id=#402]
```

dfJoin.explain(mode="cost")
This will give us only 2 plans Optimized Logical plan and Physical Plan.

UDF Check If Folder Exists

In todays bigdata world, we are often dealing with semi structured data. This data is in the form of files distributed across multiple folders.
In this scenario, while developing certain business functionality we often come across a situation where we need to check if a particular folder exists or not.so we created UDF to handle this situation.
Let's say we created storage account as adlsrajade and container named as demo and created sales folder and created multiple subfolders 2017 ,2018,2019,2020 etc.



Now we need to check if particular folder is present in the container or not.
In order to read data from ADLS , we have to create mount point.in mount point we have to give details about data lake storage along with access key.

```python
dbutils.fs.mount(
      source = "wasbs://demo@adlsrajade.blob.core.windows.net",
      mount_point = "/mnt/adls_test",
      extra_configs =
{"fs.azure.account.key.adlsrajade.blob.core.windows.net":"MBnAFm5wvajLPJ4pYwsi0Tycq4zOogW2DW0w13MA0CZ+cTrY9swcLVp79+96MTfnkUAtsCtlt9zx+ASte90fNw=="})
```

```
Out[1]: True
```

Now we need to list down all files or folders present in sales folder.

%fs ls /mnt/adls_test/sales/

	path	name	size	modificationTime
1	dbfs:/mnt/adls_test/sales/2017/	2017/	0	0
2	dbfs:/mnt/adls_test/sales/2018/	2018/	0	0
3	dbfs:/mnt/adls_test/sales/2019/	2019/	0	0
4	dbfs:/mnt/adls_test/sales/2020/	2020/	0	0

```
def folderExists(path):
   try:
     dbutils.fs.ls(path)
     return True
   except Exception as err:
     if 'java.io.FileNotFoundException' in str(err):
        return False
```

Here we have created UDF. This UDF function accepting one parameter as path or folder location which we want to test. Inside we are giving try except. Within try, we are giving function as dbutils.fs.ls command. basically, we are trying to list down all the content under that particular folder. In case that folder is present, that command is successful. In case the folder is not present, then it will throw error it will go to exception part. In the exception, we are going to capture if it is throwing file not found exception then we are going to return false. basically, if folder is present then try block will be successful then based on that it will return true. Incase if folder is not present, then it will throw error java.io.FileNotFoundException, then based on that it will return false. So basically, it is returning one Boolean value true or false.

Now we need to test one of the folder, we need to check 2018 folder is present or not.

```
#folder path which we need to check if exists
path= "/mnt/adls_test/sales/2018"

if not folderExists(path):
   print("Folder is not present")
else:
```

```
    print("Folder is already present")
```

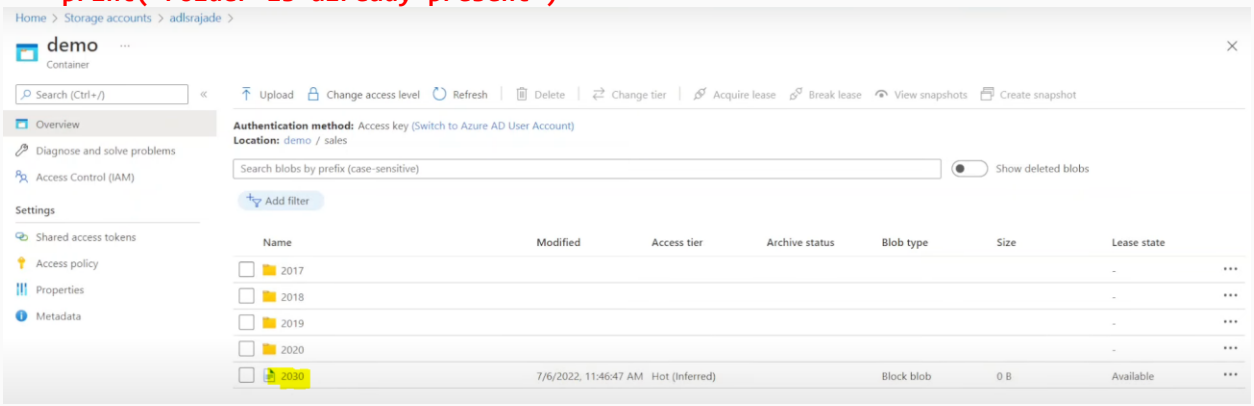
output: Folder is already present.

Here we are calling folderExists function, then we are going to pass this path as a parameter. If path is not found, it will return Folder is not present then if it go to else part its returning true, folder is already present.

```
# folder path which we need to check if exists
path="/mnt/adls_test/sales/2030/"

if not folderExists(path):
  print("Folder is not present")
else:
  print("Folder is already present")
```

In real time project, if folder doesn't exist , we can create as below, based on business requirement logic can be changed.

```
path= "/mnt/adls_test/sales/2030"
if not folderExists(path):
   print("Folder is not present")
   dbutils.fs.mkdirs(path)
else:
    print("Folder is already present")
```



SORT MERGE JOIN(SMJ)

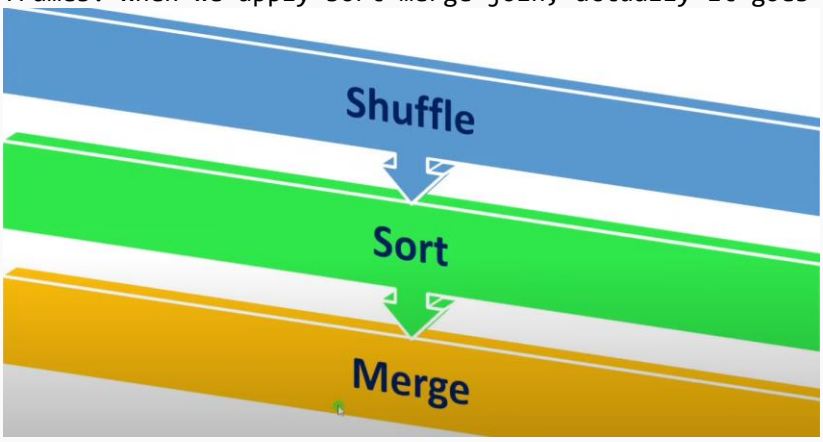
There are joining concepts like inner join default one, full outer join, anti-join, semi join. But these are available with developers' control, we can choose according to our requirement and we can apply. But when we apply those joinings, what's happening internally, how spark works internally to get that output, for that spark uses multiple different joining concepts.

JOIN TYPES : 3 different join types available in spark. Broadcast Join, Sort Merge Join, Shuffle Hash Join.

Broadcast join is one of the performance optimization. By default, it will be applied if we are joining with smaller dimension table. Let's say if we have to join big fact

table with small dimension table, now dimension table which is less than 10MB, then by default broadcast join will be applied by spark.

Sort Merge Join is the default joining method for spark when we are joining 2 data frames. When we apply sort merge join, actually it goes through 3 different phases.



The first phase is shuffle, then sorting and then merging.

We can understand this with an example.

Let's say we are having 2 data files, one is holding employee data , another is holding department data. We have to create data frame out of these 2 data files and we need to perform the join and based on that we have to get department name for each employee. That is the requirement.

Raw data

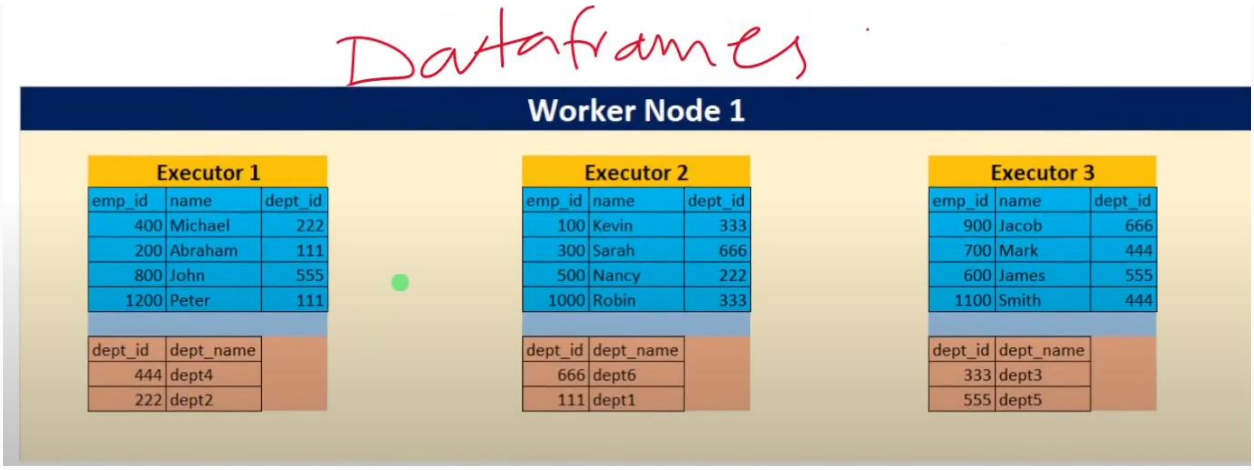
emp_id	name	dept_id
400	Michael	222
200	Abraham	111
800	John	555
1200	Peter	111
100	Kevin	333
300	Sarah	666
500	Nancy	222
1000	Robin	333
900	Jacob	666
700	Mark	444
600	James	555
1100	Smith	444

dept_id	dept_name
444	dept4
222	dept2
666	dept6
111	dept1
333	dept3
555	dept5

emp_id	name	dept_id	Dept_name
400	Michael	222	dept2
200	Abraham	111	dept1
800	John	555	dept5
1200	Peter	111	dept1
100	Kevin	333	dept3
300	Sarah	666	dept6
500	Nancy	222	dept2
1000	Robin	333	dept3
900	Jacob	666	dept6
700	Mark	444	dept4
600	James	555	dept5
1100	Smith	444	dept4

In dept data file, we are having 6 unique department names and it is not sorted in a proper way its scattered actually. Same for employee there are 12 records. For each department there are 2 records in employee data file and they are not sorted properly in employee data file too.

Let's assume this is our raw data , so based on that , first we are going to create data frame by reading these two data files from the storage. So After creating data frame , it would look like this as shown below,



Let's assume in our spark environment , we are having only 1 worker node but practically we can have more than 1 worker node as well. In this example we took 1 worker node which is having 3 executors. So, when we create data frame which is nothing but collection of partition distributed across multiple executors.

This csv file will be splitted in to 3 parts in this example but no of partitions will be decided by various parameters .

In this example let's assume this is the data file so while reading within the spark environment it is creating 3 partition. Each partition is containing just 4 records. This is how employeeDF got created. Now second dataframe departmentDF that is also splitted into 3 partition and 1 partition is residing within one executor which means out of those 6 records, 2 records are part of one partition that is stored as shown above.

Now if we look at above data, data is not sorted even within one executor within same on-heap memory. We are not having data for right combination. For example in our requirement we have to join based on dept_id, in this employeeDF dept_id, we are having keys like 2,1,5 and again 1. But coming to departmentDF of dept_id its having 444 and 222. So, there is no proper matching so in that case we cannot join. So, for that data should be shuffled so spark by default will use sort-merge join. First step is it will shuffle the data. So, for shuffling spark will decide which keys should go to which executor. For example, in this case let us assume spark is going to decide in the first executor in the first partition, it is going to retain only the dept_id's 111 and 222 and coming to 2nd partition in the 2nd executor it will retain dept_id 333 and 444. Coming to executor 3, it will retain dept_id as 555 and 666. So at the end of shuffle we are going to have employeeDF, departmentDF and each key will be matching and keys are not scattered around so there is no need to look the data from other executor.

So in the first step we shuffled the data. SO in the shuffle data base what happens is spark has decided to get only 111 and 222 in first partition. So it is just checking.Look at 1st record in 1st partition of executor1, 222 dept_id, it is placed here and 111 dept_id so placed here and 555 dept_id , that is not part of this partition1 because spark has decided to put it in the 3rd partition which will go to executor3. So this

is going to put 555 dept_id in executor 3 .and next 111 is part of 1st partition so it will be placed here

Shuffle Phase

emp_id	name	dept_id
400	Michael	222
200	Abraham	111
1200	Peter	111
500	Nancy	222

dept_id	dept_name
222	dept2
111	dept1

emp_id	name	dept_id
100	Kevin	333
1000	Robin	333
700	Mark	444
1100	Smith	444

dept_id	dept_name
444	dept4
333	dept3

emp_id	name	dept_id
900	Jacob	666
600	James	555
800	John	555
300	Sarah	666

dept_id	dept_name
555	dept5
666	dept6

Coming to 2nd partition in the 2nd executor, this partition should retain only 333 and 444. So, 1st record in executor 2 is 333 so it will be retained. Next record in executor2 is 666, actually it is part of 3rd partition then it will push that value to the next one. Next record in executor2 is 222, 222 should go to partition1 and next record is 333. It would be retained. Now its looking at next partition in executor3 we are having 444 dept_id as 2 records . Those 2 records will come and sit in this 2nd partition. So executor3 will just give up records which are not needed for 3rd partition i.e.; 444 to partition2 and partition3 will receive records that are part of 555 and 666.

With in shuffle phase, we are having 3 executors. Executor1 for 111 and 222. Executor2 for 333 and 444. Executor 3 for 555 and 666. With in each executor, within each on heap memory of executor we are having data for employeeDF and deptDF. If we look at the key, its matching 111 and 222 in both empDF and deptDF. These keys are not available with any other partitions. Each executor will contain partitions with matching keys. This is shuffle phase. But at the same time, if we look at the data carefully, it is not sorted based on the joining key. In this requirement, joining with dept_id but it is not sorted properly based on the dept_id . 1st partition of empDF and 1st partition of deptDF its not sorted based on the joining key.

Based on the shuffle data, sorting phase will start. In the sorting phase,dept_id 111 will go to top and dept_222 will come to an end. Internally it is sorting the data then partition is properly sorted and its happening within all partitions. SO, all the partitions(in executor) are sorted.

Sorting Phase

emp_id	name	dept_id
200	Abraham	111
1200	Peter	111
500	Nancy	222
400	Michael	222

dept_id	dept_name
111	dept1
222	dept2

emp_id	name	dept_id
100	Kevin	333
1000	Robin	333
700	Mark	444
1100	Smith	444

dept_id	dept_name
333	dept3
444	dept4

emp_id	name	dept_id
600	James	555
800	John	555
300	Sarah	666
900	Jacob	666

dept_id	dept_name
555	dept5
666	dept6

At the end of sorting phase what happens is it will contain partition of both dataframe. At the same time matching keys would be retained in the same executor. At the same time, data would be sorted properly.if we see executor1 , we will have 1st partition of employeeDF, and 1st partition of departmentDF. If we see both , keys are matching. Unique keys are 111 and 222. And at the same time, it is properly sorted. It's the same case with all the executors. This is how sorting phase happened. First data got shuffled and it was receiving data from other executors over network. In the 2nd phase, sorting happened it has internally sorted the data. So, from spark optimization perspective shuffling and sorting these are very costly operations and we have to try to avoid as much as possible but by default most of the joins sort merge join so it will involve shuffle and sorting operations.

Now we have performed shuffling and sorting then our data is ready to merge now. So, in the 3rd phase in the last phase merging will happen.in merging it will start merging 1 partition with the corresponding another partition within same executor. So, it is not depending on the data from another executor. If we will look at executor1 , partition1 of employeeDF and partition1 of departmentDF both can be merged. Similarly, within executor2, partition2 of employeeDF and partition2 of departmentDF both can be merged. Same goes for 3rd executor also it can be merged.

HOW MERGING HAPPENS? There is a certain algorithm for merging which is used by spark. So, executor 1 employeeDF partition1 dept_id is compared with the next data frame. We are having 111 that is matching. Now it will take that record and placed in output as shown below. Similarly, it will compare for all records. So, in one time it will compare from right table and another time it will compare from left table. There is a specific algorithm used by spark. What is advantage of sorting? We have sorted that's the reason we can just compare with the next record, that's enough. In case it is not sorted one value will be compared with list of many values. In practical world we will be having billions of records , its not possible to compare with billions of records each time. That's the reason we are performing sorting. So based on the sorting, each record will be compared with each record of another dataframe. Once it is matching or not matching, then based on that it will take other value from another dataframe and it will be compared with previous dataframe or partition once again. SO, this is the algorithm. Based on that , it is performing the merge then this will be output at the end of merge phase.

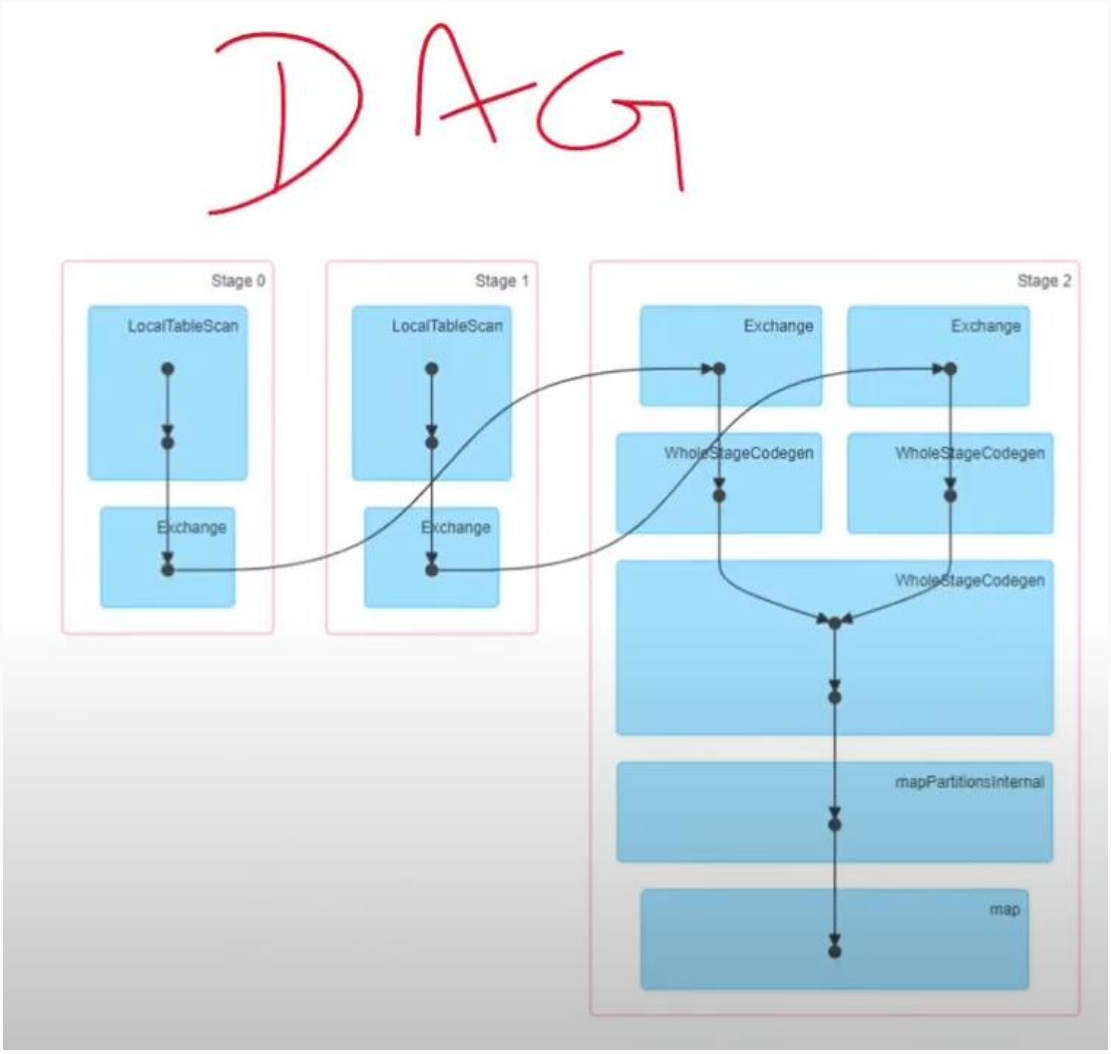
Merging Phase

emp_id	name	dept_id	dept_name		emp_id	name	dept_id	dept_name		emp_id	name	dept_id	dept_nam
200	Abraham	111	dept1		100	Kevin	333	dept3		600	James	555	dept5
1200	Peter	111	dept1		1000	Robin	333	dept3		800	John	555	dept5
500	Nancy	222	dept2		700	Mark	444	dept4		300	Sarah	666	dept6
400	Michael	222	dept2		1100	Smith	444	dept4		900	Jacob	666	dept6

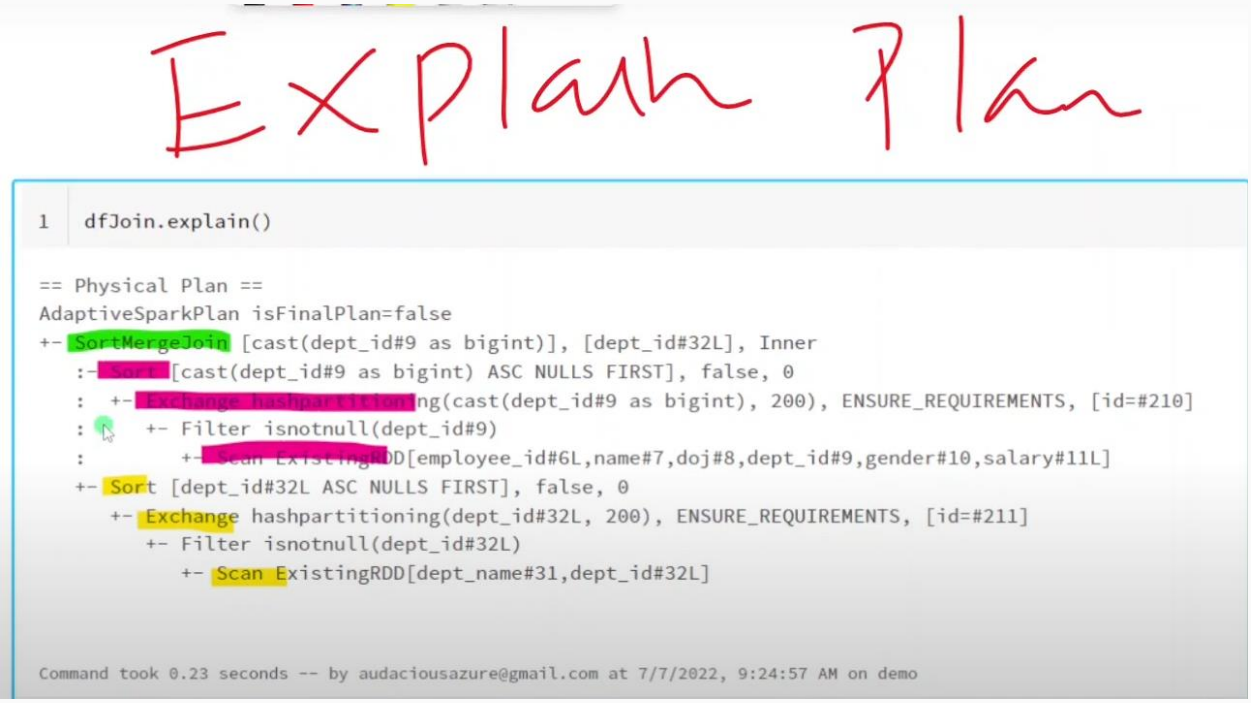
At the end of this, if we are going to display dataframe then each executor is going to pass their own output to the driver and driver will consult all the outputs and it will display in below format.

emp_id	name	dept_id	Dept_name
400	Michael	222	dept2
200	Abraham	111	dept1
800	John	555	dept5
1200	Peter	111	dept1
100	Kevin	333	dept3
300	Sarah	666	dept6
500	Nancy	222	dept2
1000	Robin	333	dept3
900	Jacob	666	dept6
700	Mark	444	dept4
600	James	555	dept5
1100	Smith	444	dept4

Now when we look at sparkUI, when we are performing sort merge join then it will look like below.



So, whenever we are joining 2 dataframes, first dataframe it would be scanned it would be read then it would exchange its nothing but wide transformation then it will shuffle the data which is happening in the shuffle phase. Same happening with other dataframe, its reading the raw data and then its exchanging in the shuffling phase. Once that is done, in the next stage sorting will happen. Based on the sorting again it will merge the data . So basically, it will create 3 stages.one stage for reading one data frame.2nd stage for reading another data frame and exchange is happening and based on the exchange, exchange data will start in the 3rd stage.so it will sort the data then it will merge.

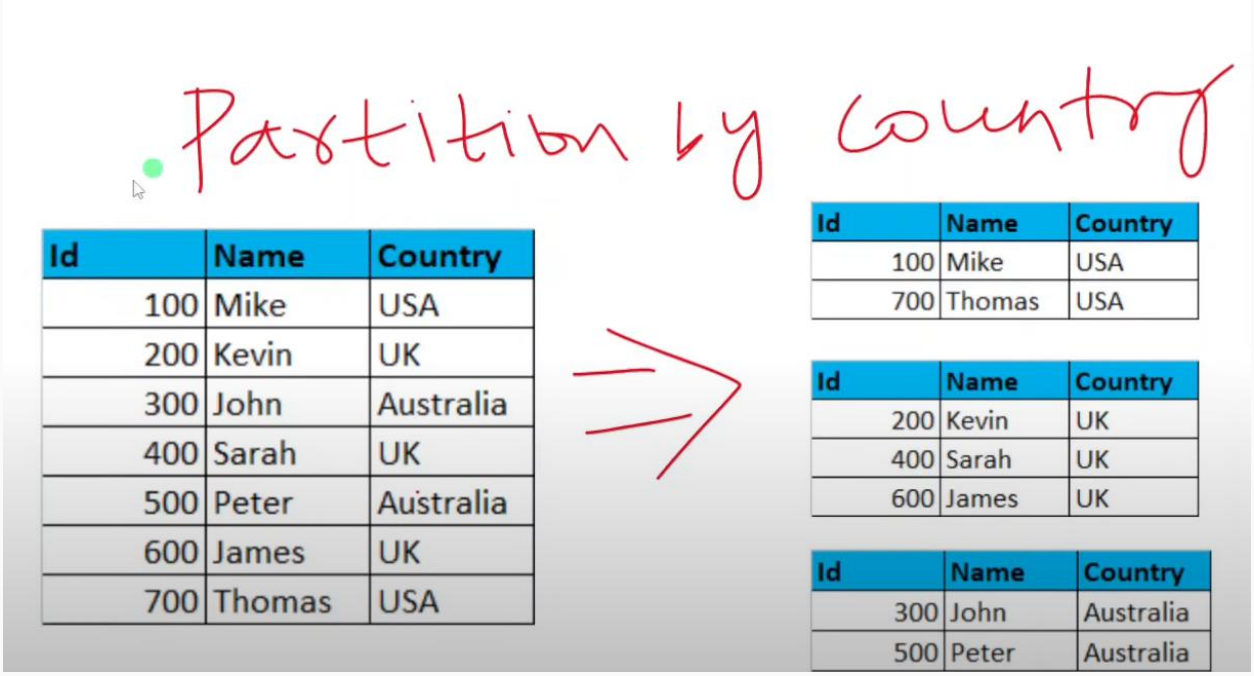


Coming to explain plan as shown above , its first reading 1 data frame deptDF, then shuffle happened and then sort. Coming to 2nd data frame empDF, it read 2nd data frame, then exchange i.e; data shuffle and then sorting the data. Once that is done, in the final stage , it is merging ,sort merge join will be applied based on dept_id.so this is about explain plan.

PERFORMANCE OPTIMIZATION- BUCKETING

Bucketing is splitting large dataset into multiple buckets based on certain key. Hash value would be generated on the key column. Based on that data would be sorted and it will be put into particular bucket. Then when we are performing wide transformation on the bucketed data , it will improve the performance. How it improves the performance? In data bricks, when we apply certain wide transformation then it will go through multiple phases such as shuffling the data and sorting. Shuffling and sorting these are costly operation in spark development. Let’s say when we are joining 2 dataframes then what happens internally is sort merge join. When we are joining 2 data frames,

basically it is done through sort merge join. In the sort merge join, there are 3 phases.in the first phase, data would be shuffled across executors or across worker nodes then sorting phase will happen. On top of shuffled data, sorting will be happen based on certain key then finally merge operation will happen on sorted and shuffled data then merge can efficiently join 2 datasets. Let's say we are doing join on dataframe and many actions are depending on that transformation. As per spark architecture, it works on lazy evaluation which means any transformation will not be executed immediately. Whenever we call certain action then dependent transformations would be executed one by one and finally action would be executed. Let's say we have a join-wide transformation, several actions are depending on the join. So, when we are calling those several actions each and every time as per lazy evaluation this join will be evaluated which means each and every time this sorting and shuffling will happen. So this is costly operation and its happening multiple times and it will hit the performance. In order to avoid that, we can shuffle the data and sort it once and store it in storage.Then in that way, we are going to preshuffle and presort the data and store it in storage location.then in the subsequent process, how many actions we are going to call, how many times we have to perform a join then there wont be any shuffle or sort because data is already preshuffled and presorted.so this is how entire execution will be benefitted.



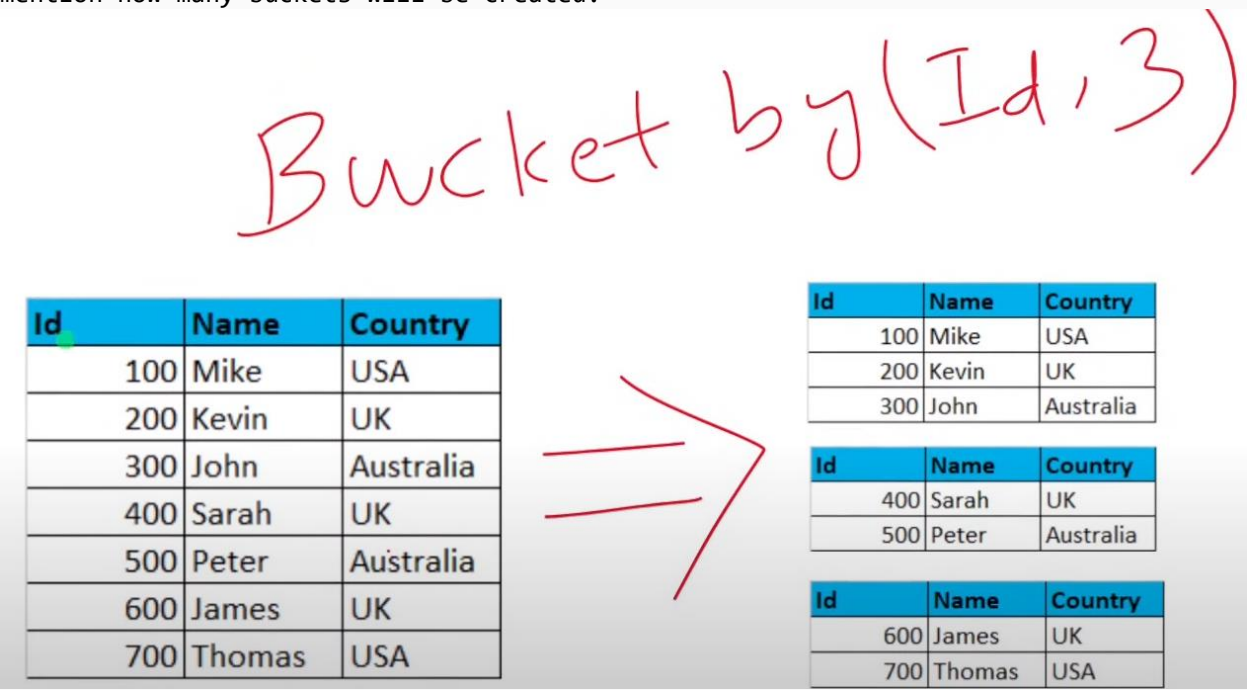
Let's consider this as a dataframe with cols Id,Name and Country. Id is primary key.it cannot be duplicated. All are unique. Coming to country, we can see there are 3 distinct countries. Now we want to partition By country.When we apply partition based on country then this is the output we are going to get as shown above.

whenever we are creating partition on particular column, it is going to create one folder for each unique partition key.

In our example, we partitioned on country so its creating 3 folders. It will create one folder for USA, one folder for UK , one folder for Australia. Under that its going to create multiple partition files. Let's say in this case,we are having only 2 records but let us assume we are going to have billions of records let's say 1GB of data then which means by default block size is 128MB. For 1GB of data, it will create 8 partitions.so it will create 8 partitions part1, part2 …… it will create 8 partitions. so, this is how partition works.so in some cases let's say we are having 2 unique keys. One key is having billions of records , another key is having only 1 record. Even in this case its going to create one separate folder for that one record also. Within that it will create one partition file.so this is how partition works. Within one particular partition folder, we can see data only for that particular partition key.it cannot mix. Partition is suitable when we have to split the data, we have to partition the data based on certain key where column is having low cardinality which means we are having less no of unique keys.

When we have to split data based on id then its going to create seven partitions because we are having seven unique values. So, it's going to create 7 folders. Under that it will create partition files by containing 1 record. So that is not suitable because in this example we have only 7 records but in the real time we will have millions or billions of records then it is going to create millions or billions of separate folders.so that is also going to be costly operation.

Partition By is always suitable for the column only for the situation where we have to split based on certain column which is having low cardinality. If column is having high cardinality, its always better to go with bucket By.In bucket By basically its not considering one particular key . It is considering range of the key. For example, for the same data we have to apply bucket By based on id column and also we have to mention how many buckets will be created.



For the same data, we will create buckets based on id column and we have to give number of buckets. Number of buckets giving as 3. So, it will split the data as shown above.Basically 3 records it will create as one partition.its not first 3 records, basically on the column which we have to create bucket By. Then hash key will be created for each key.For example hash value created for Id 100,even for 200 etc for all Id values. Then based on hash key value, it will decide which record should be part of which partition.so in this example, first 3 records are going to one bucket then next 2 records are going to another bucket and last 2 records are going to another bucket.so it is actually shuffling the data then sorting the data based on hash key.Its keeping relevant data into buckets and also its not going to create separate folders for each bucket like partition.Here it will create only 1 folder.under 1 folder, it will keep 3 files 1 ,2,3.

Whenever we are comparing partition by and bucket By, let's say we are having 10 partitions then which means we would have 10 different folders. And we are applying partition based on a column then what happens is let's say we gave 3 number of partition then for each partition folder it is going to create 3 buckets i.e; 3 bucket file which means if we are having 10 partitions and we are creating 3 buckets then it means it is going to create 30 files.

CODE:

```
#Check If Bucketing Enabled
spark.conf.get("spark.sql.sources.bucketing.enabled")
```

Output: 'true'
By default bucketing will be enabled.

```
#Create Sample Data for Demo
from pyspark.sql.functions import col,rand
df=
spark.range(1,10000,1,10).select(col("id").alias("PK"),rand(10).alias("Attribute"))
df.display()
```

We want to create 10000 records so we will create some dummy data using range function. 3rd parameter in range function is 1 which means interval between 1 value to another value is 1 , increment by 1. 4th parameter in range is it will tell us how many partitions we have to create within spark memory.here we mentioned 10 which means 10 partitions will be created. If we are going to write this into some storage , it will create 10 separate files for this output

	PK	Attribute
1	1	0.1709497137955568
2	2	0.8051143958005459
3	3	0.5775925576589018
4	4	0.9476047869880925
5	5	0.2093704977577
6	6	0.36664222617947817
7	7	0.8078688178371882

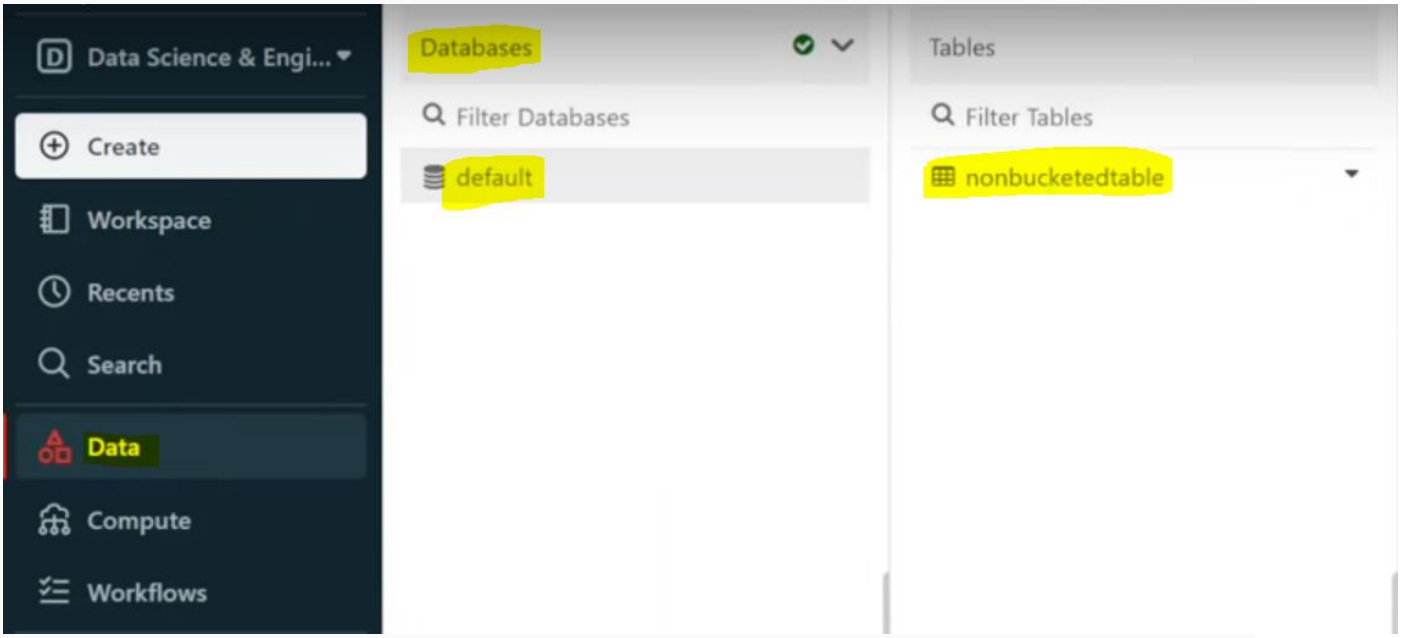
Truncated results, showing first 1000 rows.

```
df.count()
Out: 9999
```

#Create Non-Bucketed Table

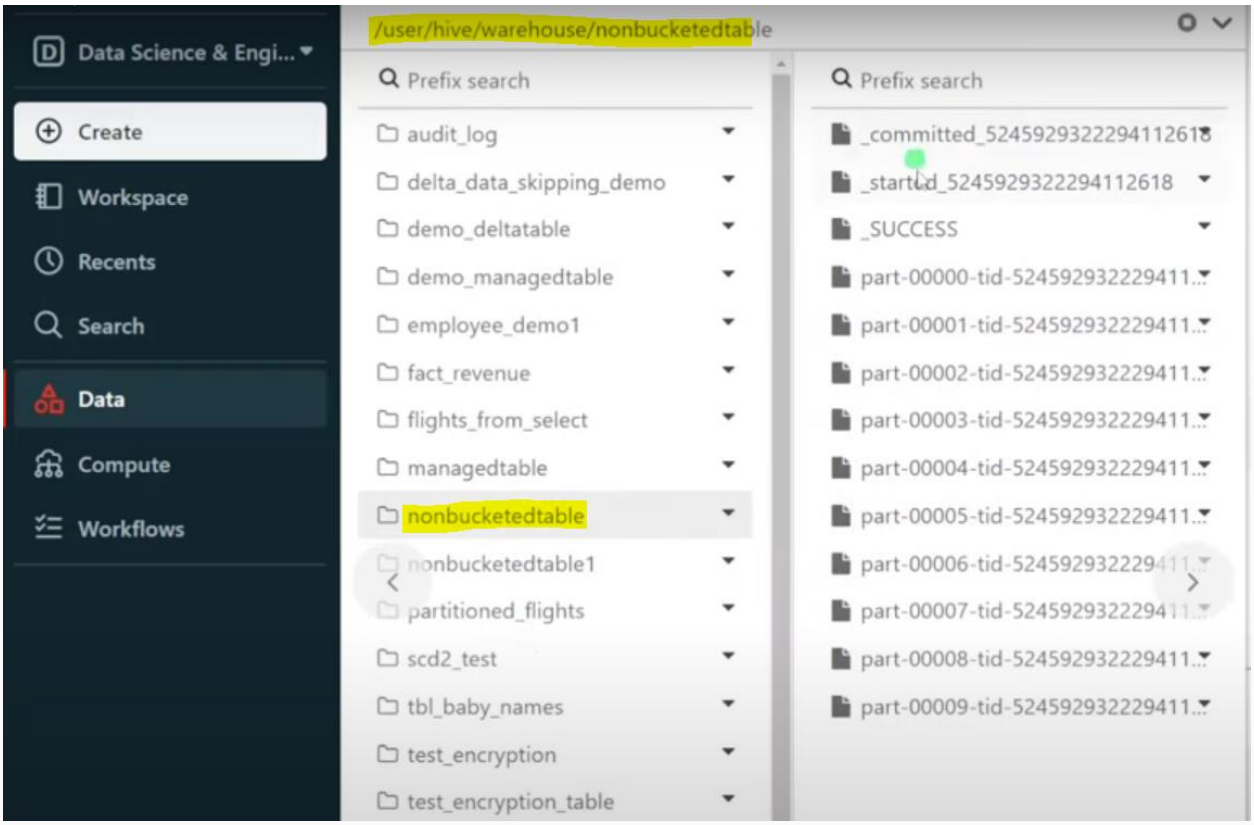
df.write.format("parquet").saveAsTable("nonbucketedTable")

Now non bucketed table was created in the default database as shown below.



At the same time if we get into location where it has stored, By default spark would store data into below folder as shown below.

Here in this folder, we are having 10 different files. Now we have created 10 different 10 partition. As a result, it has created 10 different partition files.

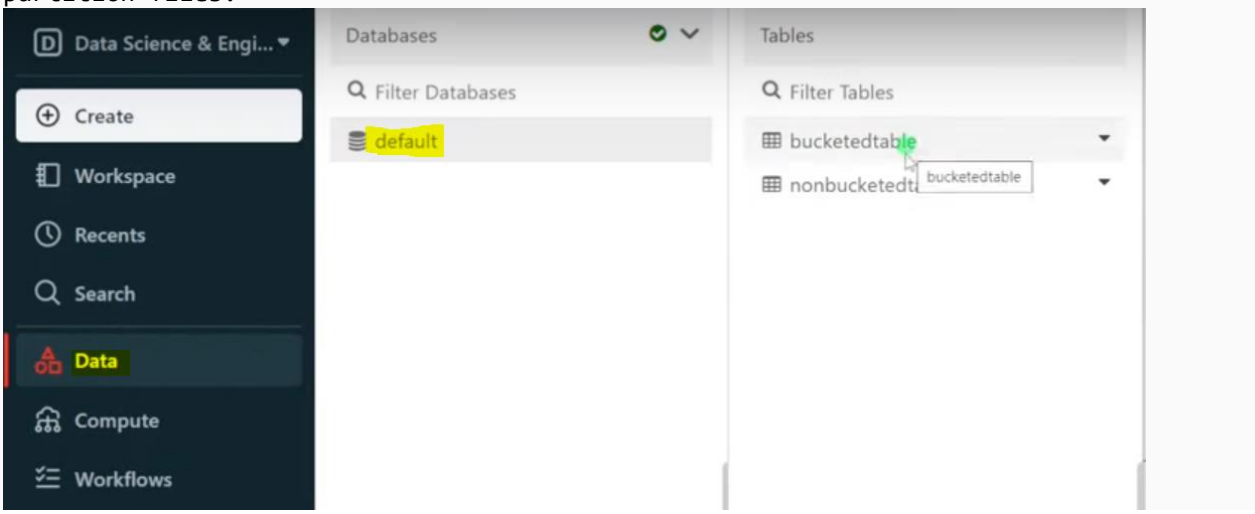


Now we will apply bucketBy for the same data and we will create Bucket Table. Now in bucketBy we will pass how many buckets we need to create and based on which key, we need to generate hash key i.e.; based on which key we need to sort the data. This is the syntax of bucketBy.

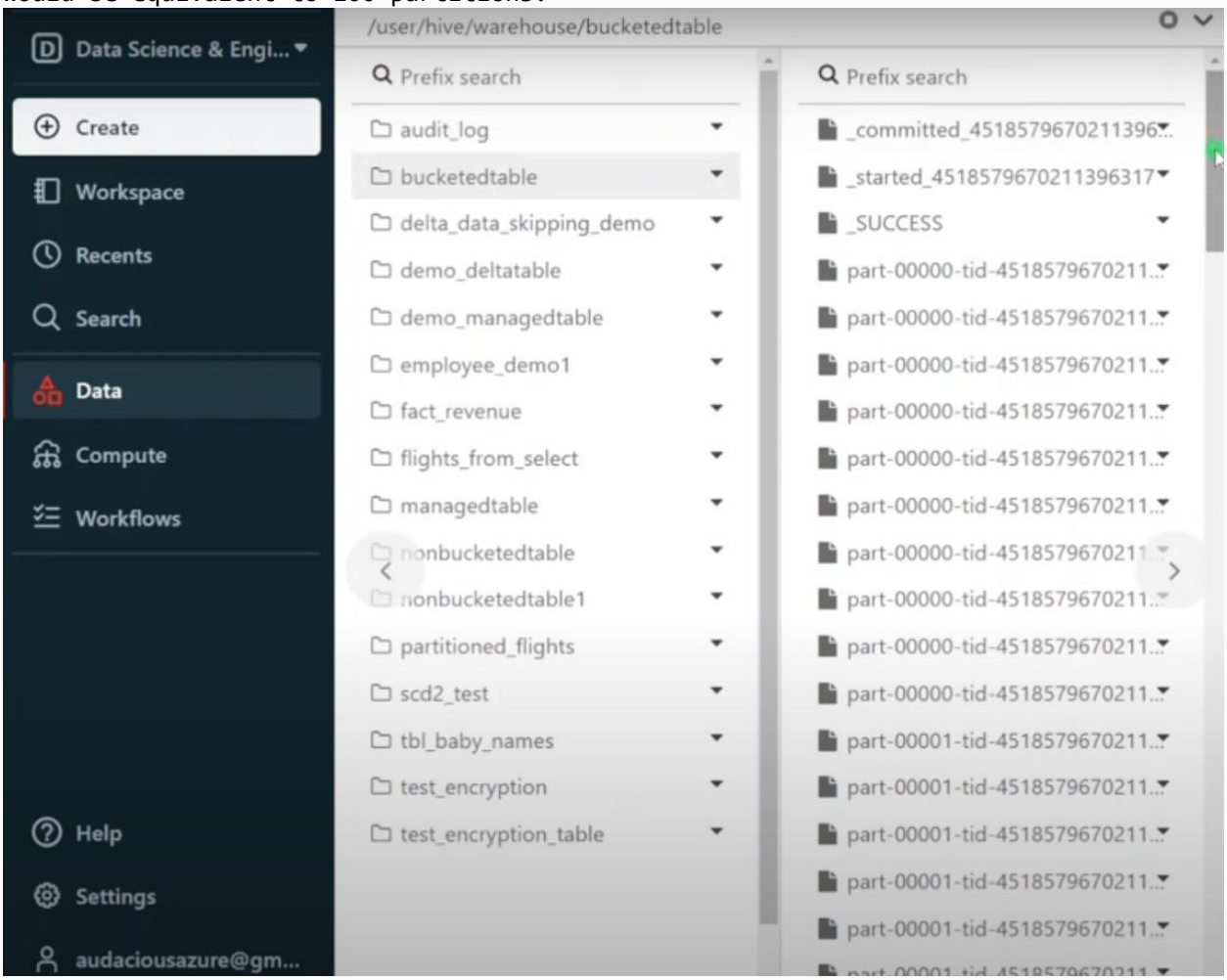
#Create Bucketed Table

df.write.format("parquet").bucketBy(10,"PK").saveAsTable("bucketedTable")

Here our data frame is already partitioned by 10. On top of that we are giving bucket By 10. Basically it will create 10 multiplied by 10, its going to create 100 data files partition files.



Now if we go to dbfs location, /user/hive/warehouse/bucketedtable , we can see there are many number of part files. There are 100 files. So, for each partition it has created 10 buckets. Overall, we are having 10 partitions so 10 buckets per partition would be equivalent to 100 partitions.



Now we have created 2 tables, one is non-bucketed where we haven't created any buckets under partition. So, there are 10 partition files. Coming to next one, we have created bucket. So basically, under each partition, we have created 10 buckets.

Now we will perform some wide transformation which would require data shuffling across executors. So, we are going to create 2 data frame based on the bucketed table and 2nd data frame on the nonbucketed table. On the same dataset, we are going to create 4 different dataframes but in real time scenario, we might have different data for joining may be one fact and dimension or may be fact to fact , it could be anything. In this scenario, it's the same table same data but one table we have created bucket one and another one is nonbucketed. So we are creating 2 dataframes df1 and df2 which are coming from bucketed table and df3 and df4 which are coming from non-bucketed table.

```
df1= spark.table("bucketedTable")
```

```
df2=spark.table("bucketedTable")
df3=spark.table("nonbucketedTable")
df4= spark.table("nonbucketedTable")
```

Now we need to see how it will works internally.So it is going to improve or keep the performance? So, for that we are going to join df3 with df4 based on the primary key column PK then by default it will be inner join.

In spark, in joining if one of the data set is going to be less than 10 MB then by default, it will be broadcast join by spark.

```
df3.join(df4,"PK","inner")
```

if we want to see explain plan for above one,

```
df3.join(df4,"PK","inner").explain()
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Project [PK#164L, Attribute#165, Attribute#171]
   +- BroadcastHashJoin [PK#164L], [PK#170L], Inner, BuildRight, false
      :- Filter isnotnull(PK#164L)
      :  +- FileScan parquet default.nonbucketedtable[PK#164L,Attribute#165] Batched: true, DataFilters: [isnotnull(PK#164L)], Format: Parquet, Location: InMemoryFileIndex(1 paths)[dbfs:/user/hive/warehouse/nonbucketedtable], PartitionFilters: [], PushedFilters: [IsNotNull(PK)], ReadSchema: struct<PK:bigint,Attribute:double>
      +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]),false), [id=#247]
         +- Filter isnotnull(PK#170L)
            +- FileScan parquet default.nonbucketedtable[PK#170L,Attribute#171] Batched: true, DataFilters: [isnotnull(PK#170L)], Format: Parquet, Location: InMemoryFileIndex(1 paths)[dbfs:/user/hive/warehouse/nonbucketedtable], PartitionFilters: [], PushedFilters: [IsNotNull(PK)], ReadSchema: struct<PK:bigint,Attribute:double>
```

Here we can see from explain plan that it is using BroadcastExchange, which means broadcast join which is going to improve performance.

Now we will disable broadcast join by using below command. we are giving threshold as -1 so that it will disable.

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
spark.conf.set("spark.sql.adaptive.enabled", False)
```

Now we will run same join query as we are getting data from non-bucketed table.

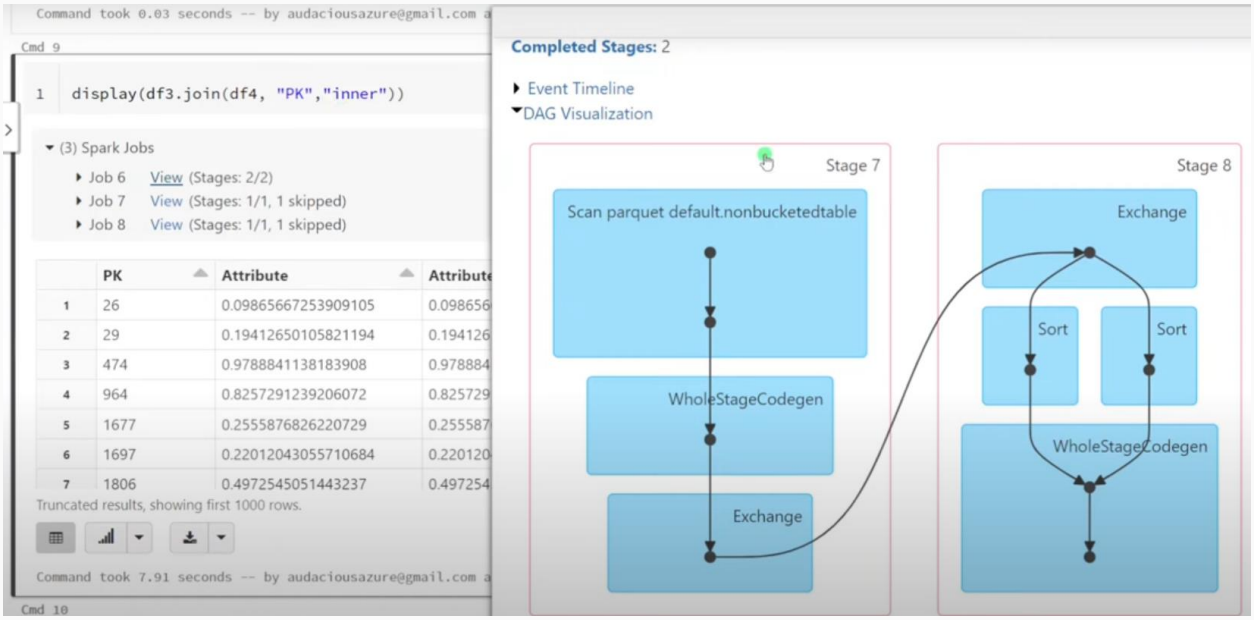
```
display(df3.join(df4,"PK","inner"))
```

	PK	Attribute	Attribute
1	26	0.09865667253909105	0.09865667253909105
2	29	0.19412650105821194	0.19412650105821194
3	474	0.9788841138183908	0.9788841138183908
4	964	0.8257291239206072	0.8257291239206072
5	1677	0.2555876826220729	0.2555876826220729
6	1697	0.22012043055710684	0.22012043055710684
7	1806	0.4972545051443237	0.4972545051443237

Truncated results, showing first 1000 rows.

If we click on job6 view, we can see that from spark UI, in stages, it is scanning nonbucketed table then after that it is performing exchange . we are using same dataset

that's the reason it has created 1 stage. If we use 2 different datasets then it would have created one more stage, that is for reading scanning another dataset. Once this nonbucketed table is scanned and it has shuffled which is nothing but exchange then finally it is being sorted, there are 2 dataframes both are getting sorted, then it would be merged. This is how it is operating inside then finally getting output for us.



#Non-bucketed to non-bucketed join. Both sides would be shuffled
Now we have turned off broadcast join.
Now we will see explain plan for same query.
df3.join(df4,"PK","inner").explain()

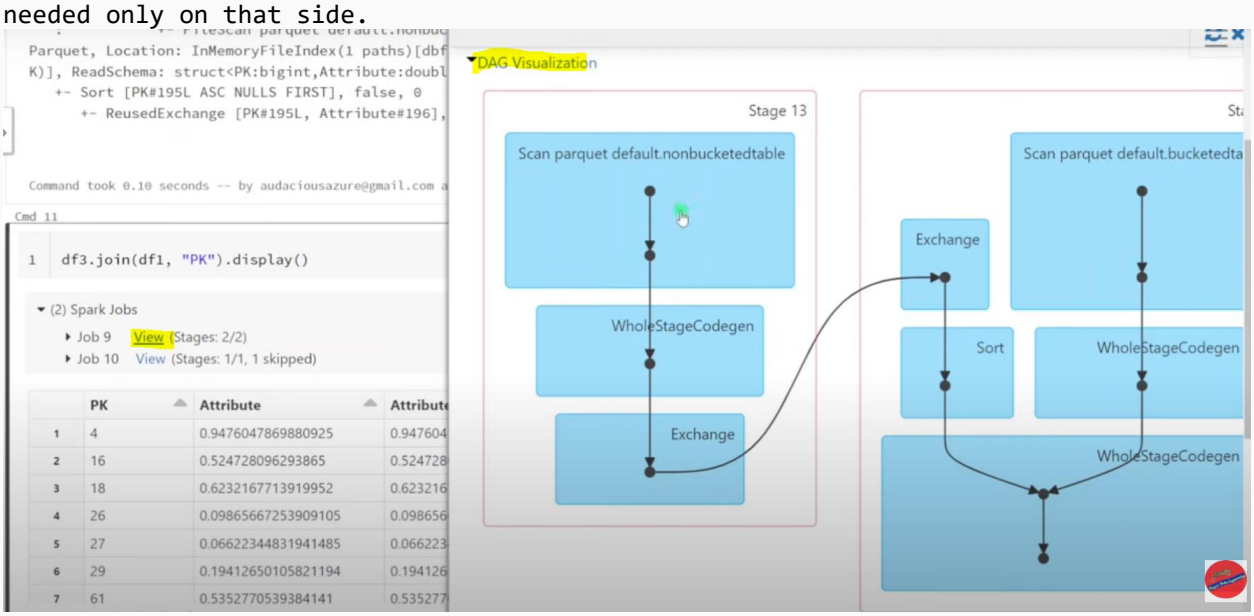
```
== Physical Plan ==
*(3) Project [PK#164L, Attribute#165, Attribute#196]
+- *(3) SortMergeJoin [PK#164L], [PK#195L], Inner
   :- Sort [PK#164L ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(PK#164L, 200), ENSURE_REQUIREMENTS, [id=#531]
   :     +- *(1) Filter isnotnull(PK#164L)
   :        +- *(1) ColumnarToRow
   :           +- FileScan parquet default.nonbucketedtable[PK#164L,Attribute#165] Batched: true, DataFilters: [isnotnull(PK#164L)], Format:
Parquet, Location: InMemoryFileIndex(1 paths)[dbfs:/user/hive/warehouse/nonbucketedtable], PartitionFilters: [], PushedFilters: [IsNotNull(P
K)], ReadSchema: struct<PK:bigint,Attribute:double>
   +- Sort [PK#195L ASC NULLS FIRST], false, 0
      +- ReusedExchange [PK#195L, Attribute#196], Exchange hashpartitioning(PK#164L, 200), ENSURE_REQUIREMENTS, [id=#531]
```

Here for 1st dataframe , it is scanning the data then it is exchanging ,shuffling the data then after that it is sorting and it is reusing here same dataset so that's why ReusedExchange and here also it is performing exchange.
Now 2 dataframes has done and then top of that it is performing merge operation that is sort merge join. This is what happening when we join 2 non bucketing tables in spark.

#Nonbucketed table to bucketed table. One side would be shuffled.

df3.join(df1,"PK").display()

we will get same output but we can see differences in spark UI. This time it will create only 2 jobs because only 1 table df3 nonbucketed table there will be exchange needed only on that side.



Here we can see non bucketed table that is scanning and also it is exchanging. Other one is bucketed table, it is not exchanging, it is directly scanning because this bucketed table is already preshuffled and presorted. That's the reason simply it is scanning the data.It's not performing shuffle exchange. Its not performing sort.But coming to nonbucketed table, it is performing the shuffle and then top of that it is performing sort and then based on the result it is merging .

Here in stage 13, it is performing extra operations like shuffle and sort so it is going to consume more time . Actually, this is coming from nonbucketed table df3 and stage 14 is coming from df1 which is already preshuffled and presorted .

Now if we see explain plan for query,

df3.join(df2,"PK").explain()

```
== Physical Plan ==
*(3) Project [PK#164L, Attribute#165, Attribute#159]
+- *(3) SortMergeJoin [PK#164L], [PK#158L], Inner
   :- Sort [PK#164L ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(PK#164L, 10), ENSURE_REQUIREMENTS, [id=#747]
   :     +- *(1) Filter isnotnull(PK#164L)
   :        +- *(1) ColumnarToRow
   :           +- FileScan parquet default.nonbucketedtable[PK#164L,Attribute#165] Batched: true, DataFilters: [isnotnull(PK#164L)], Format:
Parquet, Location: InMemoryFileIndex(1 paths)[dbfs:/user/hive/warehouse/nonbucketedtable], PartitionFilters: [], PushedFilters: [IsNotNull(P
K)], ReadSchema: struct<PK:bigint,Attribute:double>
   +- *(2) Sort [PK#158L ASC NULLS FIRST], false, 0
      +- *(2) Filter isnotnull(PK#158L)
         +- *(2) ColumnarToRow
            +- FileScan parquet default.bucketedtable[PK#158L,Attribute#159] Batched: true, Bucketed: true, DataFilters: [isnotnull(PK#158
L)], Format: Parquet, Location: InMemoryFileIndex(1 paths)[dbfs:/user/hive/warehouse/bucketedtable], PartitionFilters: [], PushedFilters: [I
sNotNull(PK)], ReadSchema: struct<PK:bigint,Attribute:double>, SelectedBucketsCount: 10 out of 10
```

So, in the explain plan, we can see here the bucketed table , it is doing file scan but after that we cannot see the exchange anywhere , it is not shuffling the data but

coming to 2nd dataframe, nonbucketed table, we can see there is a exchange hash partitioning , it is exchanging then based on that it is sorting then finally it is merging. This bucketed dataframe is simple scan from the storage. Here in bucketed dataframe, we have written the data to dbfs location. In bucketed table, it is just table scan. But in nonbucketed table, we have to just perform table scan and also data shuffle, so this is the difference.

#Bucketed Table to bucketed Table join. No shuffle at both sides

df1.join(df2,"PK").display()

(2) Spark Jobs

	PK	Attribute	Attribute
1	4	0.9476047869880925	0.9476047869880925
2	16	0.524728096293865	0.524728096293865
3	18	0.6232167713919952	0.6232167713919952
4	26	0.09865667253909105	0.09865667253909105
5	27	0.06622344831941485	0.06622344831941485
6	29	0.19412650105821194	0.19412650105821194
7	61	0.5352770539384141	0.5352770539384141

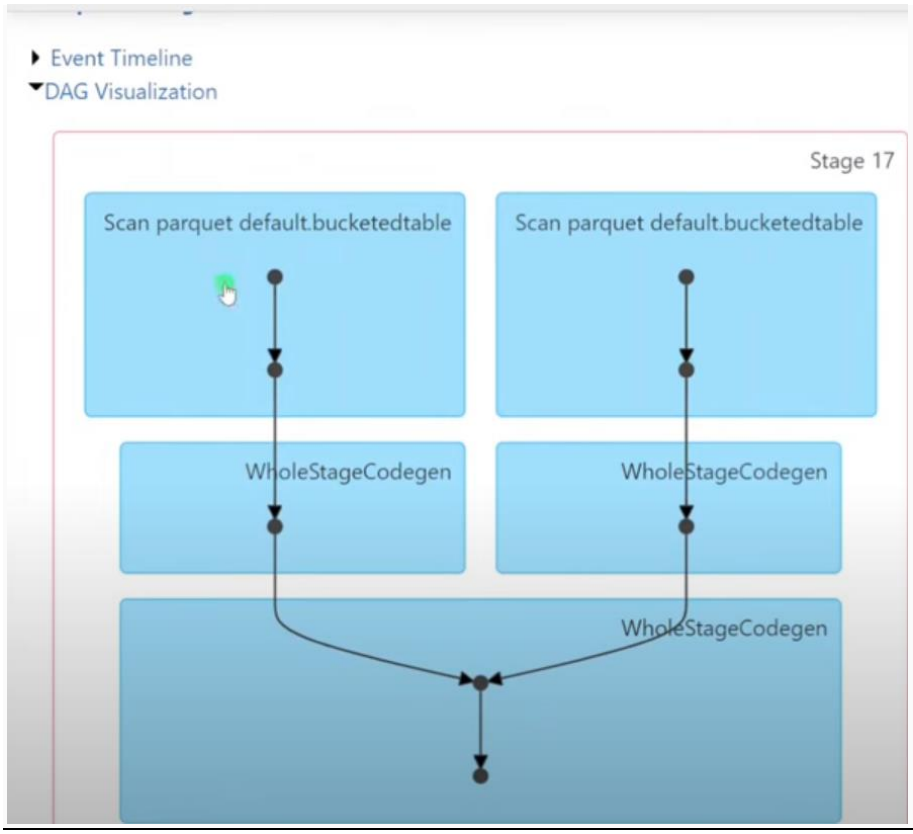
Truncated results, showing first 1000 rows.

Here we are having only 1 stage, its scanning the data bucketed one and other data frame also just scanning the bucketed one. There is no exchange and there is no sort. Simply it is scanning the data and then immediately it started merging because data is already pre-shuffled and pre-sorted. This is the advantage of bucketing.

df1.join(df2,"PK").explain()

```
== Physical Plan ==
*(3) Project [PK#158L, Attribute#159, Attribute#258]
+- *(3) SortMergeJoin [PK#158L], [PK#257L], Inner
   :- *(1) Sort [PK#158L ASC NULLS FIRST], false, 0
   :  +- *(1) Filter isnotnull(PK#158L)
   :     +- *(1) ColumnarToRow
   :        +- FileScan parquet default.bucketedtable[PK#158L,Attribute#159] Batched: true, Bucketed: true, DataFilters: [isnotnull(PK#158
L)], Format: Parquet, Location: InMemoryFileIndex(1 paths)[dbfs:/user/hive/warehouse/bucketedtable], PartitionFilters: [], PushedFilters: [I
sNotNull(PK)], ReadSchema: struct<PK:bigint,Attribute:double>, SelectedBucketsCount: 10 out of 10
   +- *(2) Sort [PK#257L ASC NULLS FIRST], false, 0
      +- *(2) Filter isnotnull(PK#257L)
         +- *(2) ColumnarToRow
            +- FileScan parquet default.bucketedtable[PK#257L,Attribute#258] Batched: true, Bucketed: true, DataFilters: [isnotnull(PK#257
L)], Format: Parquet, Location: InMemoryFileIndex(1 paths)[dbfs:/user/hive/warehouse/bucketedtable], PartitionFilters: [], PushedFilters: [I
sNotNull(PK)], ReadSchema: struct<PK:bigint,Attribute:double>, SelectedBucketsCount: 10 out of 10
```

Here in explain plan, it is scanning the first dataframe and there is no exchange. Coming to 2nd dataframe it is scanning but there is no exchange at all. Finally, it started merging using sort merge join. This is how bucketing works. Whenever we are bucketing and number of buckets are same for 2 dataframes then it will avoid shuffle and sorting so the performance will be very quicker



SCENARIO BASED INTERVIEW QUESTION

HOW TO GET ONLY MAXIMUM VALUE OF EACH COLUMN AMONG DUPLICATE RECORDS?

Let's say we are getting input file, which is containing product related data containing Product id, Name,Price,DiscountPercent.

INPUT DATA

Product id	Name	Price	DiscountPercent
100	Mobile	5000	10
100	Mobile	7000	7
200	Laptop	20000	4
200	Laptop	25000	8
200	Laptop	22000	12

Product id	Name	Price	DiscountPercent
100	Mobile	5000	10
100	Mobile	7000	7
200	Laptop	20000	4
200	Laptop	25000	8
200	Laptop	22000	12

OUTPUT DATA

Product id	Name	Price	DiscountPercent
100	Mobile	7000	10
200	Laptop	25000	12

Lets assume we have duplicate records in our dataset based on particular key. We need to remove duplicate records but at the same time need to consider the maximum value of each column among duplicate values. To develop a logic for this requirement, below are the steps

Cmd 2

Sample Data

Python

```
simpleData = ((100, "Mobile", 5000,10), \
    (100, "Mobile", 7000,7), \
    (200, "Laptop", 20000,4), \
    (200, "Laptop", 25000,8), \
    (200, "Laptop", 22000,12))

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

defSchema = StructType([ \
    StructField("Product_id",IntegerType(),False), \
    StructField("Product_name",StringType(),True), \
    StructField("Price",IntegerType(),True), \
    StructField("DiscountPercent", IntegerType(), True)
  ])

df = spark.createDataFrame(data = simpleData, schema = defSchema)
display(df)
```

	Product_id	Product_name	Price	DiscountPercent
1	100	Mobile	5000	10
2	100	Mobile	7000	7
3	200	Laptop	20000	4
4	200	Laptop	25000	8
5	200	Laptop	22000	12

Max Over Window Function

```
from pyspark.sql import Window
from pyspark.sql.functions import max,col

windowSpec = Window.partitionBy('Product_id')

dfMax=(df.withColumn('maxPrice', max('Price').over(windowSpec))
.withColumn('maxDiscountPercent', max('DiscountPercent').over(windowSpec))
# .withColumn('maxProduct_name', max('Product_name').over(windowSpec))
      )

display(dfMax)
```

	Product_id	Product_name	Price	DiscountPercent	maxPrice	maxDiscountPercent
1	100	Mobile	5000	10	7000	10
2	100	Mobile	7000	7	7000	10
3	200	Laptop	20000	4	25000	12
4	200	Laptop	25000	8	25000	12
5	200	Laptop	22000	12	25000	12

Select Max Columns

Python

```
1  dfSel =
   dfMax.select(col("Product_id"),col("Product_name"),col("maxPrice").alias("Price"),col("maxDiscountPercent").alias("DiscountPercent"))
2  display(dfSel)
```

▸ (2) Spark Jobs

Table +

	Product_id	Product_name	Price	DiscountPercent
1	100	Mobile	7000	10
2	100	Mobile	7000	10
3	200	Laptop	25000	12
4	200	Laptop	25000	12
5	200	Laptop	25000	12

Remove Duplicates

```
1  dfOut = dfSel.drop_duplicates()
2  display(dfOut)
```

▸ (2) Spark Jobs

Table +

	Product_id	Product_name	Price	DiscountPercent
1	100	Mobile	7000	10
2	200	Laptop	25000	12

Complete Code

```
from pyspark.sql import Window
from pyspark.sql.functions import max,col

windowSpec = Window.partitionBy('Product_id')

dfMax=(df.withColumn('maxPrice', max('Price').over(windowSpec))
.withColumn('maxDiscountPercent', max('DiscountPercent').over(windowSpec)))

dfSel =
dfMax.select(col("Product_id"),col("Product_name"),col("maxPrice").alias("Price"),col("maxDiscountPercent").alias("DiscountPercent"))

dfOut = dfSel.drop_duplicates()

display(dfOut)
```

HOW TO CONVERT DATAFRAME COLUMNS INTO DICTIONARY

Dictionary is nothing but map data type within spark data frame. We can use pyspark inbuilt function create_map.

```
#Create Sample Data Frame
from pyspark.sql.types import StructType, StructField, StringType,IntegerType
data = [(100,"Mobile",20000,10),
(200,"Laptop",85000,12),
(300,"Television",45000,8),
(400,"Monitor",7000,9),
(500,"Headset",6500,15)]

schema = StructType([
StructField('ProductID',IntegerType(),True),
StructField('ProductName', StringType(),True),
StructField('UnitPrice',IntegerType(),True),
StructField('DiscountPercent',IntegerType(),True)
])

df= spark.createDataFrame(data,schema)
df.display()
```

	ProductID	ProductName	UnitPrice	DiscountPercent
1	100	Mobile	20000	10
2	200	Laptop	85000	12
3	300	Television	45000	8
4	400	Monitor	7000	9
5	500	Headset	6500	15

```
#create_map - Convert columns to dictionary
from pyspark.sql.functions import col,lit,create_map
```

```
dfDict =
df.select(col("ProductID"),col("ProductName"),col("UnitPrice"),col("DiscountPercent")
,create_map(col("ProductName"),col("UnitPrice")).alias("PriceDict"))

dfDict.display()
```

Table +

	ProductID	ProductName	UnitPrice	DiscountPercent	PriceDict
1	100	Mobile	20000	10	{"Mobile": 20000}
2	200	Laptop	85000	12	{"Laptop": 85000}
3	300	Television	45000	8	{"Television": 45000}
4	400	Monitor	7000	9	{"Monitor": 7000}
5	500	Headset	6500	15	{"Headset": 6500}

Showing all 5 rows. | 0.97 seconds runtime

Or

```
from pyspark.sql.functions import col,lit,create_map
dfDict= df.withColumn("PriceDict",
create_map(lit("ProductName"),col("ProductName"),lit("UnitPrice"),col("UnitPrice")))

dfDict.display()
```

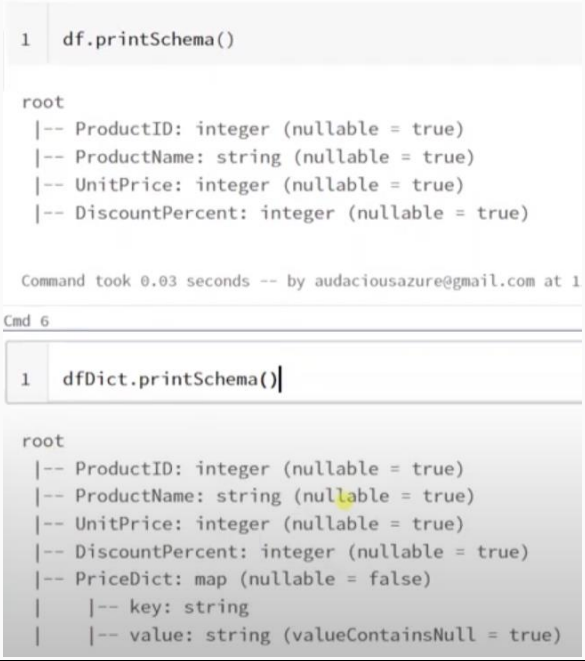
	ProductID	ProductName	UnitPrice	DiscountPercent	PriceDict
1	100	Mobile	20000	10	{"ProductName": "Mobile", "UnitPrice": "20000"}
2	200	Laptop	85000	12	{"ProductName": "Laptop", "UnitPrice": "85000"}
3	300	Television	45000	8	{"ProductName": "Television", "UnitPrice": "45000"}
4	400	Monitor	7000	9	{"ProductName": "Monitor", "UnitPrice": "7000"}
5	500	Headset	6500	15	{"ProductName": "Headset", "UnitPrice": "6500"}

Or

```
from pyspark.sql.functions import col,lit,create_map
dfDict= df.withColumn("PriceDict",
create_map(lit("ProductName"),col("ProductName"),lit("UnitPrice"),col("UnitPrice"),li
t("DiscountPercent"),col("DiscountPercent")))

dfDict.display()
```

In real time, create_map is used on multiple columns.

	ProductID	ProductName	UnitPrice	DiscountPercent	PriceDict
1	100	Mobile	20000	10	{"ProductName": "Mobile", "UnitPrice": "20000", "DiscountPercent": "10"}
2	200	Laptop	85000	12	{"ProductName": "Laptop", "UnitPrice": "85000", "DiscountPercent": "12"}
3	300	Television	45000	8	{"ProductName": "Television", "UnitPrice": "45000", "DiscountPercent": "8"}
4	400	Monitor	7000	9	{"ProductName": "Monitor", "UnitPrice": "7000", "DiscountPercent": "9"}
5	500	Headset	6500	15	{"ProductName": "Headset", "UnitPrice": "6500", "DiscountPercent": "15"}



PERFORMANCE OPTIMIZATION- DELTA CACHE

SPARK CACHE vs DELTA CACHE:

Delta cache is different from spark cache.Delta cache is one of the caching mechanism used by delta engine to speed up the data retrieval from delta tables. This delta caching is performed at hard disk or local nodes of the cluster whereas spark caching is performed at memory of local nodes of cluster. This is main difference between spark cache and delta cache. Delta cache was called as DBIO cache initially and called as delta cache later. But it is again renamed to disk cache.

When we enable delta caching, for any of delta table what happens internally?

When we create delta table, the data is distributed and stored across remote location, let it be ADLS file system or DBFS file system or S3 bucket. When we perform data analytics later on the same delta table, data would be pulled from different remote locations to serve the customer needs. As the data is fetched from remote location it will have impact and the performance. So, in order to improve data retrieval, we can enable delta cache for that table. When delta cache is enabled, when a partition file is pulled from remote server it will be saved in the local node of the cluster. So, when a particular query needs the same partition again for another query in the future delta engine doesn't need to retrieve partition once again from remote location. Instead of that , it can reuse locally persisted partition thus improving the performance. Let's take an example of sales data which is containing hundreds of partition files stored across multiple remote locations. User A submits one of the SQL query to retrieve data from sales table. That query output is depending on 10 partition out of 100 partition. So, the delta engine will pull those 10 partition for the output from remote servers. At the same time , it will persist those 10 partitions into disk of local nodes for future reference. In the future any other query needs same partition

for its output, it doesn't need to pull the data once again from the remote server. Instead of that, it will read data from the local disk. Important note here is delta cache is not performed for all the partitions as soon as delta table is created.it will cache all the partitions when the data is read from delta table for the first time after enabling the delta cache. Also, it will cache only the partition which are involved in that read but not all the partitions. In order to preload the entire table into disk of the local nodes, we can execute query like cache select * from table.
Delta cache is more efficient than spark cache because delta cache uses efficient decompression algorithm and optimal data format compared to spark cache.
How to enable disk cache?

```
spark.conf.set("spark.databricks.io.cache.enabled","[true|false]")
```

In order to disable delta cache, we can set this property to false. But it does not mean that it will remove the already persisted files from the local disk. Instead, it will just stop persisting any more files.

CODE Source Data:

```
from pyspark.sql.types import *
csvSchema = StructType([
StructField("Year",IntegerType()),
StructField("First_Name",StringType()),
StructField("Country",StringType()),
StructField("Gender",StringType()),
StructField("Count",IntegerType())
])
df=
spark.read.option("sep",",").option("header",True).schema(csvSchema).csv("/FileStore/tables/bigdata_training/read_files/Baby_Names.csv")
df.display()

#Create Delta Table
df.write.format("delta").saveAsTable("names")
```

we haven't mentioned database here. This table will be created under default database.

```
#Enable Delta Cache
```

As we are planning to use delta cache mechanism in our development, we need to enable that spark setting to true.

```
spark.conf.set("spark.databricks.io.cache.enabled","true")
#checking whether delta cache is enabled or not
spark.conf.get("spark.databricks.io.cache.enabled")
Out: 'true'

#select comes from DBFS
%sql
```

SELECT * from names

When we persisted data while creating delta table, data would be partitioned into multiple parquet files and it is distributed across multiple remote servers.

When we perform select statement now, the delta engine would pull the data from those remote servers as the data is pulled from remote servers that would be latency and it will hit the performance. The impact is applicable for any future data analytics queries as well. So, in order to prevent the remote load , we are planning to use delta cache. To cache the entire table, we are executing below command

#Preload and Cache Entire Delta Table into Local Disk

%sql

CACHE SELECT * from names

Out: Ok

This command will read data from the remote servers during the first read and it will persist those data files into local nodes of a cluster. As a result, any future queries will access only local nodes in stuff remote server. This will improve latency lens performance.

In the below step, we are once again reading the data from names table with where condition. Now the query will fetch data only from local nodes with performance efficient way.

#This time select comes from local disk.

%sql

SELECT * from names WHERE Year=2007

Table +

	Year	First_Name	County	Gender	Count
1	2007	ZOEY	KINGS	F	11
2	2007	ZOEY	SUFFOLK	F	6
3	2007	ZOEY	MONROE	F	6
4	2007	ZOEY	ERIE	F	9
5	2007	ZOE	ULSTER	F	5
6	2007	ZOE	WESTCHESTER	F	24
7	2007	ZOE	BRONX	F	13

HOW TO CONVERT DATAFRAME ARRAY ELEMENTS INTO SEPARATE COLUMNS ?

Let’s say we have a data frame with array column. In that array we have list of elements. Now the requirement is we have to split that elements into separate columns. Let’s say this is our data frame which has 2 columns Key and Value. Column named value is array data type which is having values 111,222 and 333.

In order to flatten this array into separate rows , we are flattening array list into separate rows, for that we can use function explode.

Key	Value
100	[111,222,333]

EXPLODE

Key	Value
100	111
100	222
100	333

Now we have array datatype which is having list of elements. Instead of converting these elements into separate rows, we have to convert into separate columns. Elements will be separated by column value1 , value 2 , value 3. s

Key	Value
100	[111,222,333]

???

Key	Value1	Value2	Value3
100	111	222	333

```
#CREATE SAMPLE DATAFRAME
df = spark.createDataFrame(sc.parallelize([['ABC',[1,2,3]],['XYZ',[2,None,4]],['KLM',[8,7]],['IJK',[5]]]),["key","value"])

df.display()
```

This sample dataframe is having data of array data type.

	key	value
1	ABC	[1, 2, 3]
2	XYZ	[2, null, 4]
3	KLM	[8, 7]
4	IJK	[5]

#Split Array Values into Separate Columns

Now we need to flatten this array elements into separate columns . By using python list indexing method, we can retrieve each and every element into separate columns

Split Array Values Into Separate Columns

```
1  df.select("key", df.value[0], df.value[1], df.value[2]).display()
```

(3) Spark Jobs

Table +

	key	value[0]	value[1]	value[2]
1	ABC	1	2	3
2	XYZ	2	null	4
3	KLM	8	7	null
4	IJK	5	null	null

We have converted list of elements into separated columns like value[0],value[1],value[2] but challenge comes when we are dealing with huge amount of data. In this example we have created only 4 rows and maximum no of elements is 3 but in real time example when we are dealing with millions of records, we might get 100's of elements. But in the real time, we might have 100's of elements with millions of rows so its not possible to calculate maximum no of elements manually so for that we need some automated mechanism. So, in order to make the process automated, first step is we need to calculate number of elements for each array so for that we are going to use size function. This will basically determine number of elements for each array input. So, for each column input, we want to determine number of elements , so we are creating new column(NoOfArrayElements).

#How to Automate this solution
Determine the Size of Each Array

```
from pyspark.sql.functions import size,col
dfSize = df.select("key","value",size("value").alias("NoOfArrayElements"))
dfSize.display()
```

	key	value	NoOfArrayElements
1	ABC	[1, 2, 3]	3
2	XYZ	[2, null, 4]	3
3	KLM	[8, 7]	2
4	IJK	[5]	1

```
#Get Maximum Size of All Arrays
max_value = dfSize.agg({"NoOfArrayElements":"max"}).collect()[0][0]
print(max_value)
Out: 3
```

Here we are using agg function max then it is going to pick max value for entire column NoOfArrayElements, in this case it is 3 and then using collect function we are going to assign that value into one of the variable max_value, now the value 3 will be assigned to max_value.

#UDF to convert Array Elements into Columns

In order to split elements into separate columns, we have created one UDF(arraySplitIntoCols). This is accepting 2 input parameters one is df and 2nd is maximum number of elements. df is our source dataframe which is having key and value columns. maxElements is maximum value that is calculated in the above step.
With in UDF, we have used for loop, so based on maximum number of elements, it will iterate those many number of times . for e.g., if we are having 3 no of elements, then it will iterate 3 times starting from 0,1,2.Incase we are having 10 elements, maximum

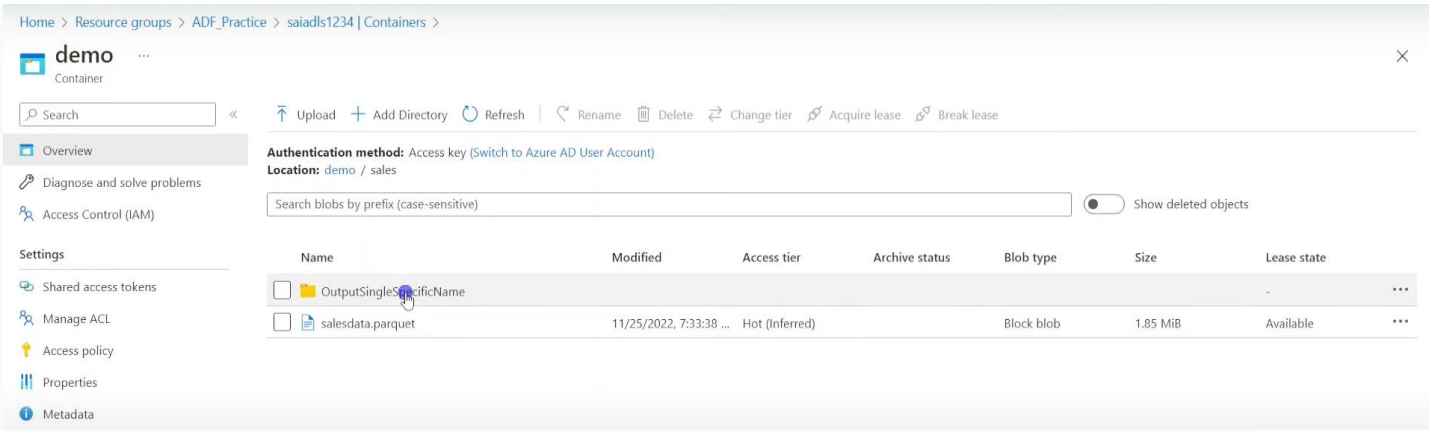
no of elements is 10 then it will iterate from 0 through 9.For each iteration, it is going to create a new column using python list indexing method. In order to make column unique we are concatenating some constant here in this case some new column then we are concatenating with index number . In this way, we are creating new columns for maximum number of elements. In this way all the elements will be splitted into separate columns.

```
def arraySplitIntoCols(df,maxElements):
  for i in range(maxElements):
     df= df.withColumn(f"new_col_{i}",df.value[i])
  return df

#UDF Call
dfout = arraySplitIntoCols(df,max_value)
display(dfout)
```

	key	value	new_col_0	new_col_1	new_col_2
1	ABC	[1, 2, 3]	1	2	3
2	XYZ	[2, null, 4]	2	null	4
3	KLM	[8, 7]	8	7	null
4	IJK	[5]	5	null	null

HOW TO WRITE DATAFRAME OUTPUT INTO SINGLE FILE WITH SPECIFIC FILE NAME?

This is quite challenging with spark environment because spark is in memory and distributed platform which means when some data is processed within spark engine, data would be splitted across multiple executors in the form of partitions. So, when we are writing certain output into target system what happens is within many executors we are having multiple cores. Those cores will be processing certain portion of the data and each and every core will be writing output into target location. As a result, we will get multiple partition files as an output under a folder.so we can only specify folder name but not the file name. Even if we want output into single file, we can apply some repartition technique repartition or coalesce. In that way we can convert multiple partitions into single partition and we can get one single output file but still getting the file with a specific file name that is quite challenging . In the spark environment, we can give only folder name but not the file name. File name will be generated automatically based on some spark standard.



There is no file in OutputSingleSpecificName folder.

Check if source file exists

```
%fs
ls /mnt/adls_test/sales
```

Table +

	path	name	size	modificationTime
1	dbfs:/mnt/adls_test/sales/OutputSingleSpecificName/	OutputSingleSpecificName/	0	0
2	dbfs:/mnt/adls_test/sales/salesdata.parquet	salesdata.parquet	1934877	1669385018000

Showing all 2 rows. | 0.98 seconds runtime

```
#Create Input Dataframe
salesDF= spark.read.format("parquet").load("/mnt/adls_test/sales/salesdata.parquet")
display(salesDF)
```

Table +

	SalesOrderLineKey	ResellerKey	CustomerKey	ProductKey	OrderDateKey	DueDateKey	ShipDateKey	SalesTerritoryKey	Order_Qu
1	43659001	676	-1	349	20170702	20170712	20170709	5	1
2	43659002	676	-1	350	20170702	20170712	20170709	5	3
3	43659003	676	-1	351	20170702	20170712	20170709	5	1
4	43659004	676	-1	344	20170702	20170712	20170709	5	1
5	43659005	676	-1	345	20170702	20170712	20170709	5	1
6	43659006	676	-1	346	20170702	20170712	20170709	5	2
7	43659007	676	-1	347	20170702	20170712	20170709	5	1

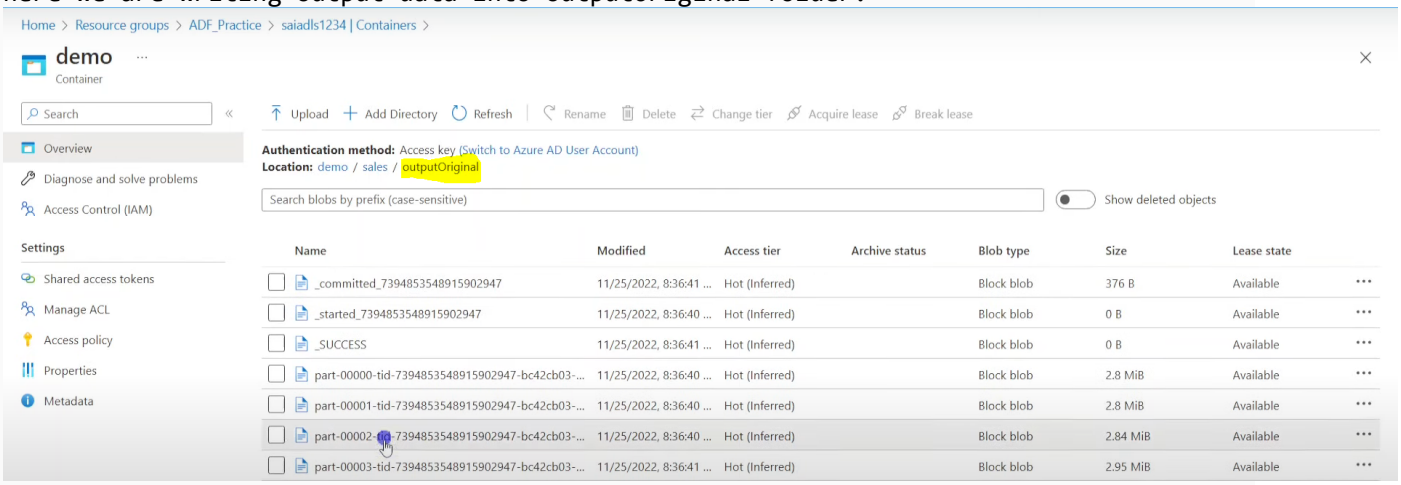
Truncated results, showing first 1,000 rows. | 0.95 seconds runtime Refreshed no

```
#Number of Records
print(salesDF.count())
op:121253

#Wide Transformation
from pyspark.sql.window import Window
```

```
from pyspark.sql.functions import row_number,lit
windowSpec =Window.partitionBy("SalesOrderLineKey").orderBy("SalesOrderLineKey")
outDF= salesDF.withColumn("row_number",row_number().over(windowSpec))
display(outDF)
```

```
#Write Dataframe into Target Location As Is
outDF.write.format("csv").option("header",True).option("sep","|").save("/mnt/adls_test/sales/outputOriginal/")
```

here we are writing output data into outputOriginal folder.



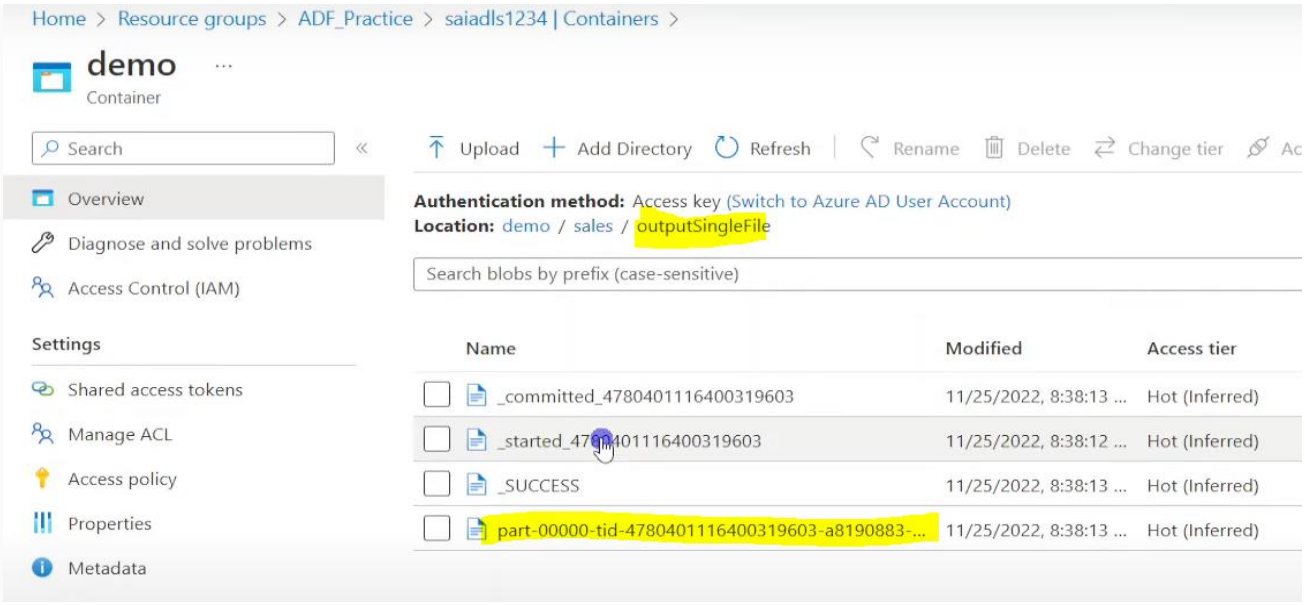
Here in screenshot, first 3 files are supporting files for spark operation. Apart from that, we are having 4 output files, these are 4 different partition files and these part files names are autogenerated by spark engine so by default spark will create many partition files for one particular dataframe.

```
%fs
ls /mnt/adls_test/sales/outputOriginal/
```

Now our requirement is to convert our output into single file so for that we will apply coalesce function. coalesce(1) means we want to convert all the partitions into 1 partition. So, this logic will ensure we are converting all the data into 1 single partition.

```
#Convert to Single Partition and Write
outDF.coalesce(1).write.format("csv").option("header",True).option("sep","|").save("/mnt/adls_test/sales/outputSingleFile/")
```

Here using above command, single partition file got created with part name as auto generated.



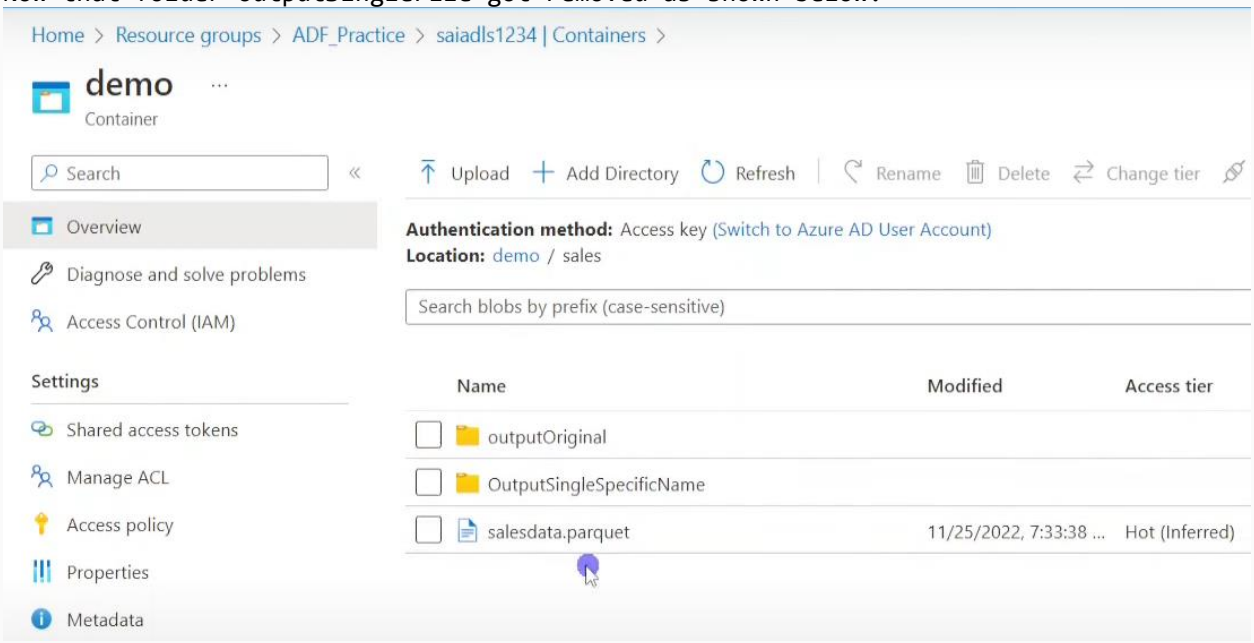
Now our requirement is we have to create single file with specific file name.

Now first we need to remove outputSingleFile so for that we are using below command.

```
%fs
rm -r /mnt/adls_test/sales/outputSingleFile/
```

```
res3: Boolean = true
```

Now that folder outputSingleFile got removed as shown below.



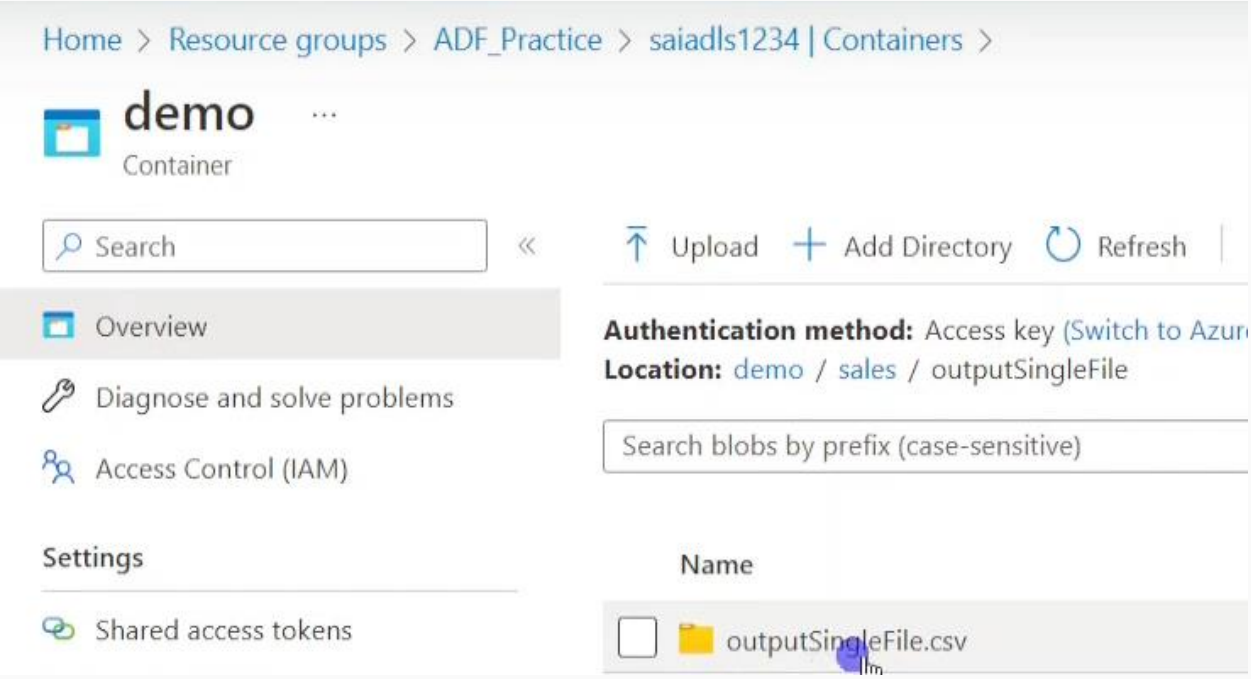

```
#Try writing with specific file name
outDF.coalesce(1).write.format("csv").option("header",True).option("sep","|").save("/mnt/adls_test/sales/outputSingleFile/outputSingleFile.csv")
```

```
%fs
ls /mnt/adls_test/sales/outputSingleFile/
```

Table +

	path	name	size	modificationTime
1	dbfs:/mnt/adls_test/sales/outputSingleFile/outputSingleFile.csv/	outputSingleFile.csv/	0	0

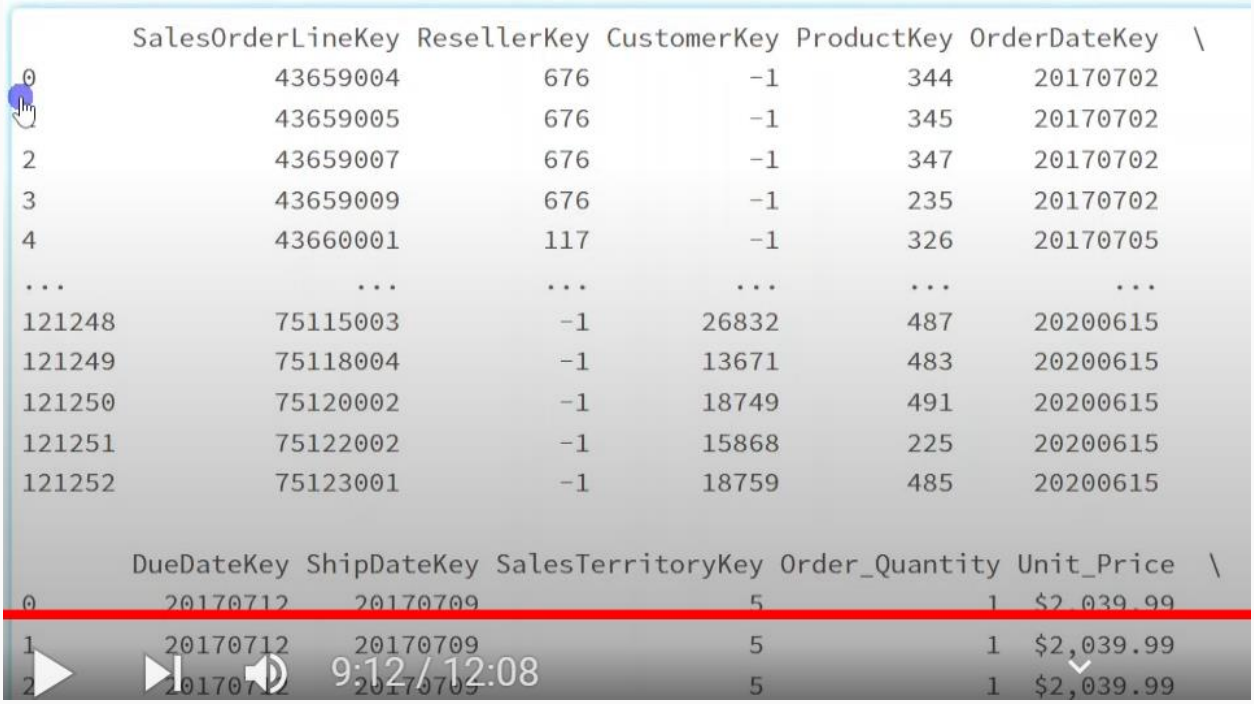
If we go to ADLS and check
In sales folder → outputSingleFile folder → inside outputSingleFile folder we will have outputSingleFile.csv folder



Within outputSingleFile.csv folder, it has created partition file with auto generated name but this is not we are looking for. We want output with specific file name. so, it included that we cannot give specific file name in spark engine. So, we are using pandas dataframe here. so, solution is we will convert our data frame into pandas dataframe. Pandas dataframe can be displayed by using print statement.

```
#Solution: Convert to Pandas Dataframe
pandasDF = outDF.toPandas()
print(pandasDF)
```

Now our data frame is converted into pandas data frame. In output, along with pandas data frame we can see extra column called index. Whenever we are converting spark data frame into pandas dataframe, it will always add one extra column that is index. If we don't want this index column in our output then we should remove while writing.



	SalesOrderLineKey	ResellerKey	CustomerKey	ProductKey	OrderDateKey	\
0	43659004	676	-1	344	20170702	
1	43659005	676	-1	345	20170702	
2	43659007	676	-1	347	20170702	
3	43659009	676	-1	235	20170702	
4	43660001	117	-1	326	20170705	
...	...	...	...	...	...	
121248	75115003	-1	26832	487	20200615	
121249	75118004	-1	13671	483	20200615	
121250	75120002	-1	18749	491	20200615	
121251	75122002	-1	15868	225	20200615	
121252	75123001	-1	18759	485	20200615	

	DueDateKey	ShipDateKey	SalesTerritoryKey	Order_Quantity	Unit_Price	\
0	20170712	20170709	5	1	$2,039.99	
1	20170712	20170709	5	1	$2,039.99	
2	20170712	20170709	5	1	$2,039.99	

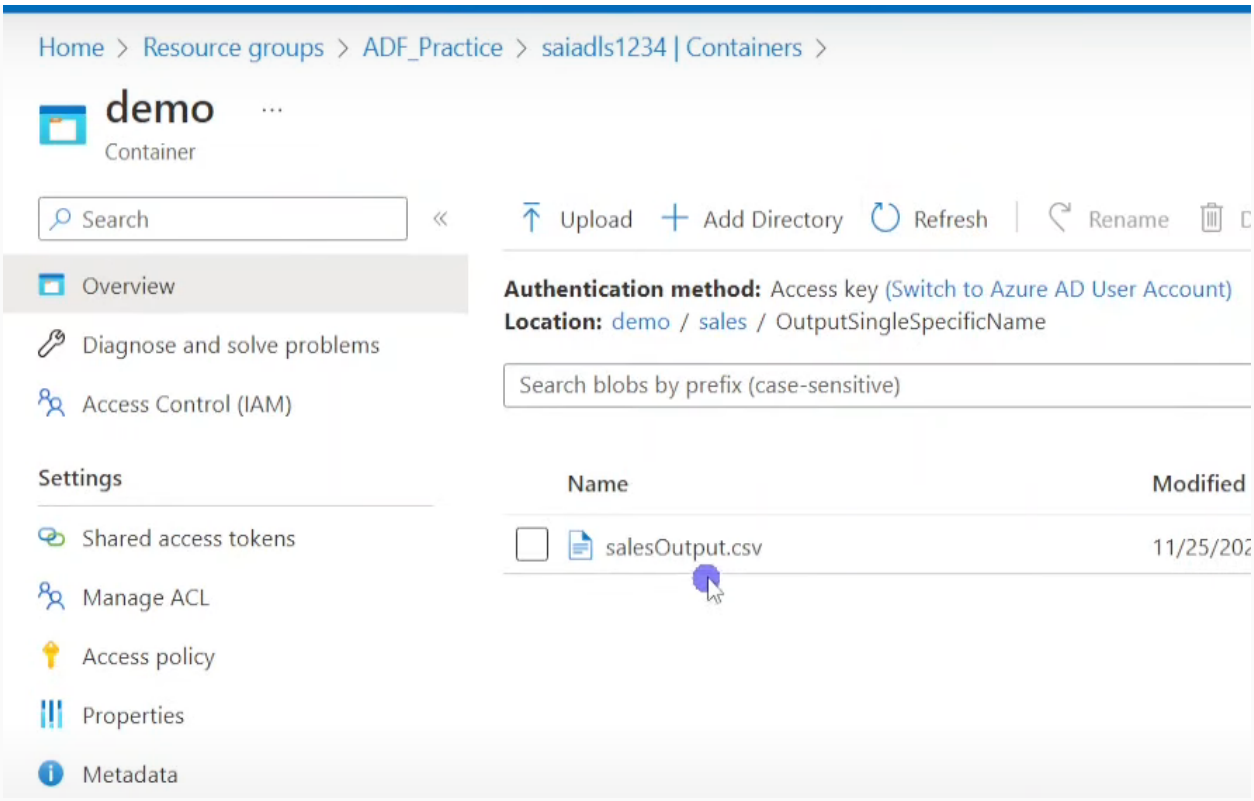
9:12 / 12:08

#listing output files we are adding under OutputSingleSpecificName
%fs
ls /mnt/adls_test/sales/OutputSingleSpecificName

Out: Ok
We don't have any files in this folder currently.
Coming to pandas, we have to create the folder manually then at the time of writing the output, pandas would expect that folder to be present. Incase if folder is not present , it will throw error that's the reason we have already created this storage location OutputSingleSpecificName

#Write pandas dataframe output with Specific File Name

pandasDF.to_csv('/dbfs/mnt/adls_test/sales/OutputSingleSpecificName/salesOutput.csv', index=False)
By default, index is true, if we don't want index we will keep index as false.
Output shown below.
And now we got output with specific file name salesOutput.csv



Now we want to see output data which was created based on pandas data frame so for that we used file system command head

%fs

head /mnt/adls_test/sales/OutputSingleSpecificName/salesOutput.csv

```
[Truncated to first 65536 bytes]
SalesOrderLineKey,ResellerKey,CustomerKey,ProductKey,OrderDateKey,DueDateKey,ShipDateKey,SalesTerritoryKey,Order_Quantity,Unit_Price,Extended_Amount,Unit_Price_Discount_Pct,Product_Standard_Cost,Total_Product_Cost,Sales_Amount,row_number
43659004,676,-1,344,20170702,20170712,20170709,5,1,"$2,039.99","$2,039.99",0.00%,"$1,912.15","$1,912.15","$2,039.99",1
43659005,676,-1,345,20170702,20170712,20170709,5,1,"$2,039.99","$2,039.99",0.00%,"$1,912.15","$1,912.15","$2,039.99",1
43659007,676,-1,347,20170702,20170712,20170709,5,1,"$2,039.99","$2,039.99",0.00%,"$1,912.15","$1,912.15","$2,039.99",1
43659009,676,-1,235,20170702,20170712,20170709,5,1,$28.84,$28.84,0.00%,$31.72,$31.72,$28.84,1
43660001,117,-1,326,20170705,20170715,20170712,5,1,$419.46,$419.46,0.00%,$413.15,$413.15,$419.46,1
43661003,442,-1,304,20170705,20170715,20170712,6,2,$714.70,"$1,429.41",0.00%,$617.03,"$1,234.06","$1,429.41",1
43661007,442,-1,348,20170705,20170715,20170712,6,3,"$2,024.99","$6,074.98",0.00%,"$1,898.09","$5,694.28","$6,074.98",1
43661010,442,-1,292,20170705,20170715,20170712,6,2,$818.70,"$1,637.40",0.00%,$706.81,"$1,413.62","$1,637.40",1
```

ACCESS CONTROL MECHANISM'S IN DATABRICKS WORKSPACE OBJECT

We should always follow minimum privilege model which means whenever we have to grant some privilege to some resource in order to perform their day-to-day activities we should only give minimum access that will suffice for them in order to perform day-to-day activities. We should never go with extra privileges even if in certain cases we need to give extra privileges we need to revoke as soon as the purpose is completed for that particular resource.

Coming to access control in databricks, first of all user should be added in Azure Active Directory. Any user we want to grant access to databricks environment databricks workspace, first of all that user should be added into Azure Active Directory. In case

we don’t have any user in Azure Active Directory then we can’t grant access to workspace. Once we have added that user in Azure Active Directory, next step is within databricks we have to create users and groups. User is nothing but one single user for whom we want to grant access . Group is nothing but combining multiple users into separate groups. Lets say in our environment we are having 20 users, 10 users are part of development team, 10 users are part of production team. In this case instead of controlling access for individual users what we can do is we can create 2 groups, one is for development we can add those 10 users in that group. Another one is production, we can add those 10 users into production group. Now whatever access we want to control for development , all the developers we can do in one place using that single group. That’s the advantage of group.

Once we have created users or groups then we can give access to those groups or users using grant access method.

Basically, in databricks there are 4 different access ,

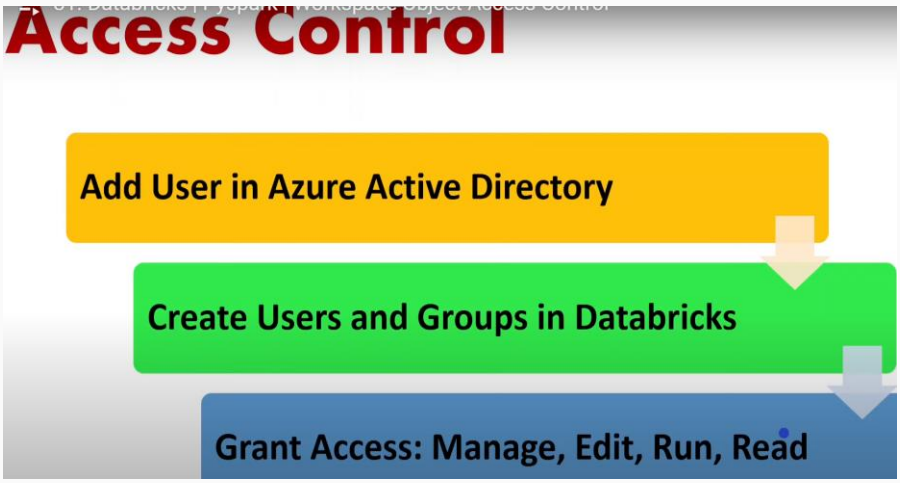
Manage: which means that user can be able to create a new notebook or edit run read and delete . its kind of full access that is a master one.

Edit: user who got edit access they can edit the notebooks even they can run the execution and also, they can read the files.

Run: They will be able to run and read the notebooks.

Read: Its only read access.

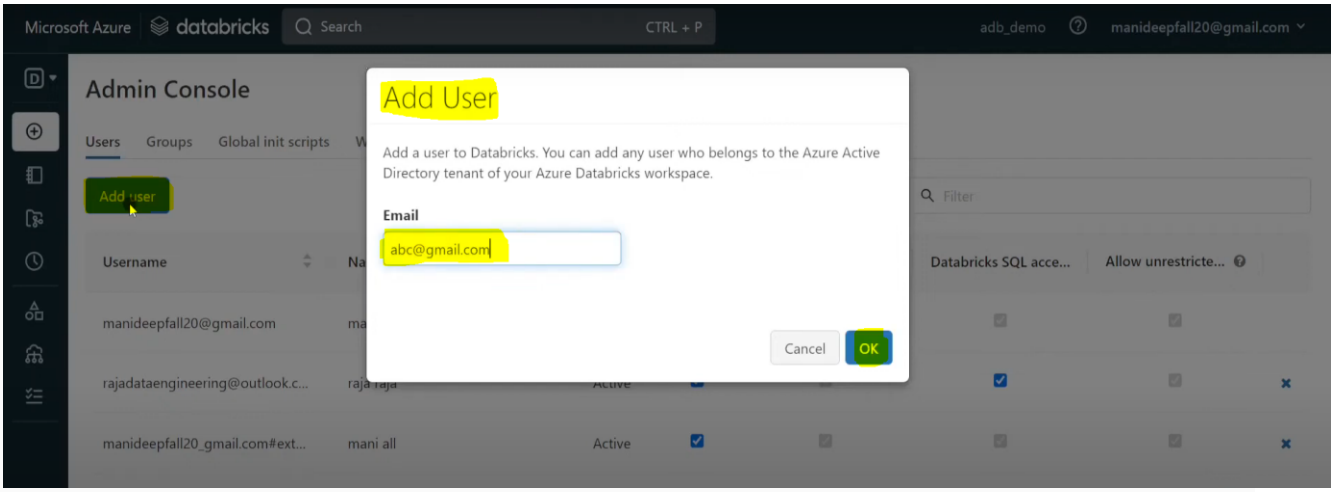
These are the different access provided in data bricks environment.



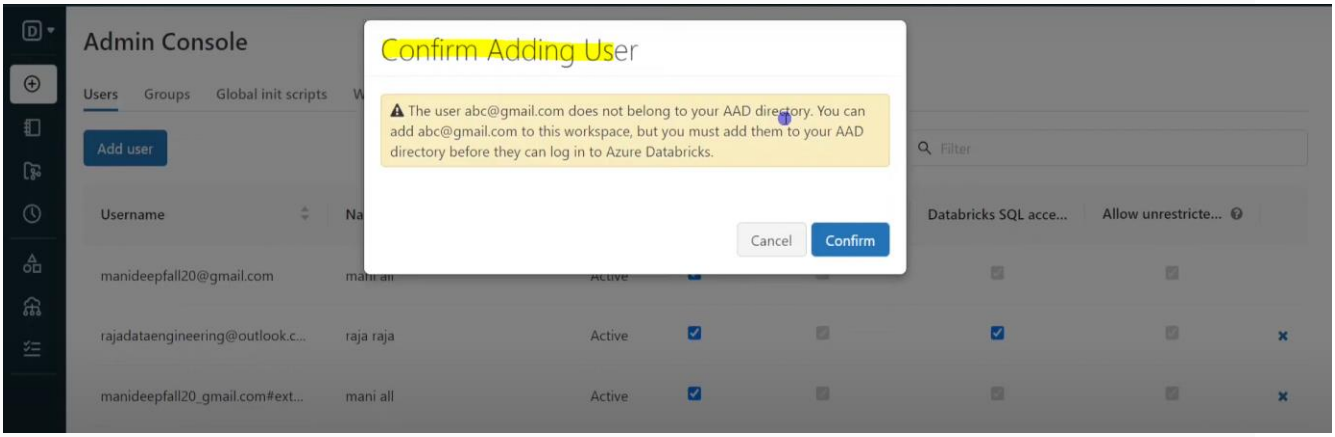
CODE

We need to create users and groups in databricks environment. In order to create users and groups, we need to go to admin console(click on mail id at top right → select Admin Console).

In Admin console currently we are having 3 users, in case if we have to add one more user, we need to add user as shown below. We are adding abc@gmail.com so that user can access our DBWS.



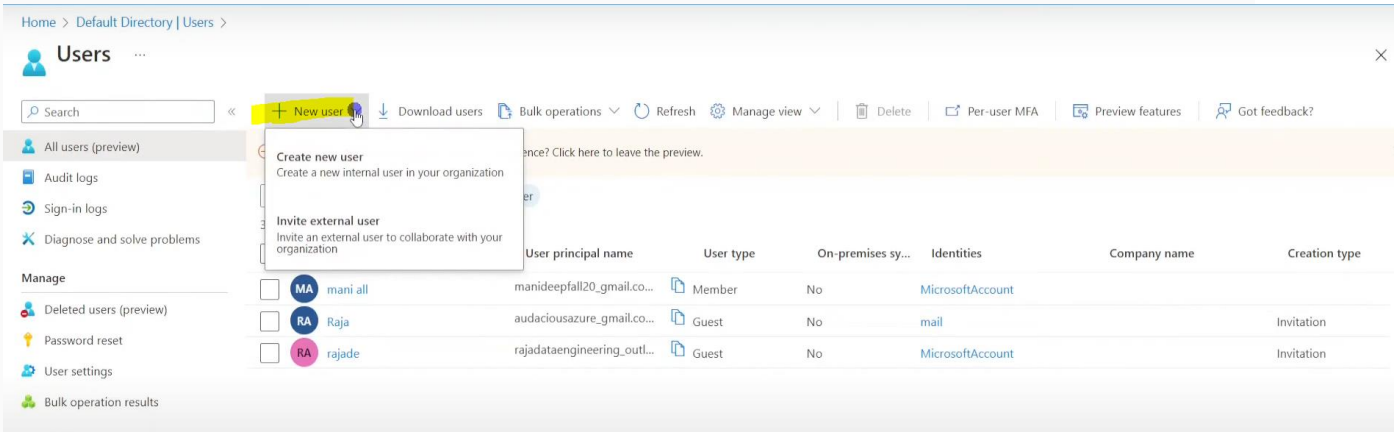
Once we click on Ok, we will get below pop up.



The above screenshot means first of all user should be added to Azure Active Directory then only we will be able to add that user to workspace .

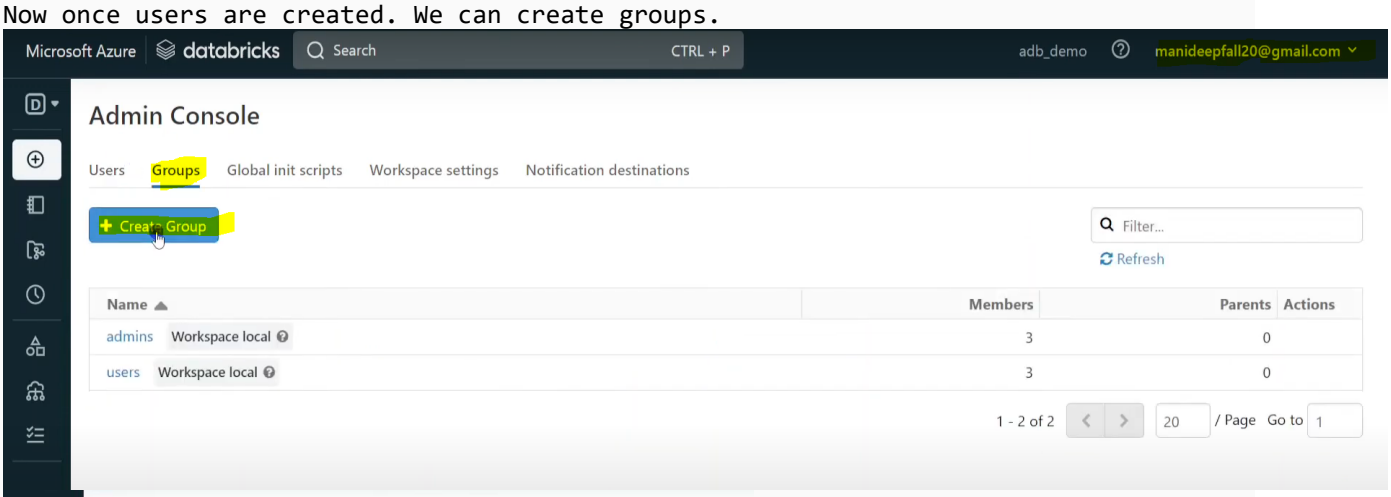
Now go to azure portal → search for Azure Active Directory→ go to Users in left side manage section.

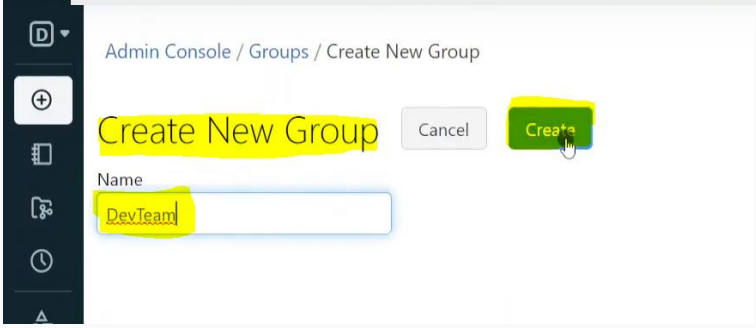
With in users currently we are having only 3 users. We can create new user by clicking on + new user.



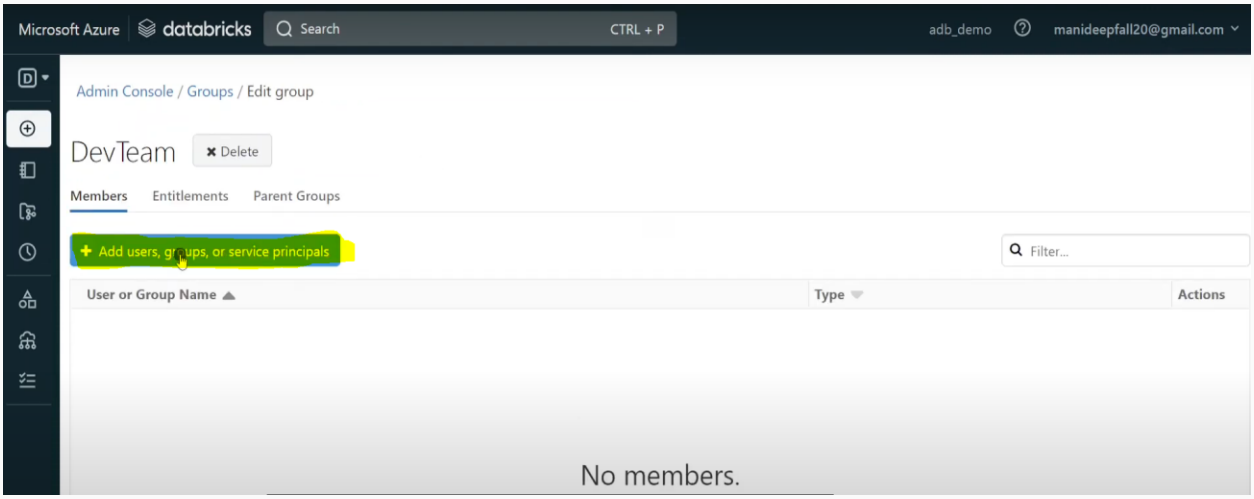
In case that particular user is not part of azure platform that user is external even we can call that user using invite external user. After adding user in AAD, now we will come back to data bricks and add user in admin console.

Now once users are created. We can create groups.

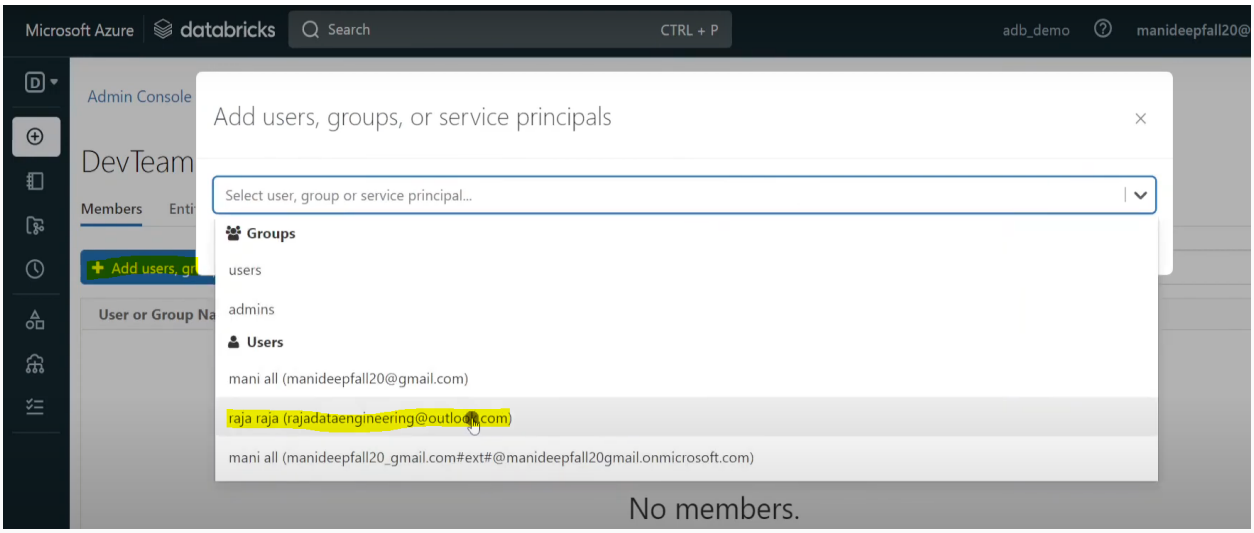




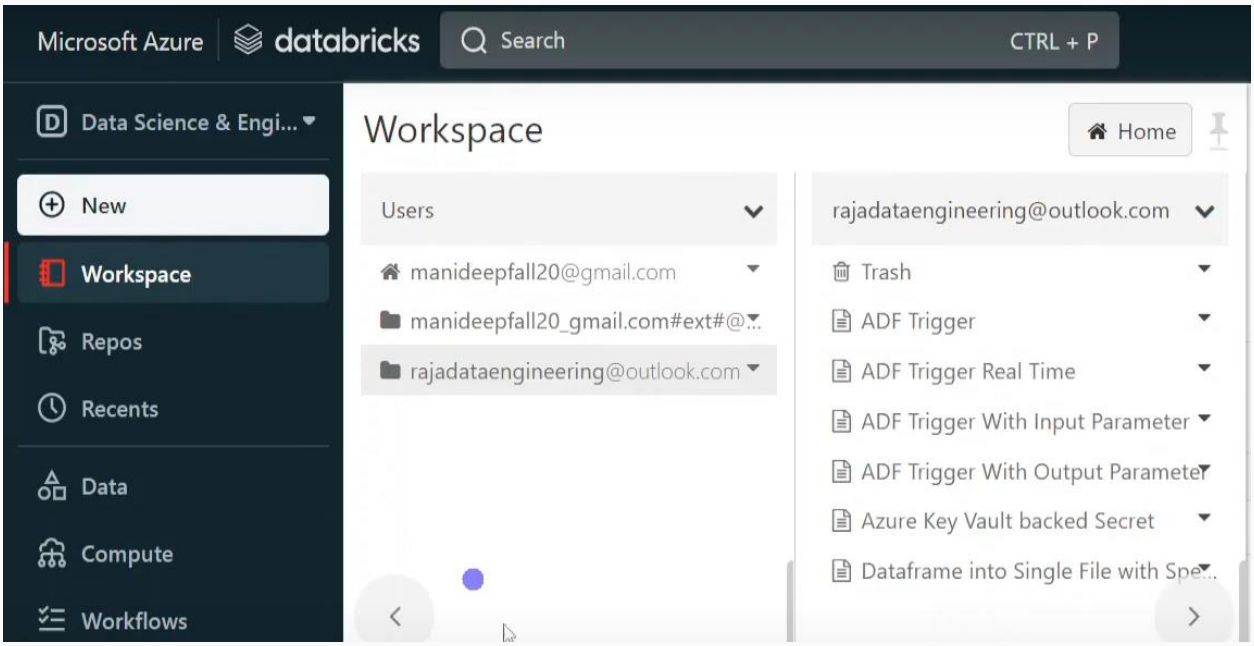
Now we will click on add users as shown below in DevTeam group.



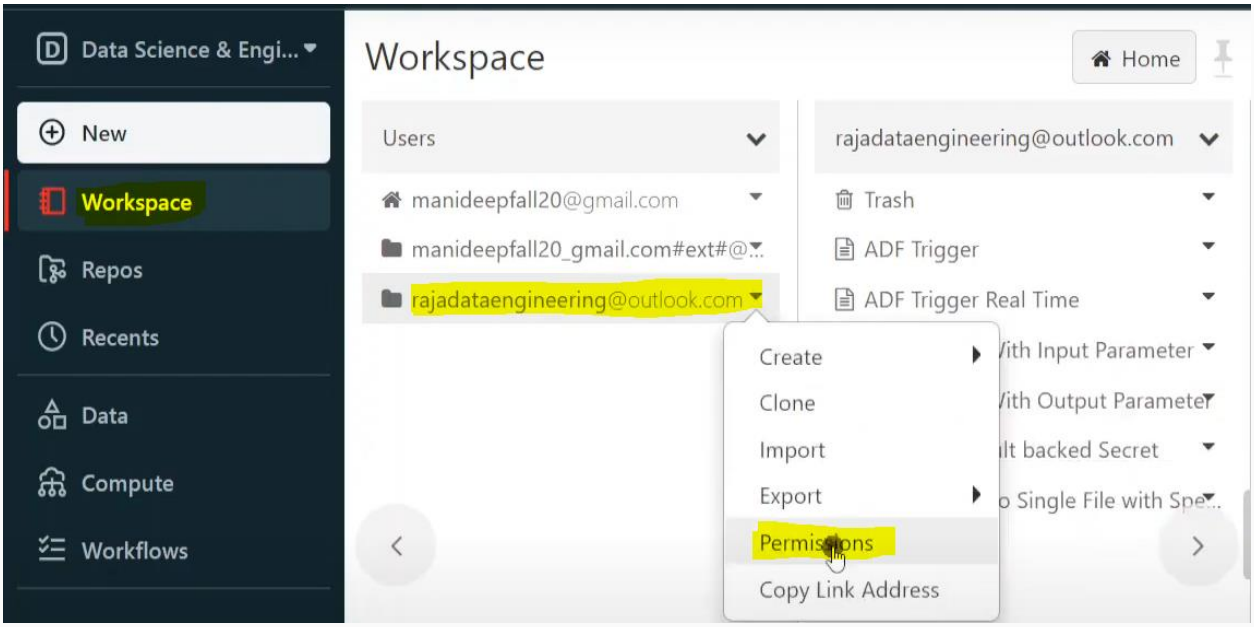
We can select required users and click on confirm.



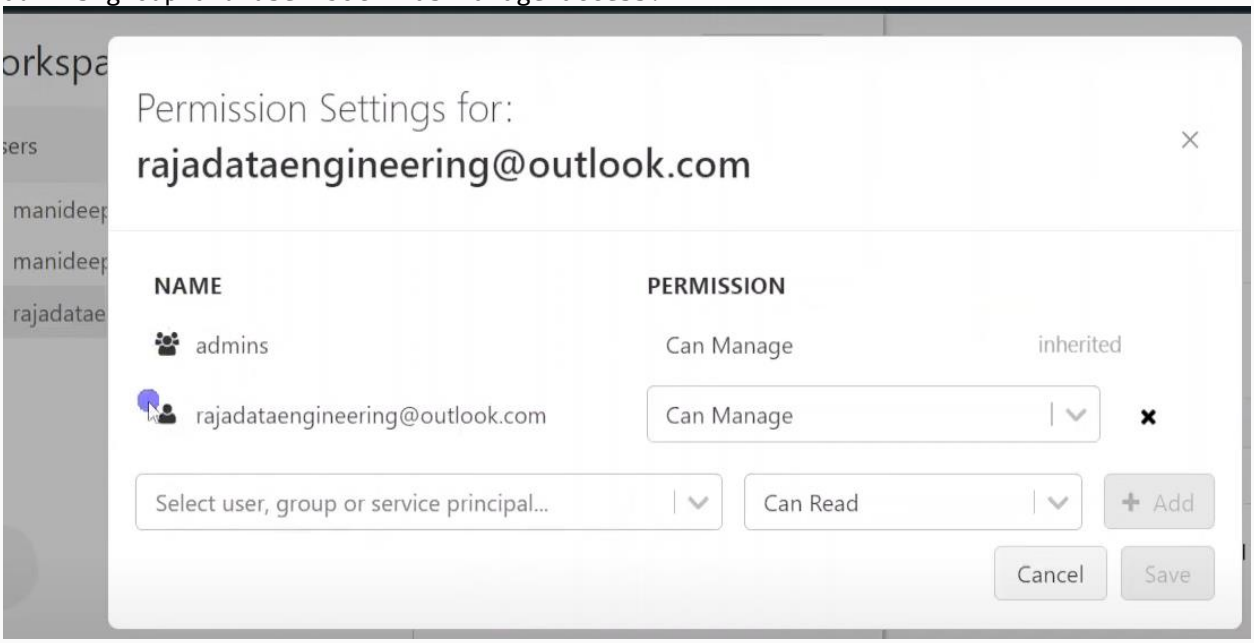
Once we created users and groups, we need to limit the access for any users. Within workspace level, at the root level we can see one folder is a shared and another one is users. Shared workspace is a place where we can put our code that can be seen by all developers. This place is more suitable for collaboration so whenever we are working as a team, we can put common files under shared. But coming to users, when we have multiple developers different folders will be created for all the developers so they will have lot of code. Let's say we are having many users. Within that we are having many notebooks its not good idea to give access to all the notebooks to all the users. In order to limit the access either we can go with folder level or with notebook level.



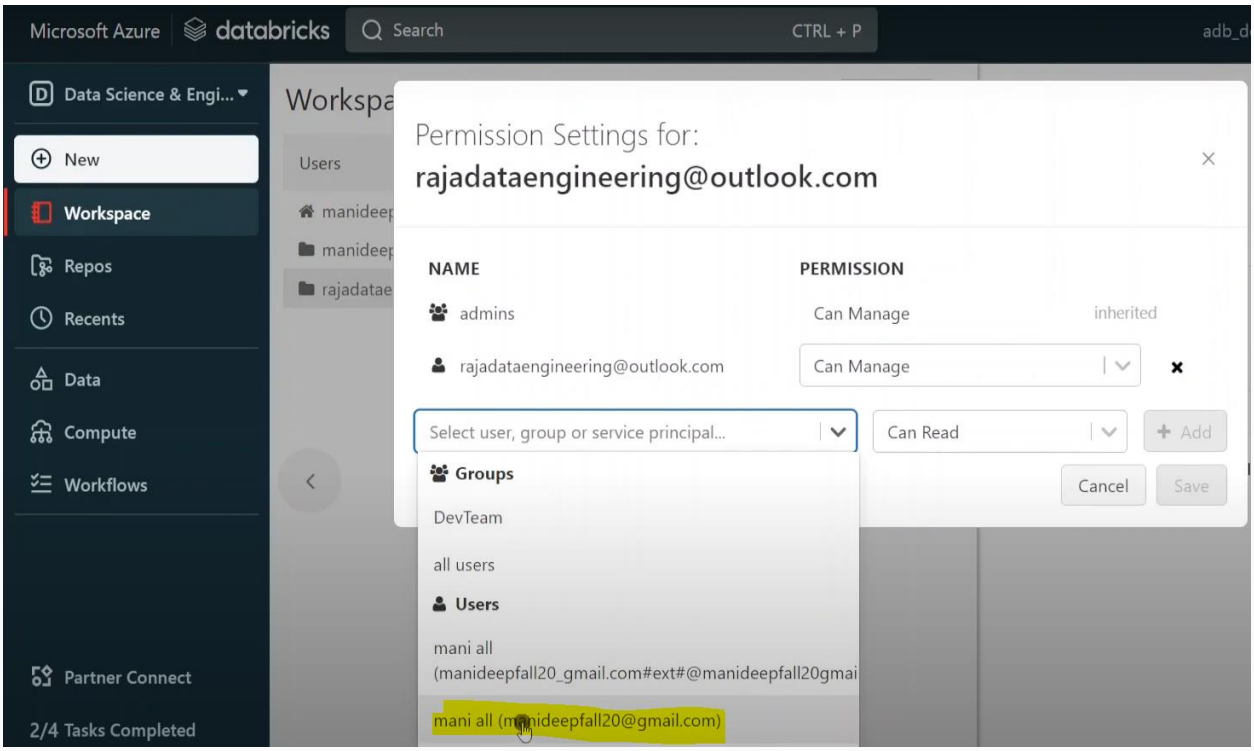
Let's say we have to restrict access for this folder as shown below, we need to go to permissions as shown below.



Here we can see admins is group and rajadataengineering@outlook.com is user. Here admins group and user both has manage access.



If I want to restrict for some other user, what can we do is



The same process is applied for notebook level. Click on notebook → click on permission→ which user or group we want to grant , which user access all need to provide same as above.

HOW TO MAINTAIN SECRETS IN DATABRICKS DEVELOPMENT?

Maintaining secrets such as password, username,server details should not be exposed to the development environment otherwise it will add risk. Any solution with hard coded sensitive details will not be approved for the deployment.

Databricks is providing few options to handle these secrets at high level called secret scopes in databricks.

WHAT IS SECRET SCOPE? Secret scope is collection of secrets needed for data bricks development . It will be preventing sensitive details such as password, username to all developers. It is avoiding hardcoding information.

TYPES OF SECRET SCOPES: 1) Azure Key Vault-backed scopes 2) Databricks – backed scopes

Azure Key Vault-backed scopes means all the secrets will be maintained within azure key vault and we can create secret scopes in data bricks which means we can integrate data bricks with azure key vault with some mechanism called Azure Key Vault- backed scopes. The purpose of azure key vault is to store the secrets securely. When we are creating this backed scopes what happens is databricks can seamlessly talk to Azure key vault and it can data retrieve passwords dynamically at runtime whenever needed and even that's cannot printed in databricks environment so it's very secure

Databricks – backed scopes means databricks itself maintaining one encrypted database. Within that database, all secrets can be maintained. So, we don't need to create any external service for this option. Within databricks itself secrets can be maintained.

URL for creating secret scope is
https://<databricks-instance>#secrets/createScope
once we enter the above url , we need to enter
secret name,
Azure Key Vault DNS Name and
Azure Key Vault Resource ID.
This 3 informations can be taken from Azure Key Vault service.

SOLUTION WITHOUT SECRETS SCOPE

Let's say we have a requirement to fetch data from one of the azure SQL database and process the data and then finally write output into ADLS

```
#JDBC CONNECTION DEFINITION
jdbcHostname = "ss-demo-rajade.database.windows.net"
jdbcPort = 1433
jdbcDatabase = "asqlDemo"
jdbcUsername = "rajade"
jdbcPassword = "Test@1234"
jdbcDriver = "com.microsoft.sqlserver.jdbc.SQLServerDriver"
jdbcUrl =
f"jdbc:sqlserver://{jdbcHostname}:{jdbcPort};databaseName={jdbcDatabase};user={jdbcUsername};password={jdbcPassword}"
```

we can clearly see we have hardcoded this in the notebook and any developer can see this sensitive information and this jdbc connection we are creating data frame.

```
#READ FROM AZURE SQL DATABASE
empDF =
spark.read.format("jdbc").option("url",jdbcUrl).option("dbtable","dbo.emp").load()
display(empDF)
```

	name	salary
1	raja	1000

```
#Create Mount Point to Integrate with ADLS
```

We have to write data into ADLS so for that we have to write something known as mount point. Using mount point, databricks can integrate with ADLS. Here we hardcoded access key information . These are quite sensitive information

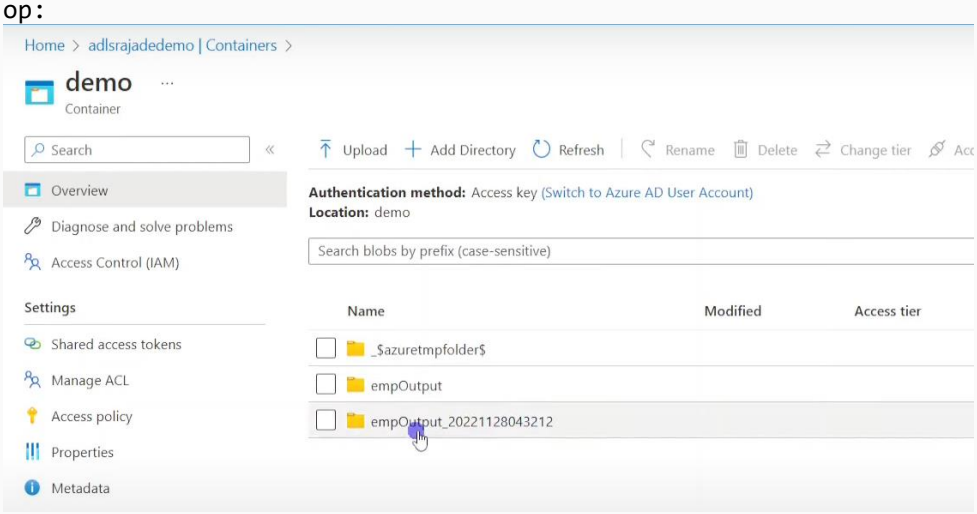
Create Mount Point to Integrate with ADLS

```
dbutils.fs.mount(
      source = "wasbs://demo@adlsrajadedemo.blob.core.windows.net",
      mount_point = "/mnt/adls_demo",
      extra_configs =
{"fs.azure.account.key.adlsrajadedemo.blob.core.windows.net":"AFxQiWpRA5zn/WUKqD8AmkHv9pXVX33jFJ5eigxWxLNAtcujCloy6iLdwh9aj1jwcAmCeGEFSUEd+AStbvk0hA=="})
```

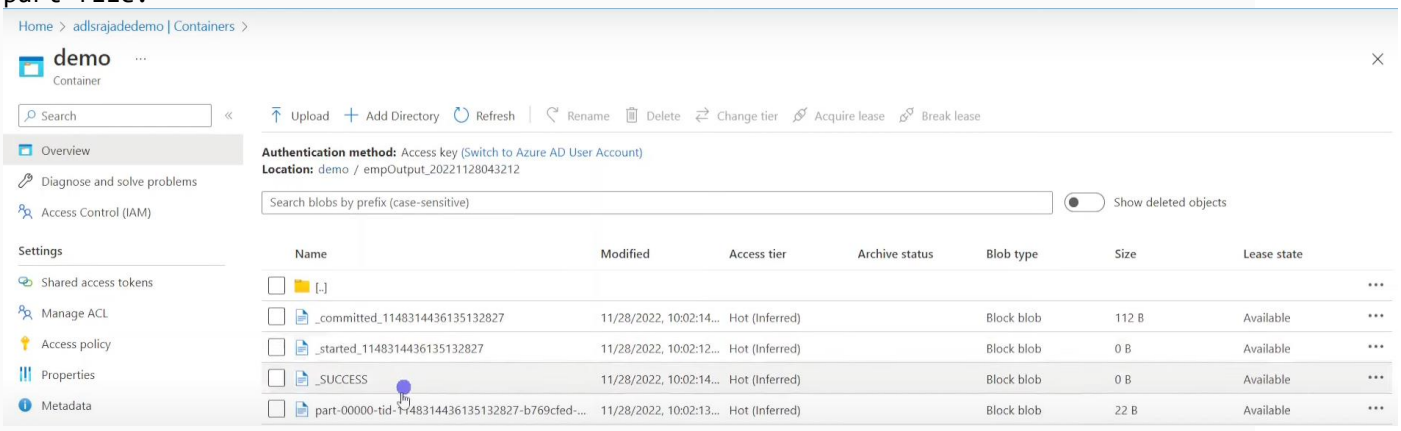
```
Out[13]: True
```

```
#Write Dataframe into ADLS
from datetime import datetime
outPath = '/mnt/adls_demo/empOutput_'+datetime.now().strftime("%Y%m%d%H%M%S")+'/'
empDF.write.format("csv").option("header",True).option("sep","|").save(outPath)
```

op:



Here empOutput appended with current timestamp. If we open folder , we can see that part file.

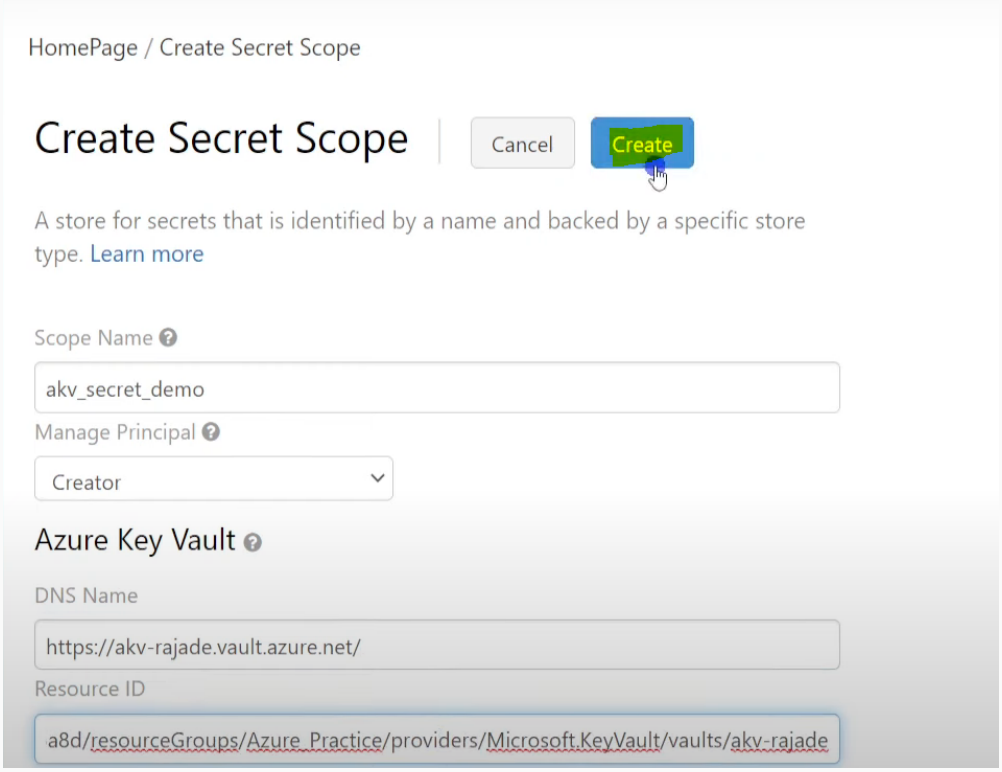


Here in above code, all secrets are exposed and we have to maintain all these secrets within that.we should not expose all the sensitive information such as username, password. So, for that we will create secret scope.

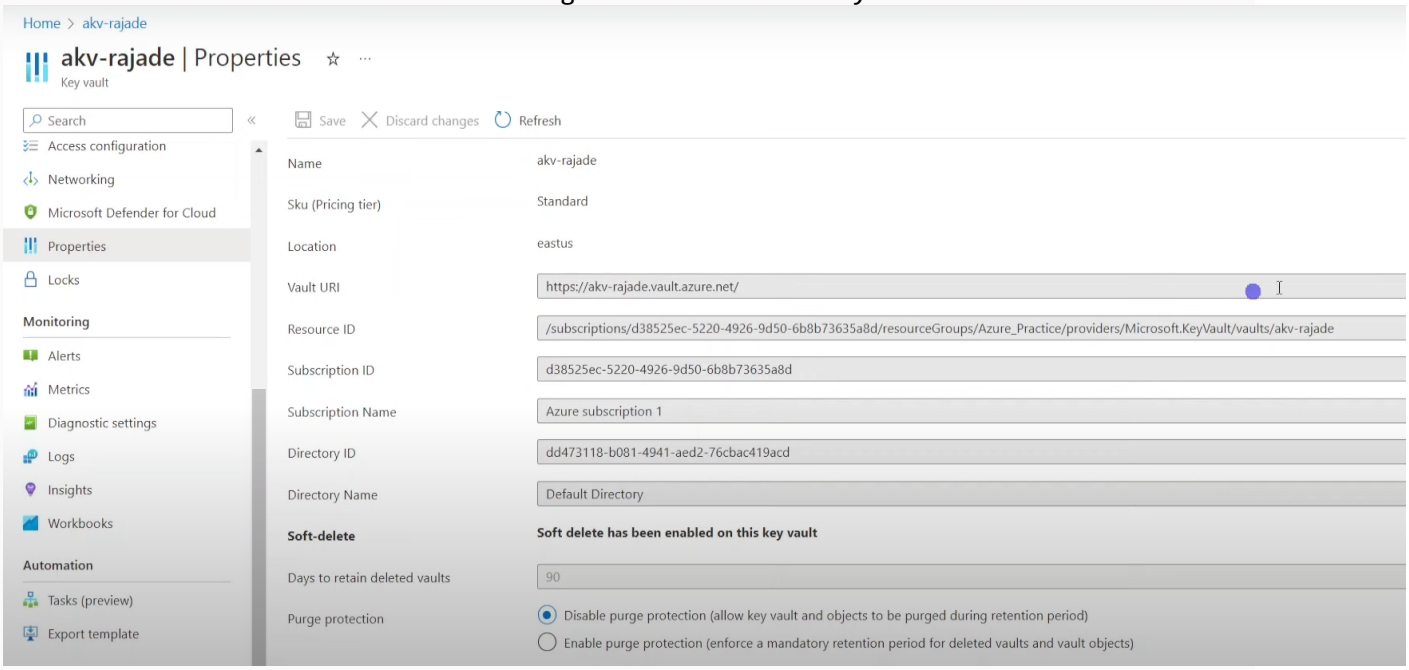
In our existing databricks url

https://<databricks-instance>#secrets/createScope

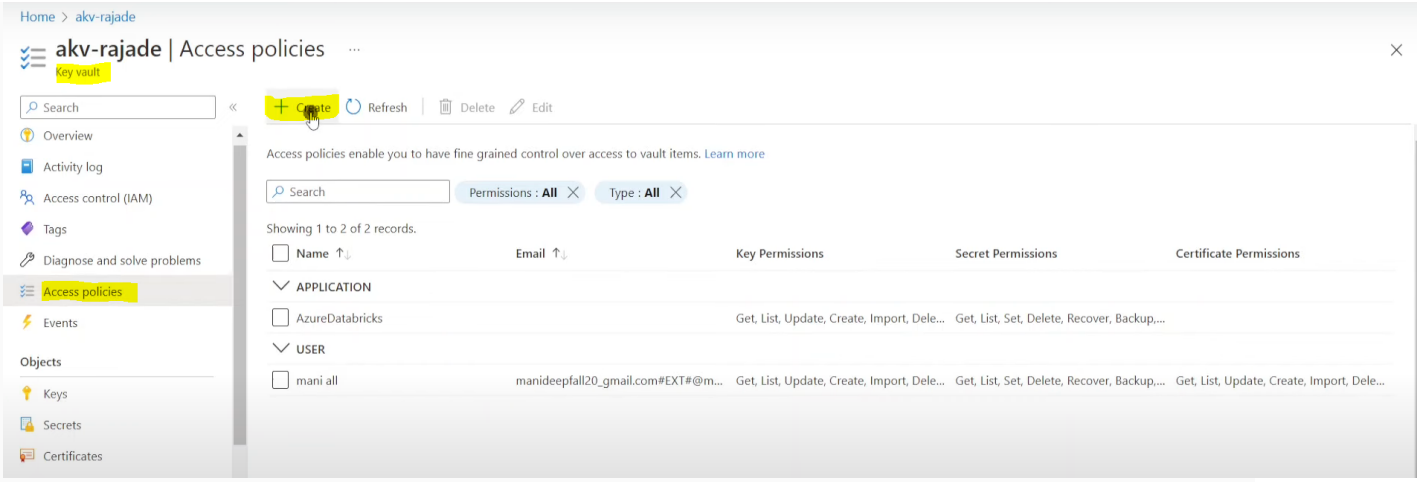
In our existing databricks url, till # we need to keep, after # we need to add secrets/createScope



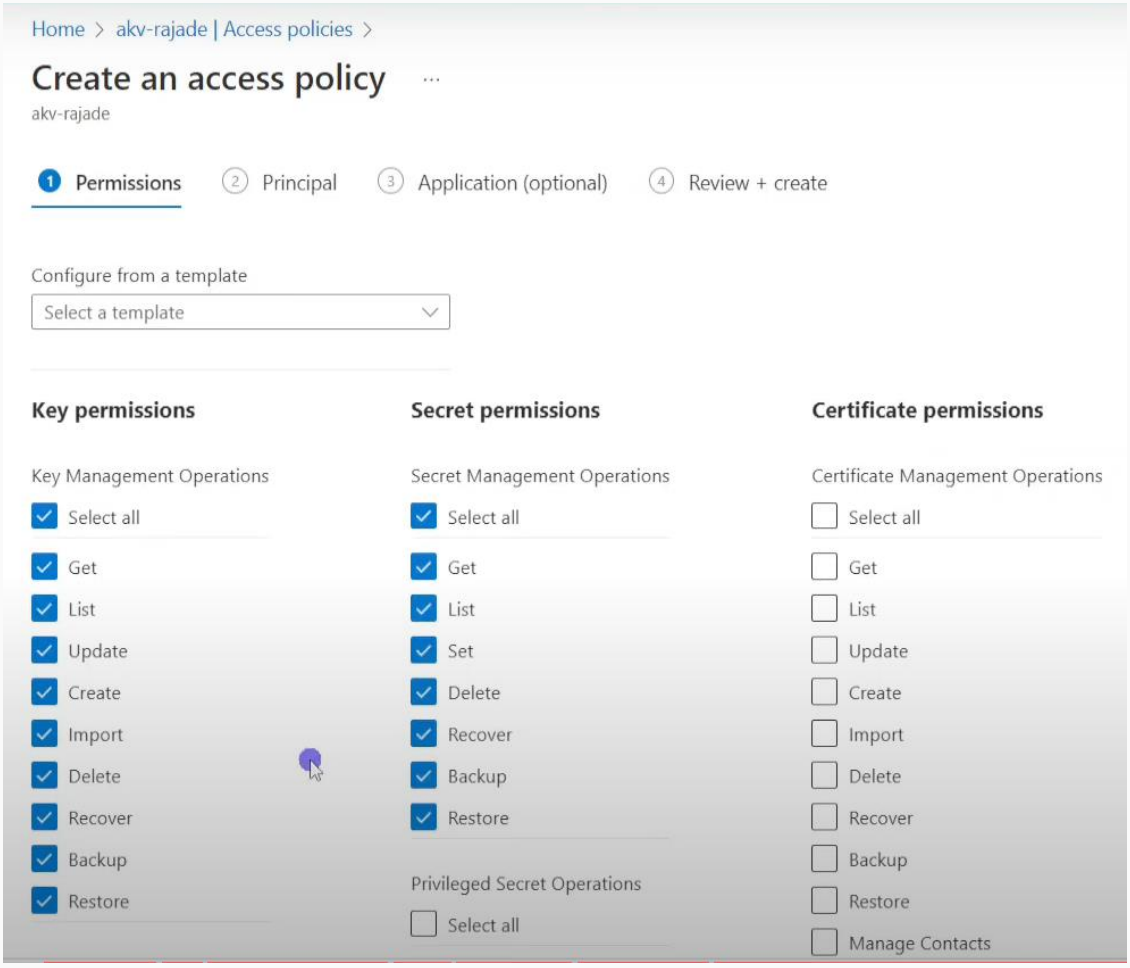
Here DNS Name and Resource ID we will get it from Azure Key Vault.



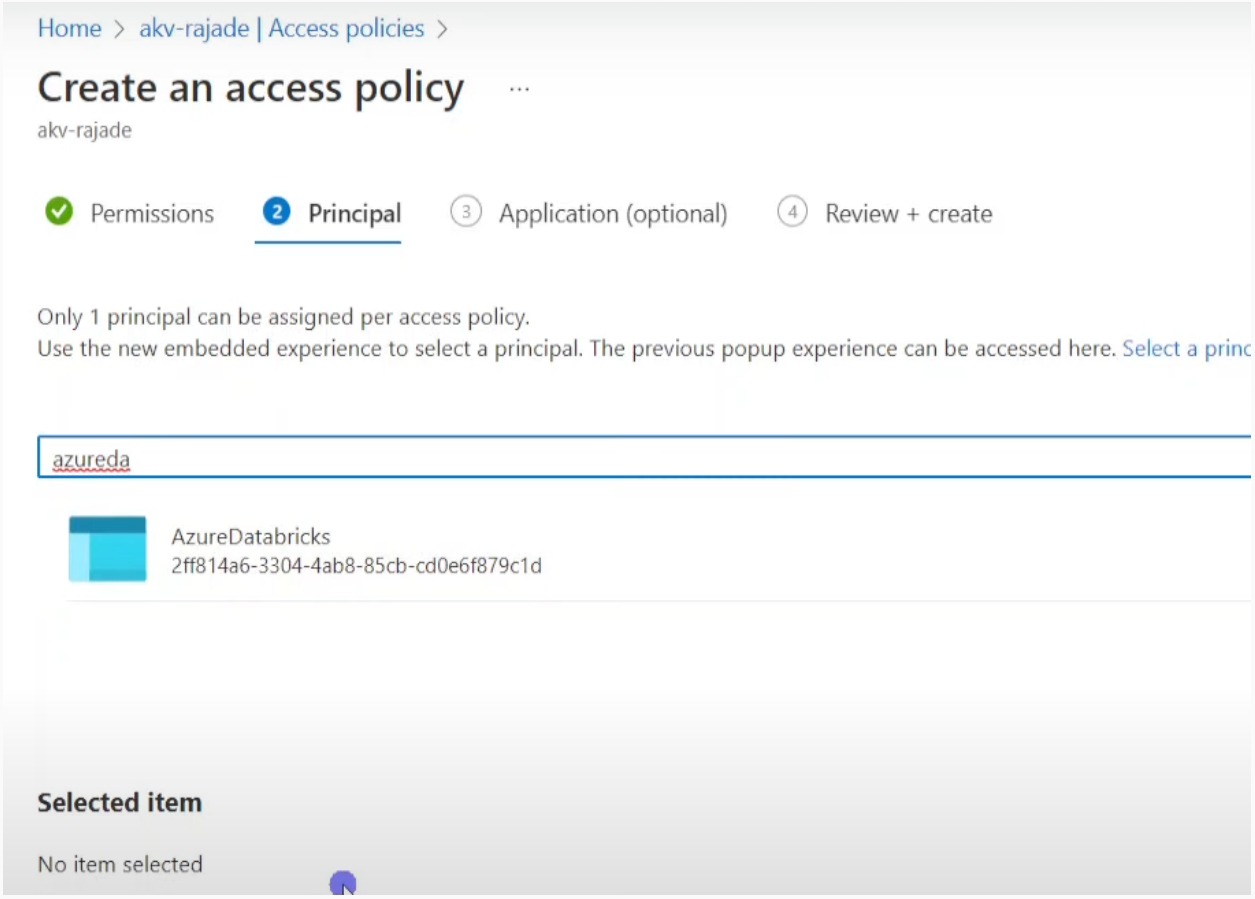
In order to create the scope in data bricks, user should have right privilege in the AAD. Apart from that we should have enabled access policies for the databricks as well.



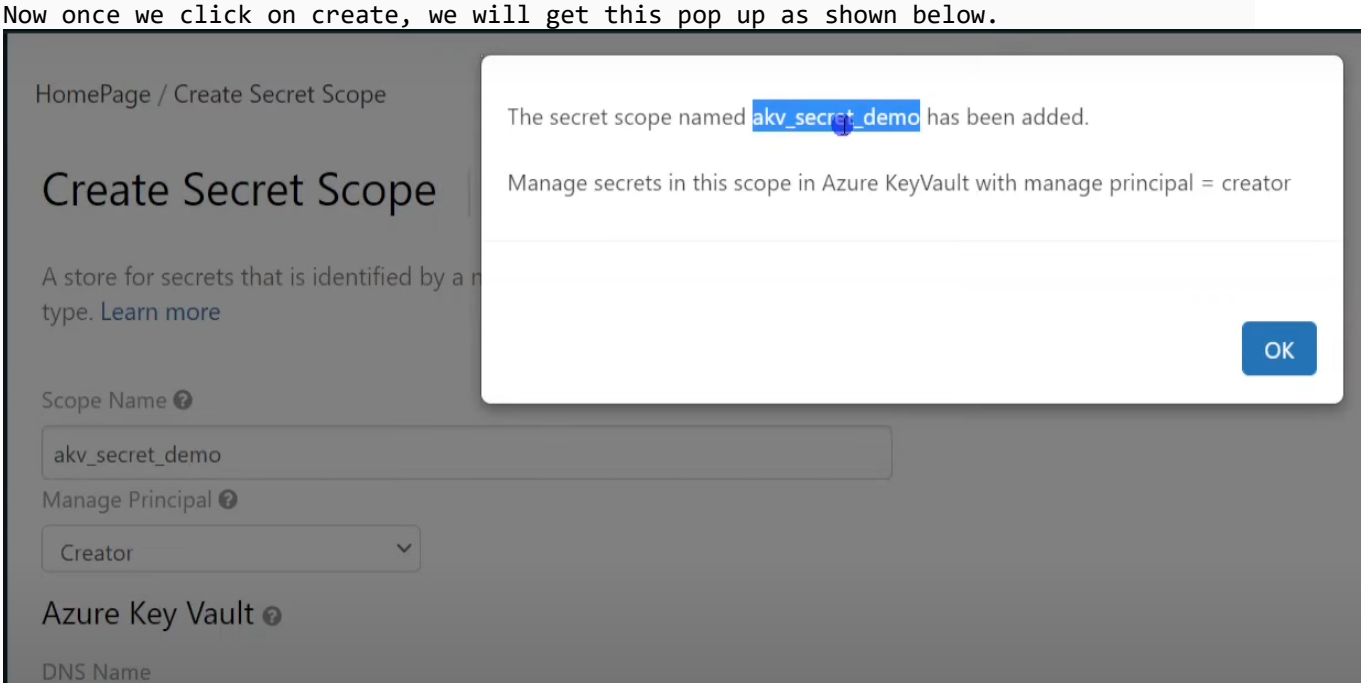
Now we need to select permissions what are the permissions want to give for any external service, in this case it is data bricks, databricks can come into this AKV and what are the operations it can perform and for what category it can perform key or secret or certificate.



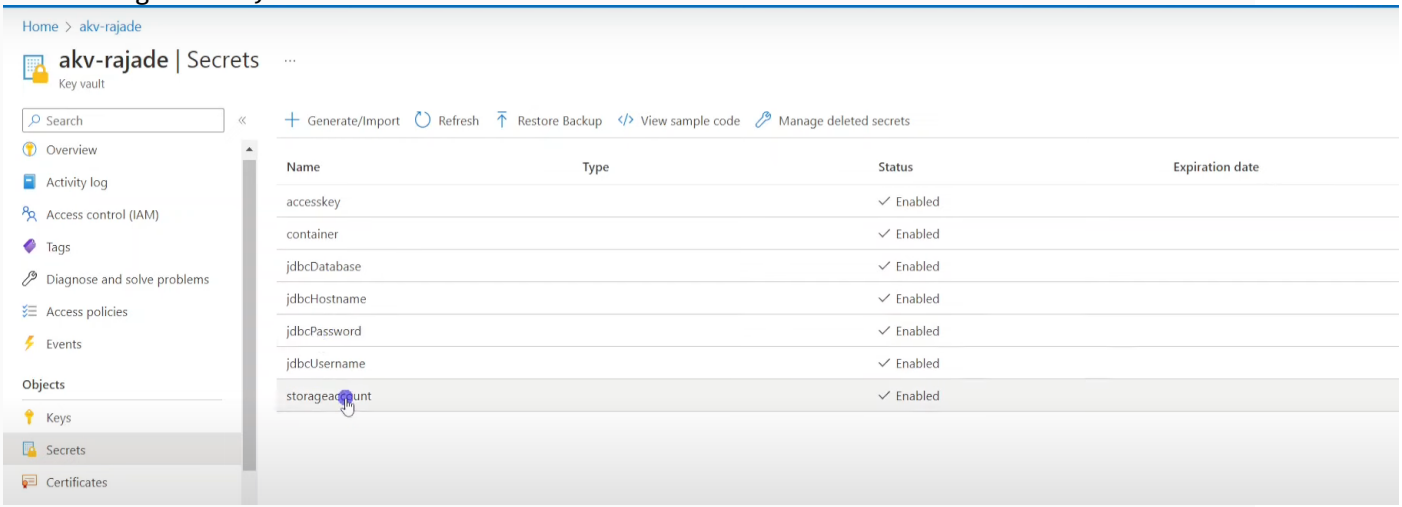
In the next step, we need to select user i.e; which service we are selecting azure data bricks in this case.



Now once we click on create, we will get this pop up as shown below.

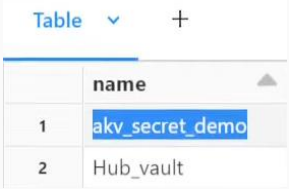


Now we have created secret scope for particular key vault which means databricks can talk to AKV and it can retrieve any secret from there.
Now coming to AKV, we have created secrets as shown below.

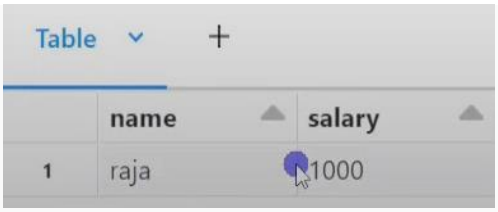


SOLUTION WITH SECRET SCOPE: AZURE KEY VAULT INTEGRATION

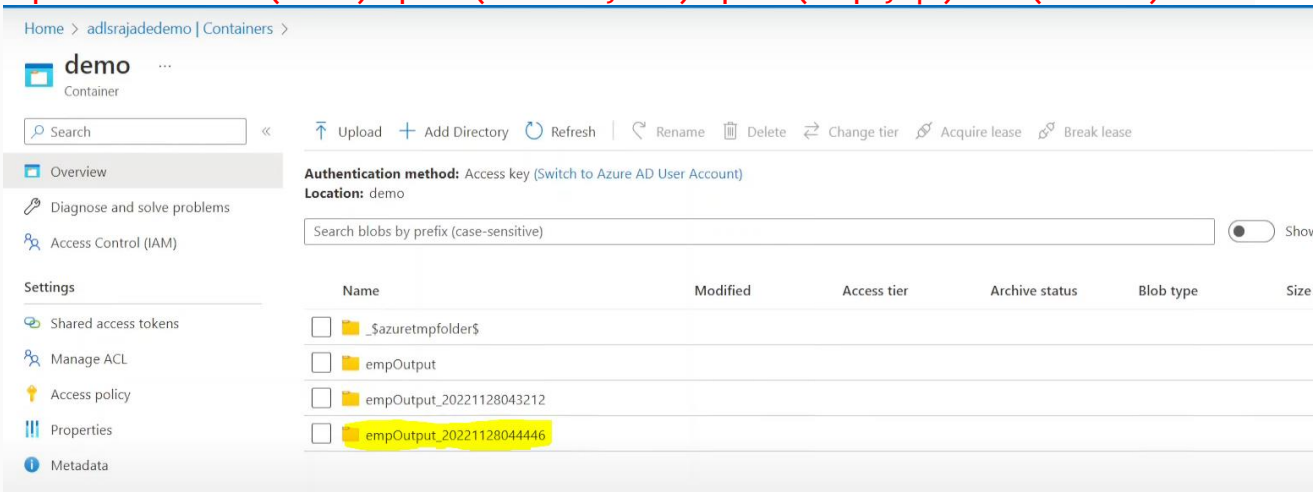
```
#List down secret scopes
display(dbutils.secrets.listScopes())
```




```
#JDBC Connection Definition
jdbcHostname = dbutils.secrets.get(scope= “akv_secret_demo” , key = “jdbcHostname”)
jdbcPort = 1433
jdbcDatabase = dbutils.secrets.get(scope = “akv_secret_demo”, key=”jdbcDatabase”)
jdbcUsername = dbutils.secrets.get(scope= “akv_secret_demo” , key = “jdbcUsername”)
jdbcPassword = dbutils.secrets.get(scope = “akv_secret_demo” , key = “jdbcPassword”)
jdbcDriver = “com.microsoft.sqlserver.jdbc.SQLServerDriver”
jdbcUrl =
f”jdbc:sqlserver://{jdbcHostname}:{jdbcPort};databaseName={jdbcDatabase};user={jdbcUsername};password={jdbcPassword}”

#Read from Azure SQL Database
empDF=
spark.read.format(“jdbc”).option(“url”,jdbcUrl).option(“dbtable”,”dbo.emp”).load()
display(empDF)
```



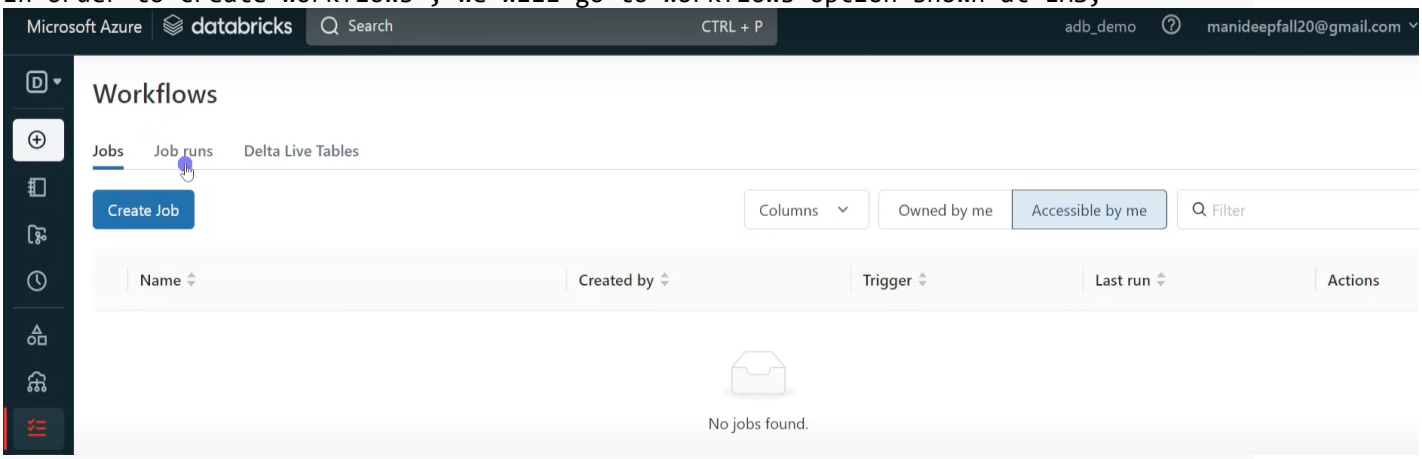

```
#Create Mount Point to integrate with ADLS
container = dbutils.secrets.get(scope= “akv_secret_demo” , key = “container”)
storageaccount = dbutils.secrets.get(scope= “akv_secret_demo” , key=”storageaccount”)
accesskey =  dbutils.secrets.get(scope= “akv_secret_demo” , key= ”accesskey”)

dbutils.fs.mount(
         source = f”wasbs://{container}@{storageaccount}.blob.core.windows.net”,
         mount_point = “/mnt/adls_demo”,
         extra_configs =
{f”fs.azure.account.key.{storageaccount}.blob.core.windows.net”:f”{accesskey}”})

Out: True
#Write Dataframe into ADLS
from datetime import datetime
outPath = ‘/mnt/adls_demo/empOutput_’+datetime.now().strftime(“%Y%m%d%H%M%S”)+’/’
empDF.write.format(“csv”).option(“header”,True).option(“sep”,”|”).save(outPath)
```



DATABRICKS WORKFLOWS: JOB SCHEDULING

Once we have completed creation of pipelines then we need to schedule them for that we need to create jobs. In databricks we will do it via workflows. Within workflows, we can create jobs . jobs is nothing but collection of tasks. Tasks is nothing but running one specific business process so tasks might execute one particular databricks notebook. So, we can create multiple tasks and also, we can set up dependency between those tasks. So, it is similar to any scheduling tools such as airflow, ADF. In those tools, we can create a task and we can set up the dependency. In the same way, with in ADB workflow also we can create multiple tasks and we can set up dependency so that one of the task can run based on the status of the previous task and also, we can orchestrate serial and parallel task.

In order to create workflows , we will go to workflows option shown at LHS,



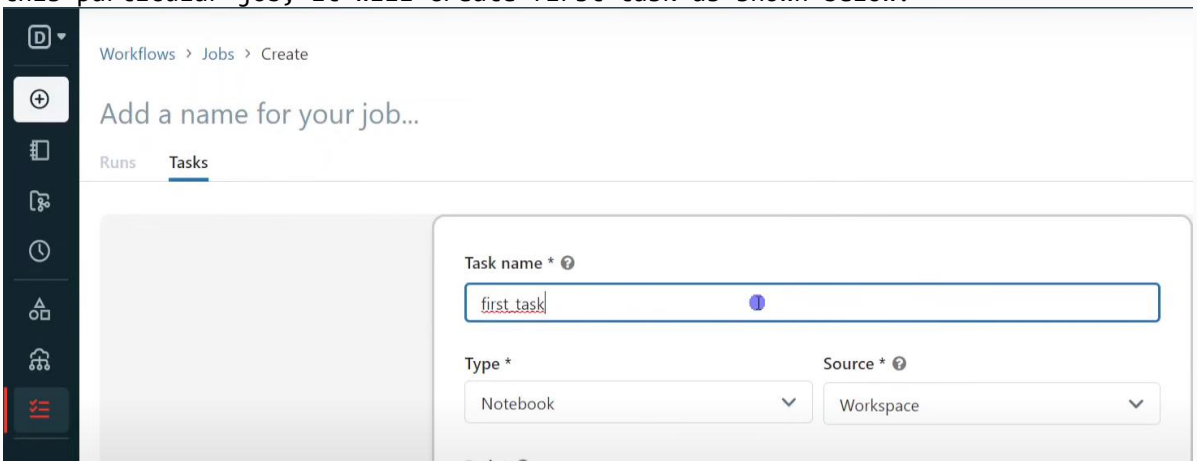
In workflows, we have few options , Jobs, Job runs and Delta Live Tables.

Jobs is the place where we can create our job.

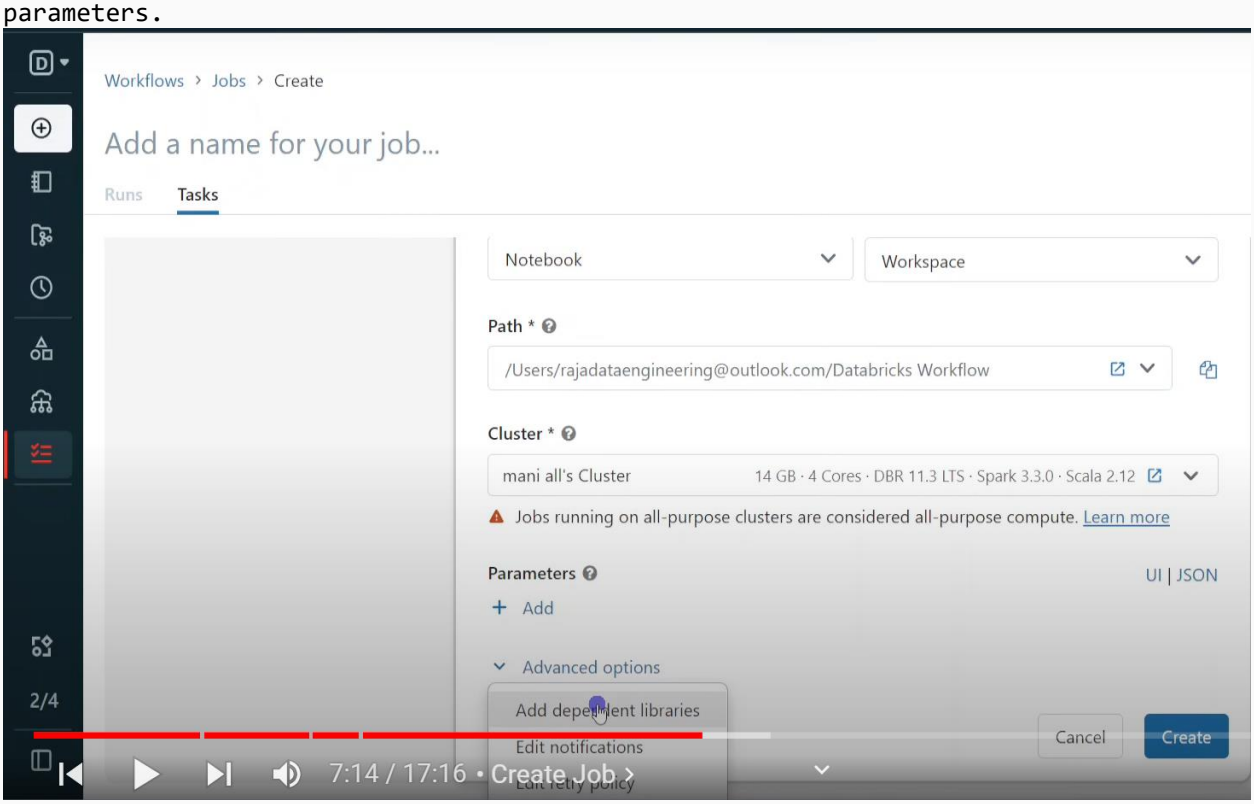
Job runs is incase we have executed any jobs earlier, those things can be seen in job runs.

Now in order to create job, we will click on create job as shown above.

Now if we click on create job, it will open one form that is for particular task. In this particular job, it will create first task as shown below.



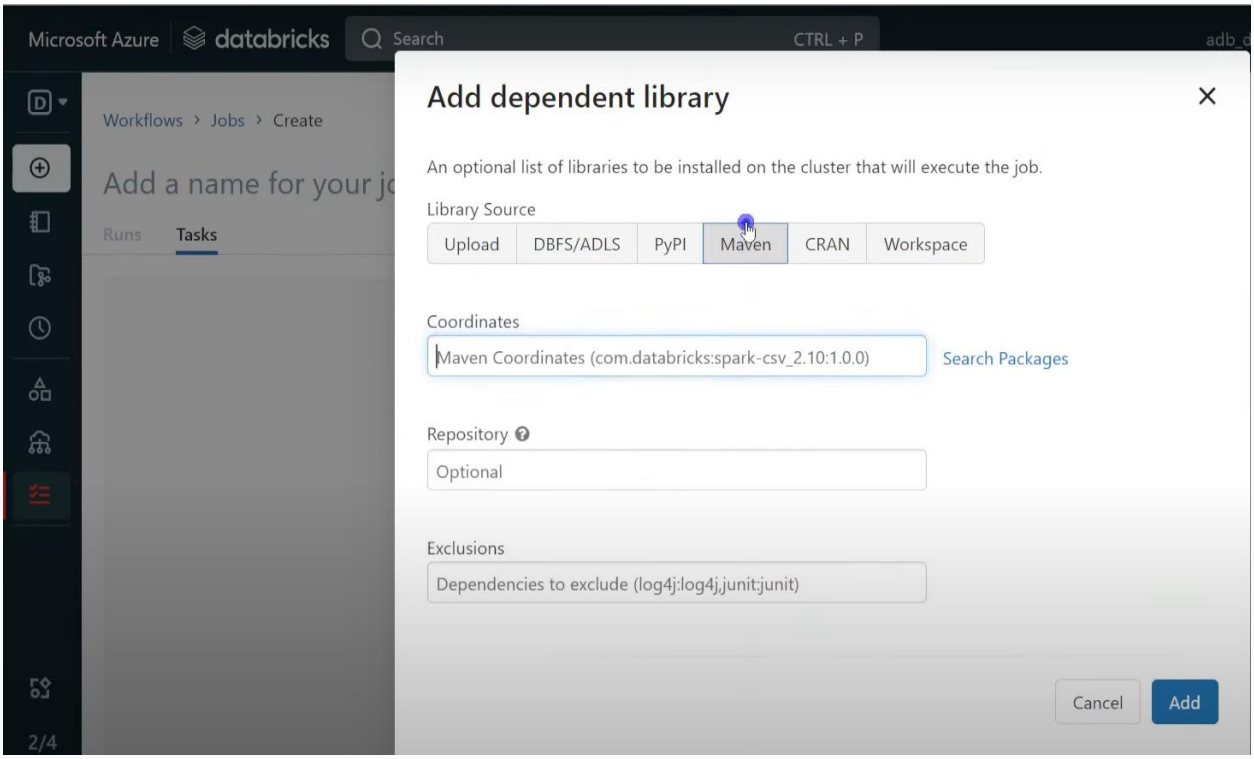
And also, here cluster will be created dynamically at the time of execution. Once time of execution is completed then it will be terminated. Coming to All Purpose cluster, cluster which we have created manually for our development purpose we can select the all-purpose cluster as well from drop down.
Now incase our notebook is expecting input parameters then that can be given using parameters . If there is no input widgets in our notebook, then we won't provide any parameters.



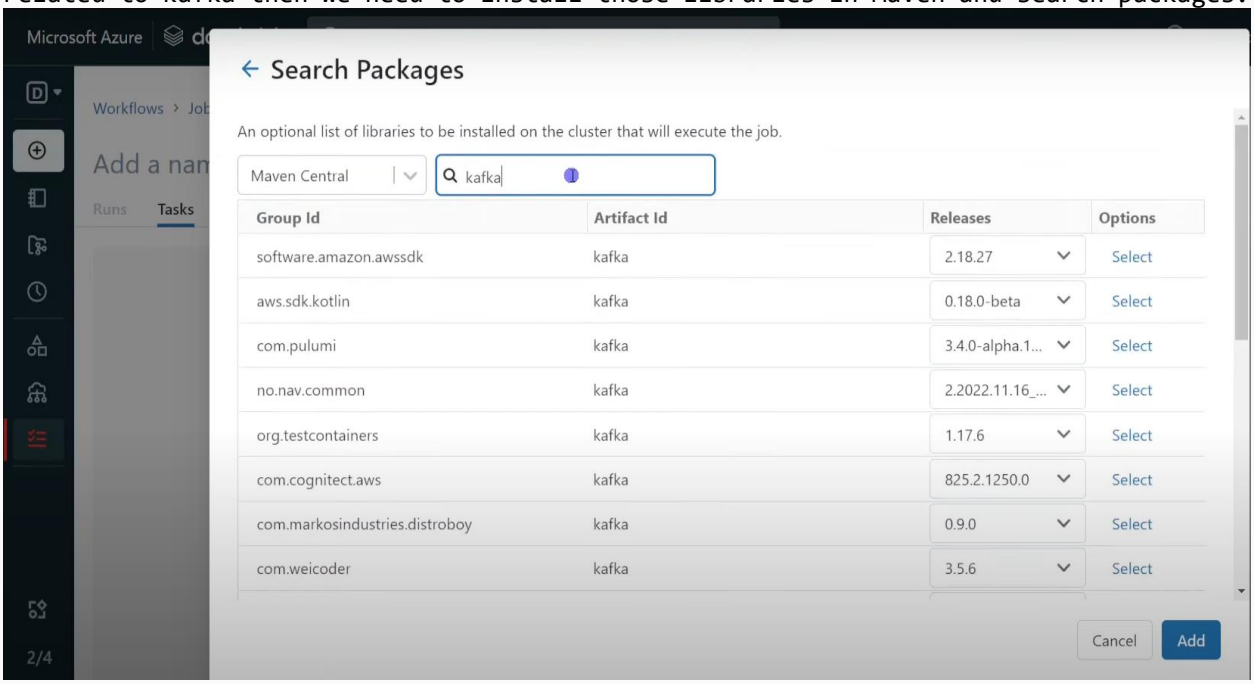
If we click on Advanced Options, we will have multiple options as Add dependent libraries, Edit Notifications , Edit retry policy.

Add dependent libraries means incase our notebook is expecting some dependent libraries that should be installed in job cluster then we can add that as well.

If we click on Add dependent libraries, we can see multiple libraries and library type as shown below.

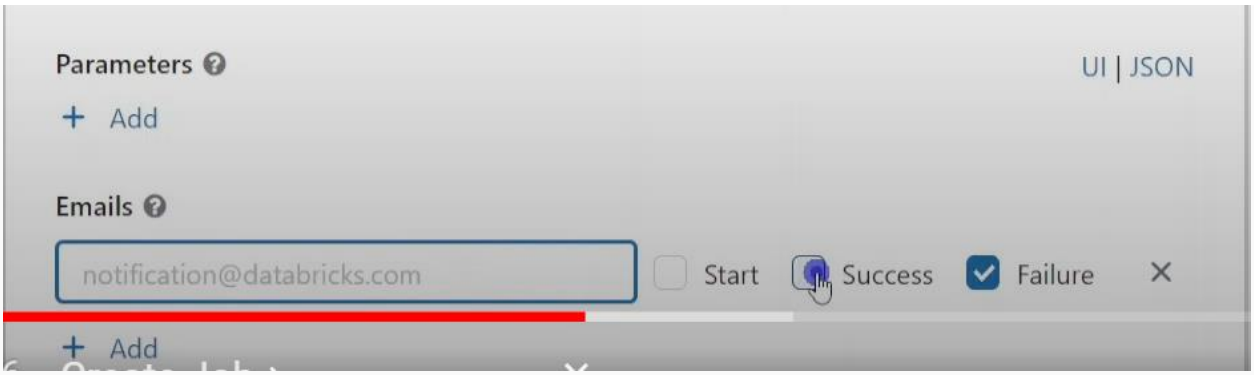


If we are developing some solution related to kafka then there are many libraries related to kafka then we need to install those libraries in Maven and search packages.



If our notebook is reading any of libraries, then we can go to that libraries.

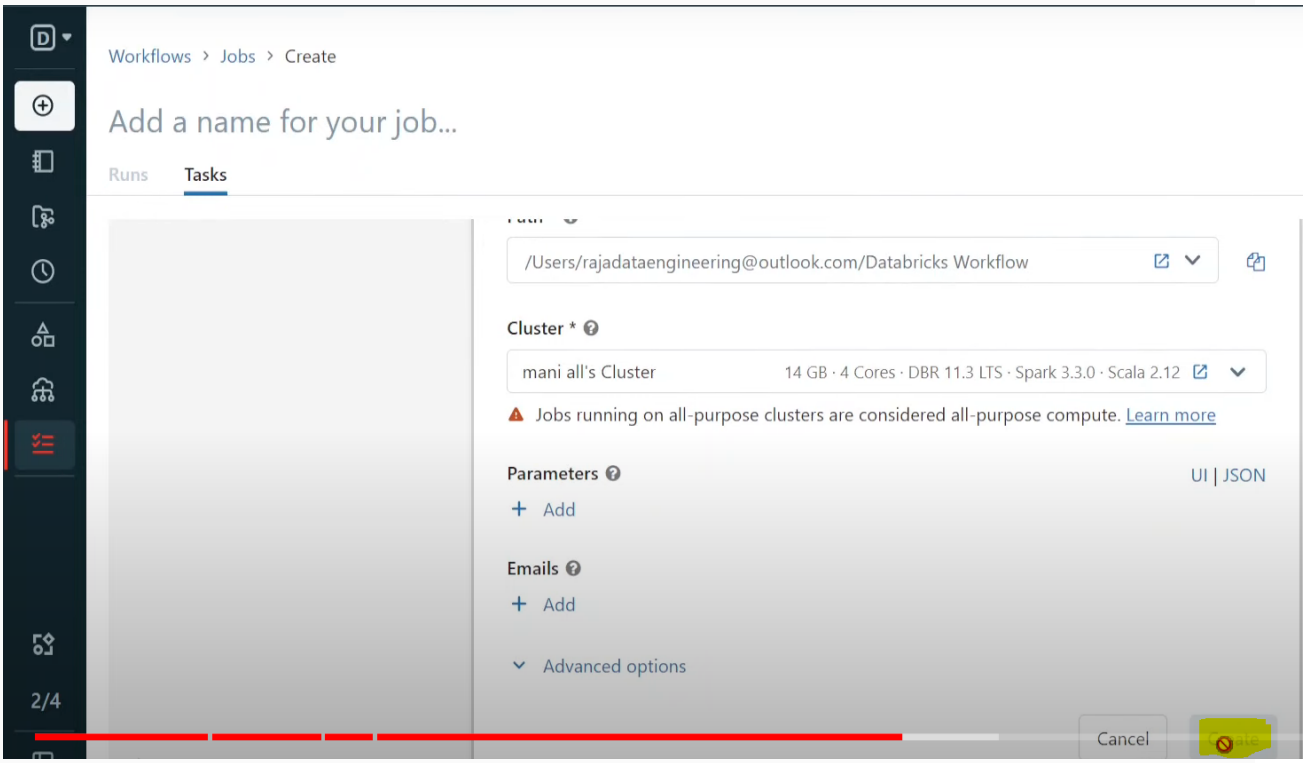
The 2nd option in advanced option is edit notifications. Incase if we want to receive some notification for this execution then we can add that as well. Once we click on edit notification, we will get Emails box as shown below.



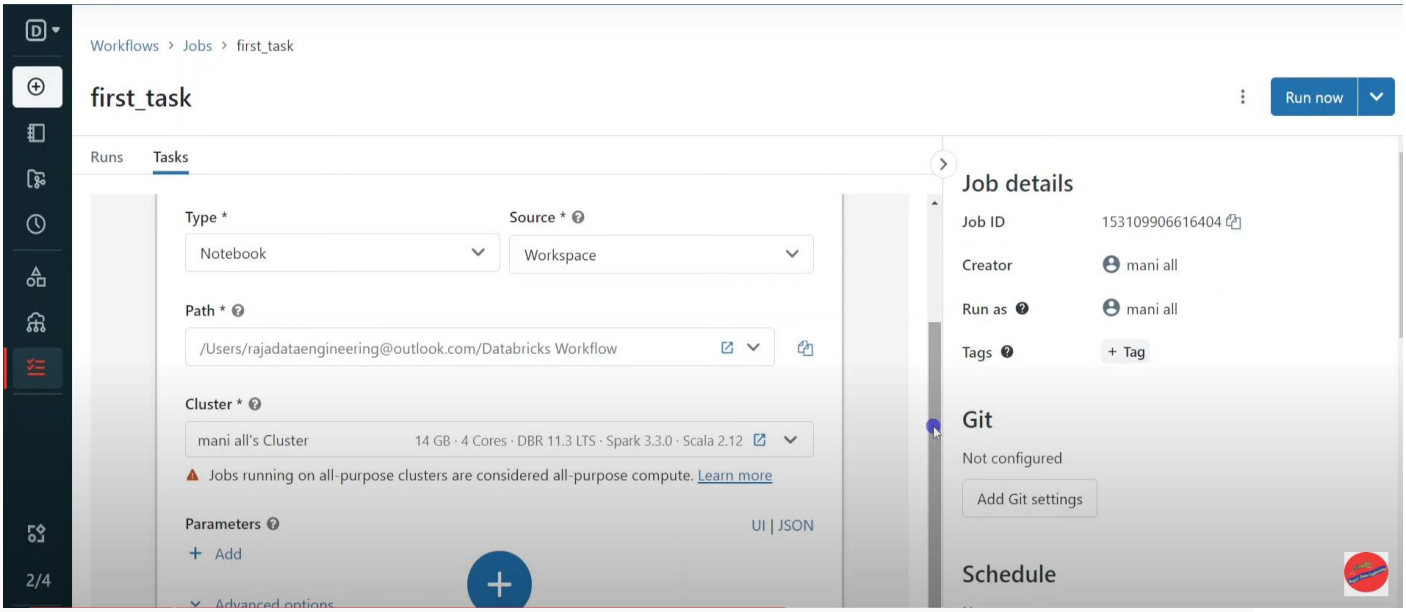
We need to mention name of the person and also for which status we want to get the email like once task is started we have to receive the alert or for success or for failure. We can choose anything we can select all 3 (start,success and failure).

3rd option in advanced option is edit retry policy is how many times the program can try to run the failed task again and again. Those parameters can be designed using that one.

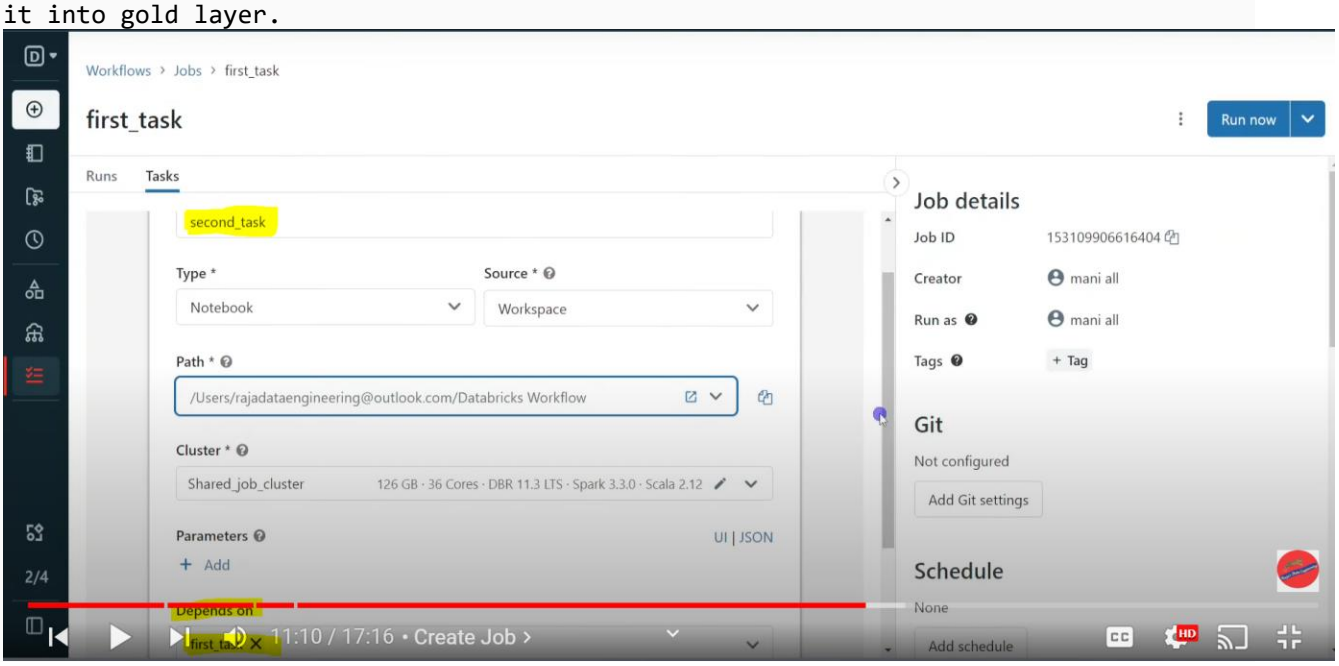
Now we will click on create.



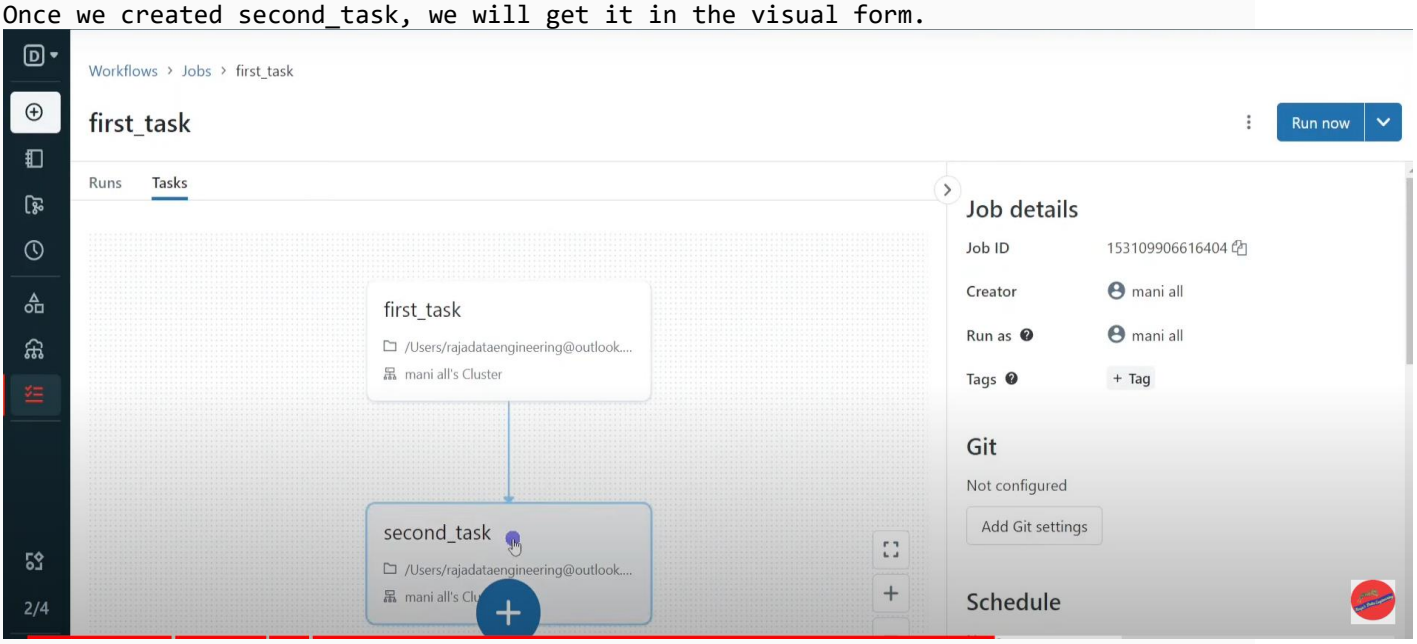
Now we have created 1 task for our job. 1 particular job can contain one or more tasks. In real time we can create very big orchestration framework as well by chaining multiple tasks.



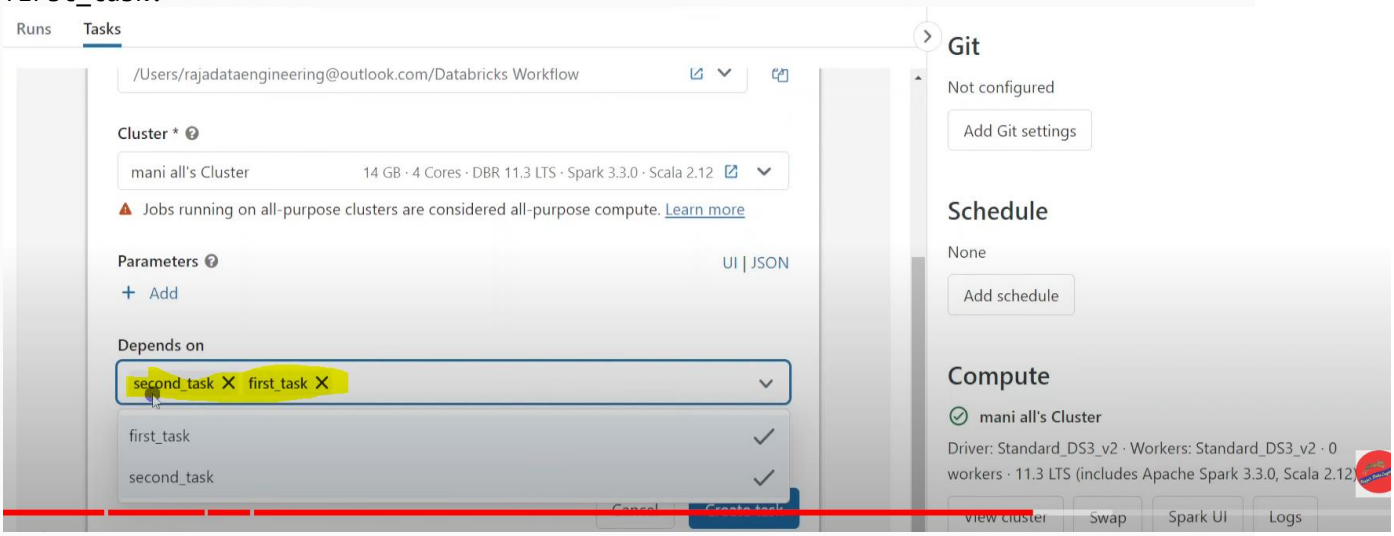
For example, if we want to add one more task on top of this task, we can click on blue + symbol. Once we click on + symbol, it will open same similar kind of form as above to perform next task. But why do we need 2nd task? Let us imagine in one architecture, one of the notebook that is pulling data from azure SQL and populating writing the data into ADLS. Then next notebook is reading data from ADLS and it is doing some basic transformations and then it is moving data into silver layer and another notebook it will read data from silver layer and again it is doing all business aggregations finally moving data to gold layer. We might have separate notebooks so we have to execute all the steps one by one. 1st notebook, it will read data from Azure SQL then writing into bronze layer then 2nd notebook it will read from bronze layer and writing the data to silver and 3rd notebook it is reading data from silver layer and writing it into gold layer.



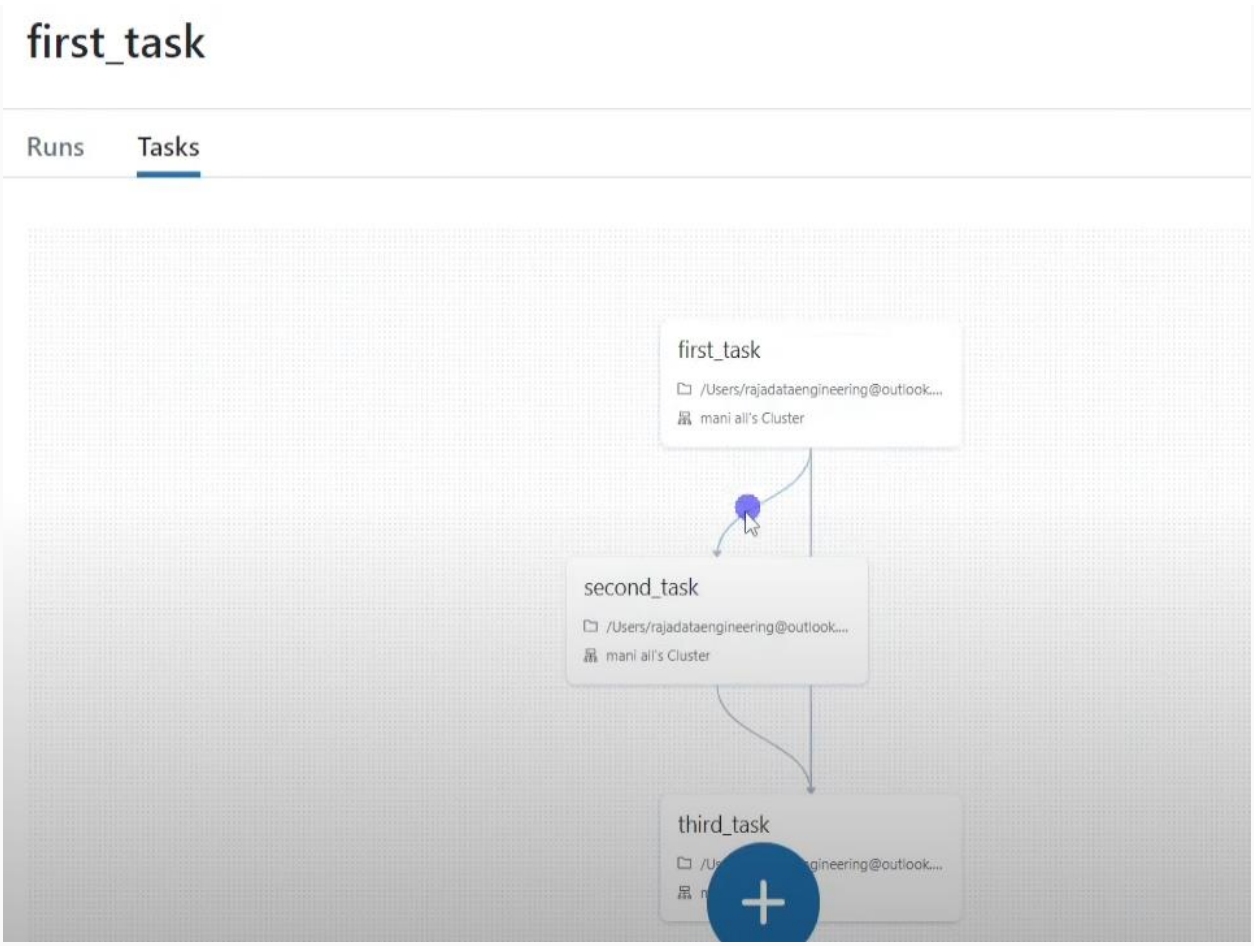
Now we created 2nd task and we have one more field Depends on. Depends on not there when we created 1st task because whenever we are creating from 2nd task onwards we can see this option depends on. So, either we can remove or add dependency.
Here in 2nd task, we has Depends on as first task which means this particular 2nd task is depending on 1st task which means first_task should be succeeded then only second_task will be executed. In case if 1st task is failed, control will not come to second_task.
Once we created second_task, we will get it in the visual form.



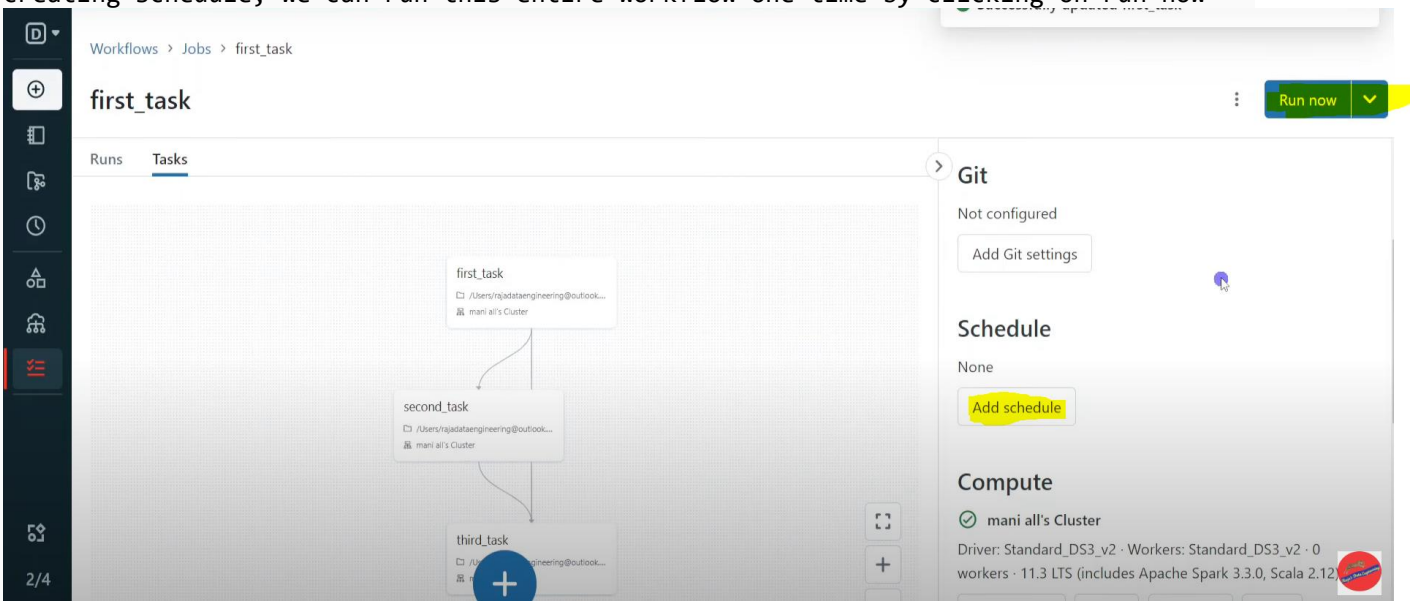
We can see in visual form, there is a first_task and a dependency line and once first_task is completed then control will go to second_task. We can even create 3rd task again by clicking on +. Now our third_task is depending on second_task and first_task.



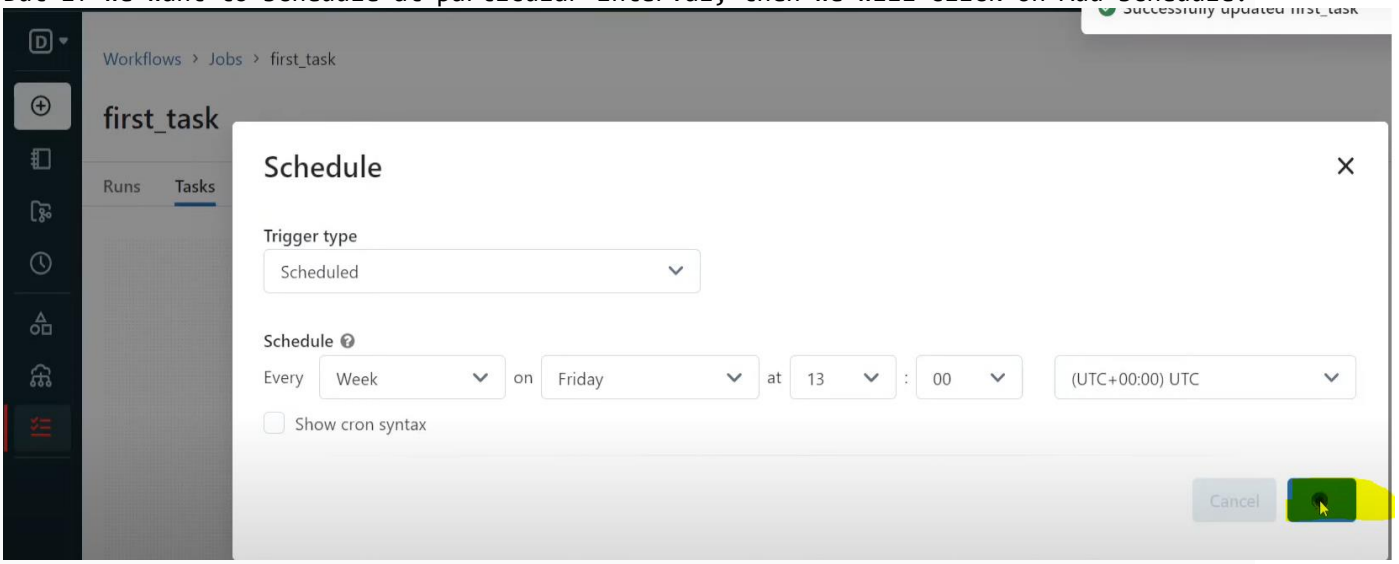
The first 2 tasks will be completed successfully then only control can come to 3rd task so we are setting up dependency.



Here we can see first_task and second_task is depending only on first_task and third_task is depending on 2nd task and 1st task. In this way we can create orchestration framework.we can create procedural logic to orchestrate particular workflows. With out creating schedule, we can run this entire workflow one time by clicking on run now

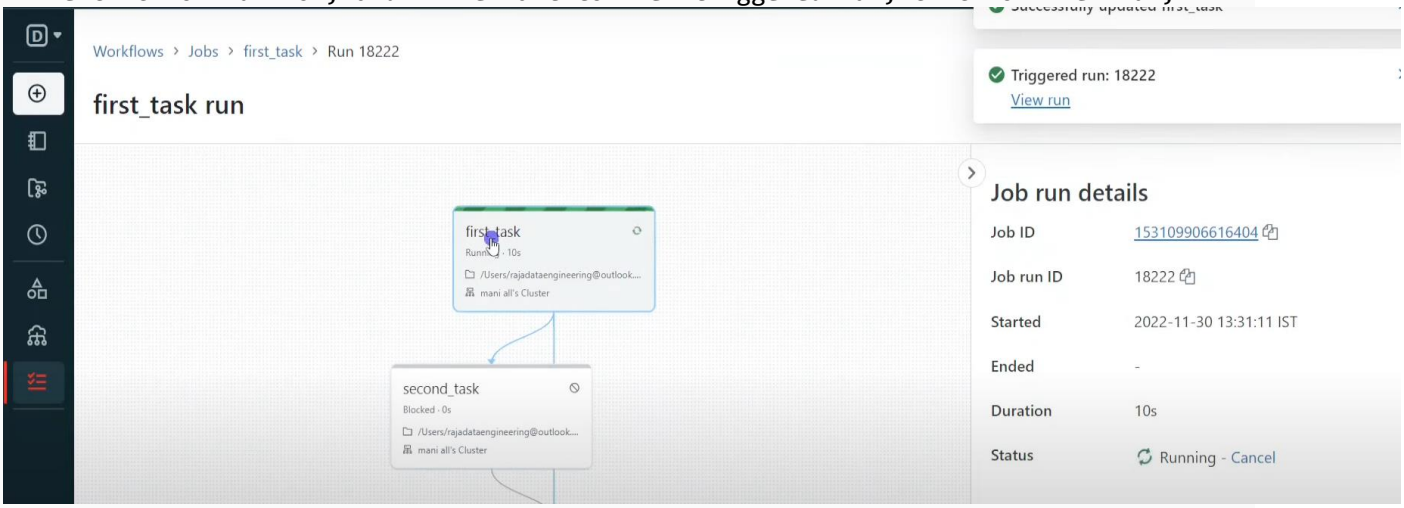


But if we want to schedule at particular interval, then we will click on Add schedule.



We will select trigger type as scheduled and schedule as schedule time and click on save.

If we click on run now, and if we want to view triggered run, click on view run,



HOW TO TRIGGER DATABRICKS NOTEBOOK EXECUTION VIA ADF

ADF is orchestration framework given by azure. Using ADF we can create ETL pipelines. ADB is a coding tool. ADF is code free visual tool. ADF is not only used for building ETL pipelines and also it acts as a scheduler. Even via ADB, notebook can be scheduled using databricks workflow or airflow or ADF.

WHY DO WE NEED ADF FOR TRIGGERING DATABRICKS NOTEBOOKS?

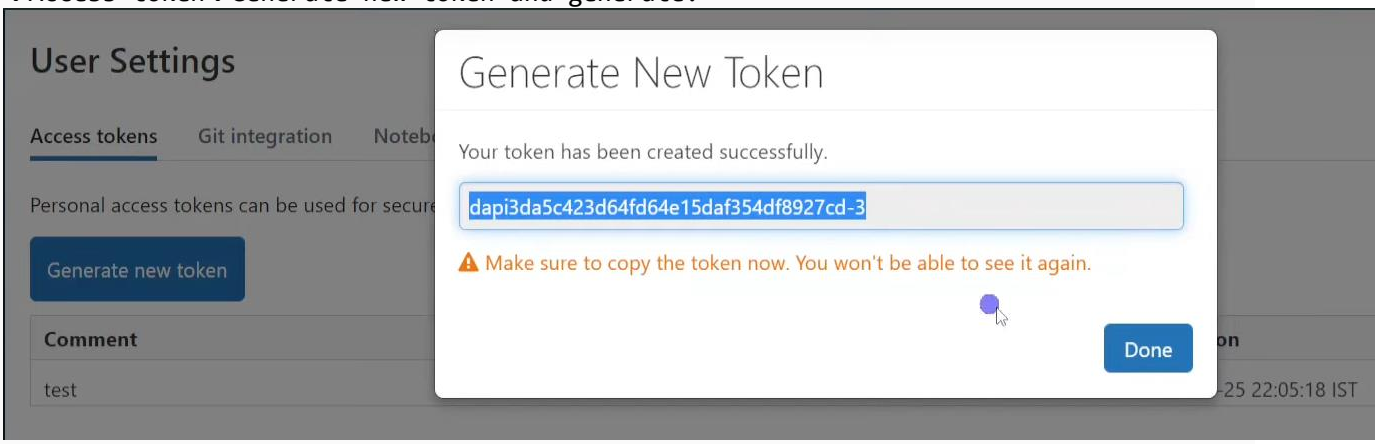
When we have to mix certain activities from ADF and some notebook from databricks in order to achieve some ETL solution then we can go with that which means mix of azure services to achieve target solution. That is one requirement.

Apart from that even though databricks is having workflow in place, it’s not suitable for some scenarios.one scenario could be like event-based loading which means when a

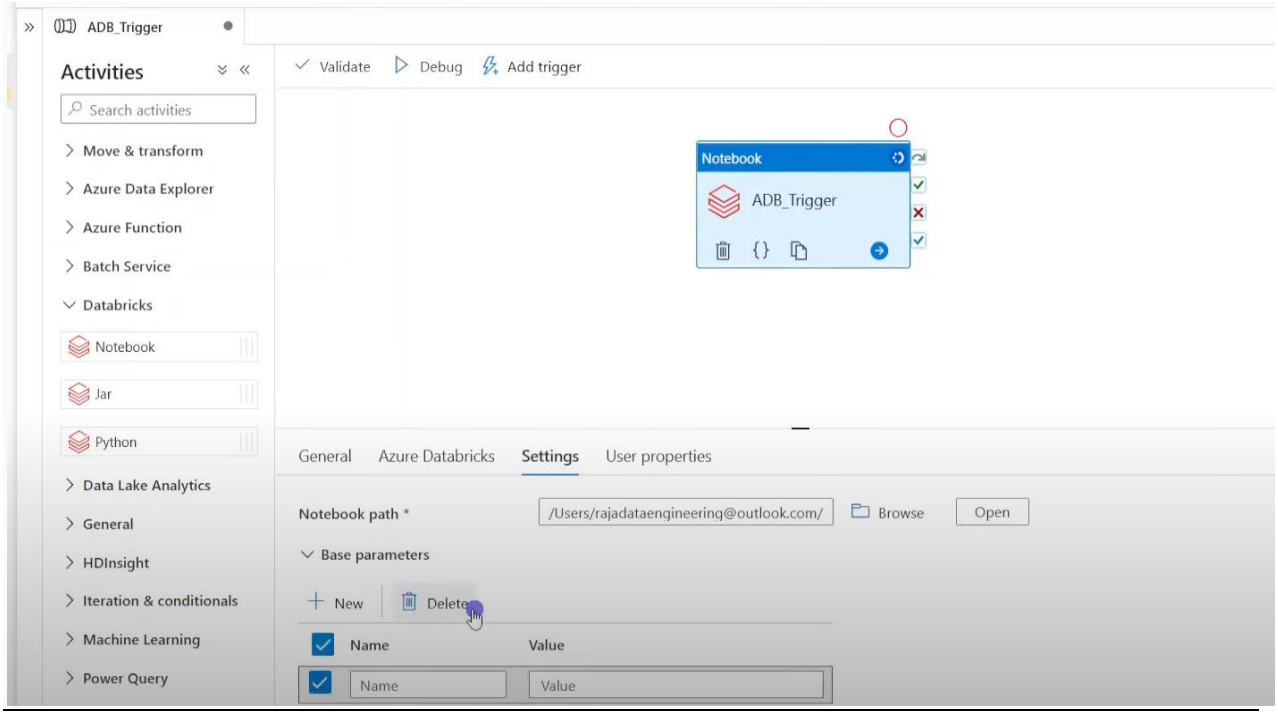
data file is arrived, as soon as notebook should be triggered. So, there are limitations in data bricks but ADF can do this solution because in ADF there are 3 different types of triggers. 1 is event-based trigger which means once the file arrives immediately it will start triggering the activity. That can be leveraged for this requirement. In databricks also there are some solutions which can detect the file as soon as it arrives and it can process one is autoloader but autoloader is more kind of a streaming solution that is not suitable for batch processing. So that is the reason in case if we have to load the data as soon as it arrives using databricks notebook then we can go with ADF integration.

In order to schedule ADB notebook using ADF, we need to have ADB notebook which is performing core ETL job. And in order to connect ADB from ADF, we need to create linked service, once we created linked service for this scheduling we should also choose cluster selection in ADB. In ADF, we will call ADB notebook we have to setup input and output parameters. In case if we are using input and output parameters in our notebook then we need to set up parameters as well. If we don't have any parameters then ip and op parameters are optional.

In ADF, in order to trigger the notebook, we have to use Notebook Activity. After dragging Notebook activity to canvas, we need to create linked service and give necessary details. Linked Service name, notebook name, cluster details and access token etc. While creating linked service, access token will get it from admin→ user settings →Access token→Generate new token and generate.



Incase if we are having any input parameters for our notebook then we can set up those input parameters in base parameters section as shown below.

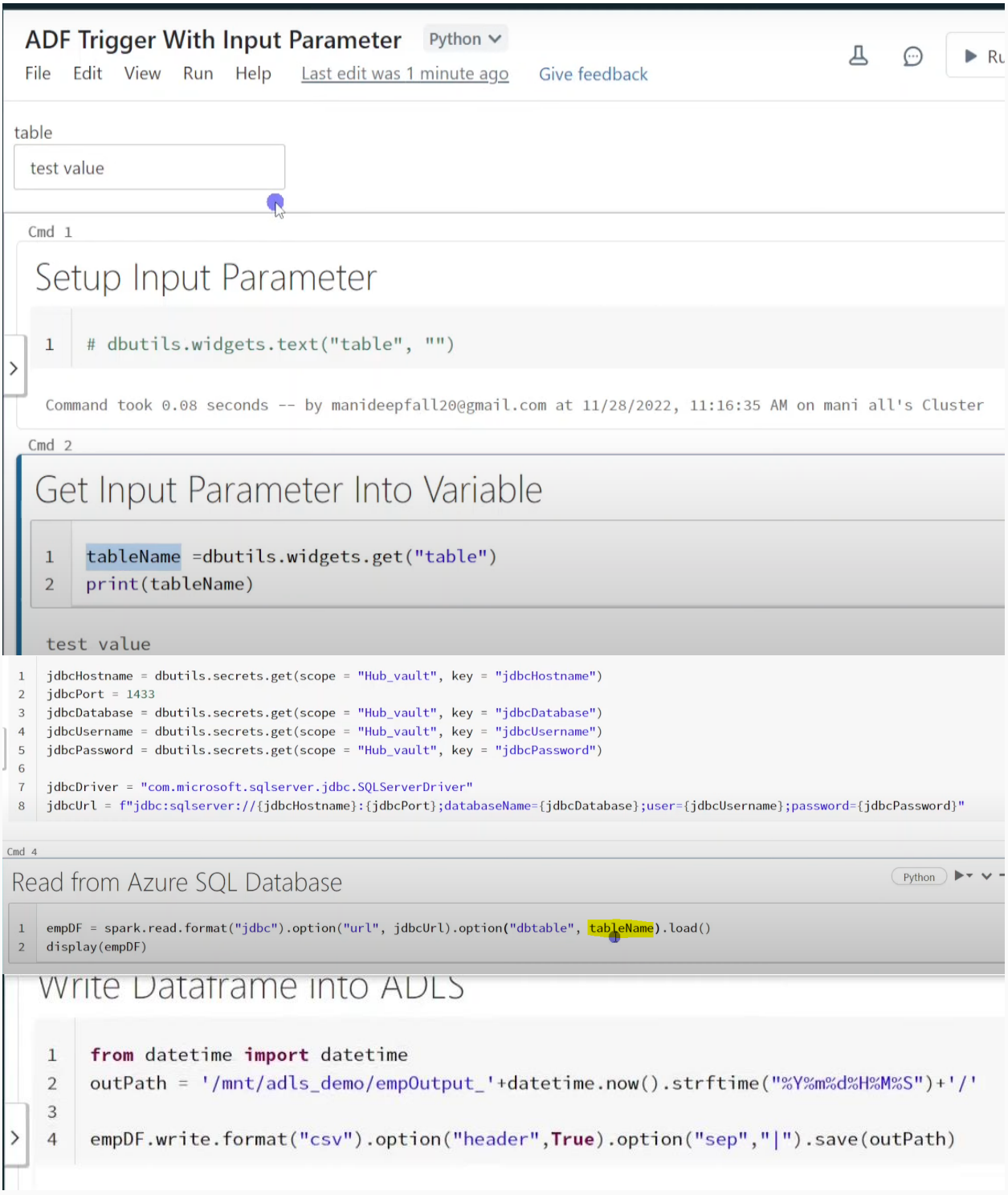


HOW TO RUN DATABRICKS NOTEBOOK ACTIVITY IN ADF WITH INPUT PARAMETER

In order to execute databricks notebook using input parameter, first we need to create input parameter in our databricks notebook using widgets. There are different types of widgets for handling input parameters such as text box, drop down box and combo box.

CODE

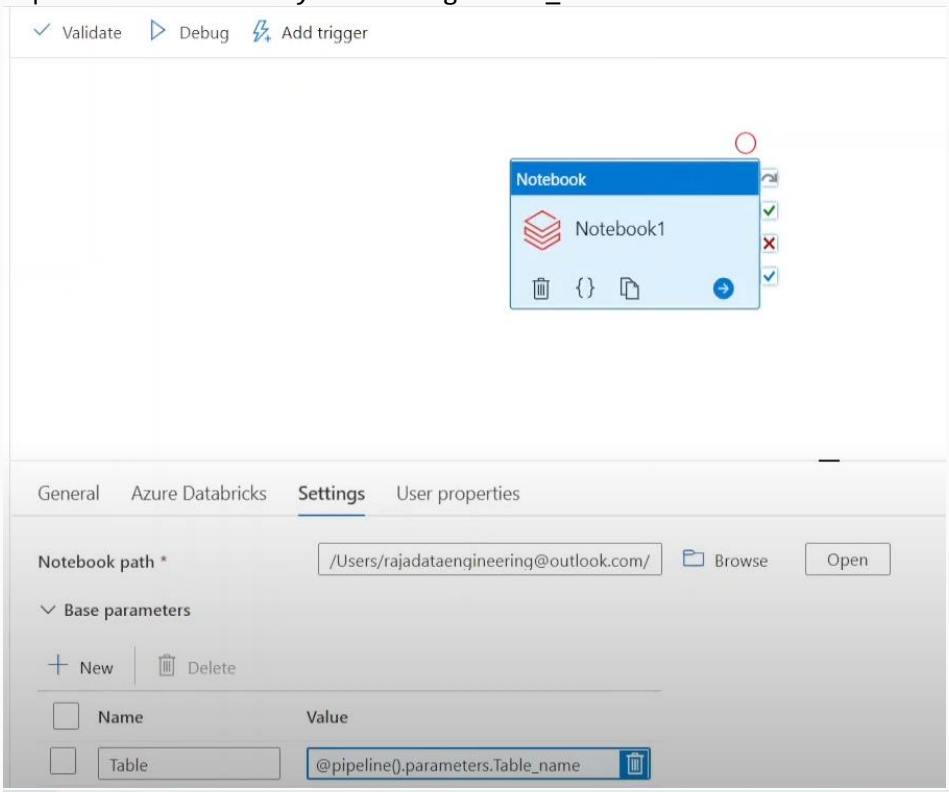
```
#SET UP INPUT PARAMETER
dbutils.widgets.text("table","")   → This is one time activity.

#Get Input Parameter Into Variable
tableName = dbutils.widgets.get("table")
print(tableName)
```

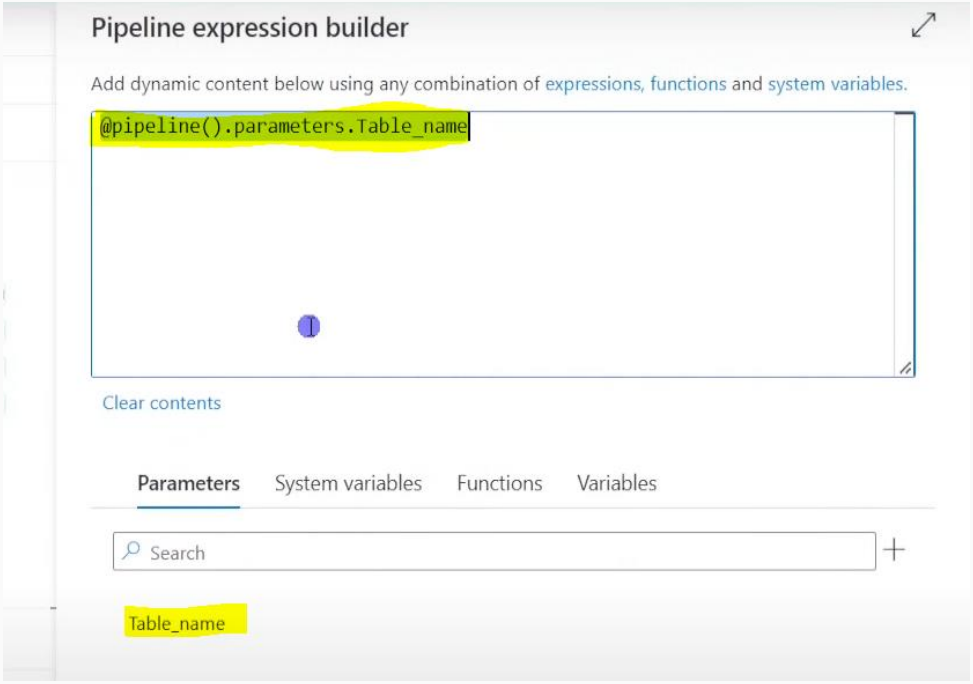


When we execute the ADB notebook through ADF, we will pass table name in table text box and it will come to variable tableName. Till now we created input parameter in databricks side.

Now in ADF, we will drag Notebook Activity to canvas and select required details in linked service and notebook path.

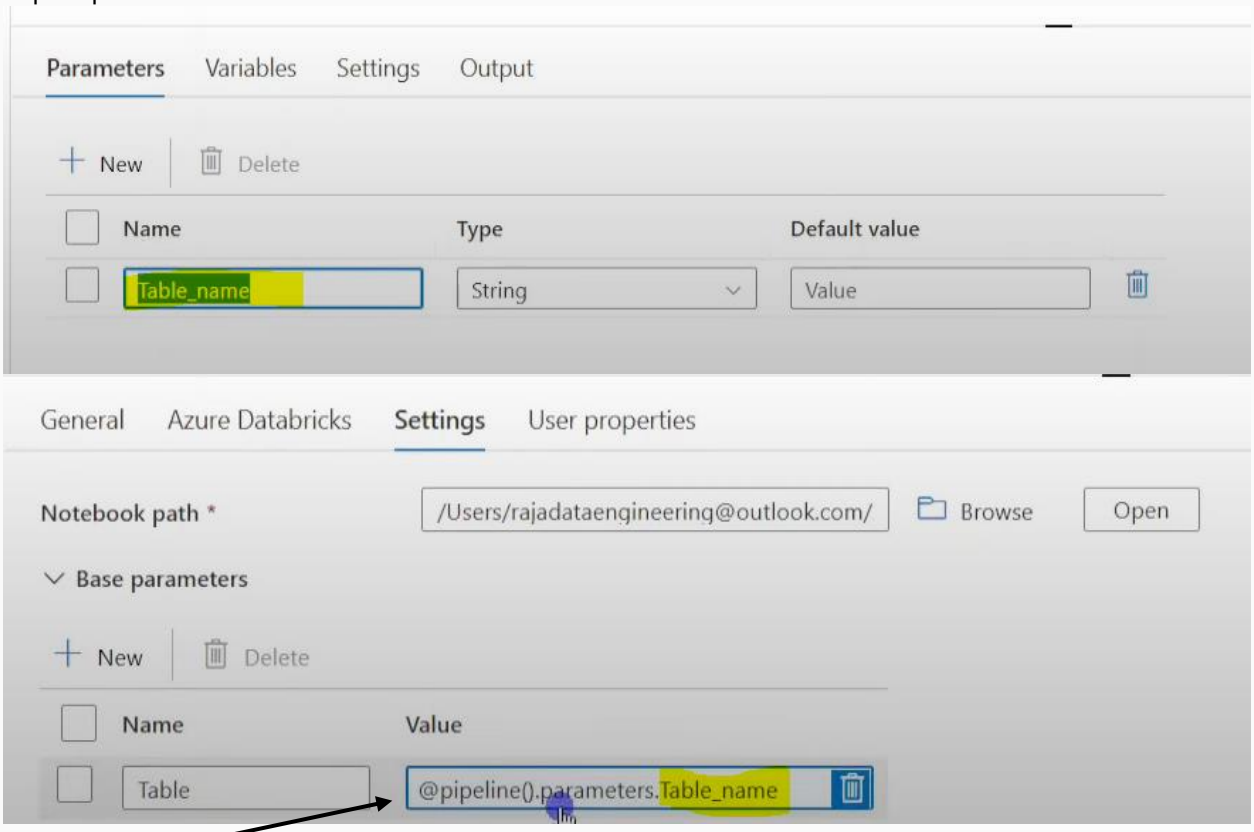
One particular notebook is accepting some input parameters,so for that in ADF notebook activity in base parameters we are giving name as Table and giving value from dynamic expression builder by selecting Table_name.



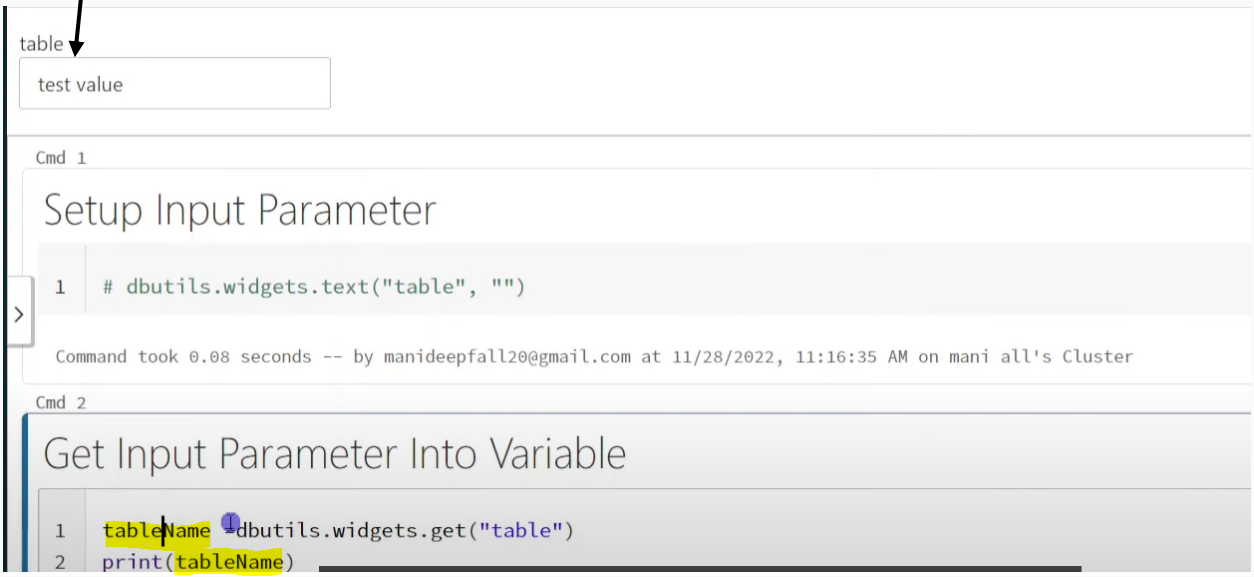


Now if we open linked service, in parameters section we will give Table_name and default value we will not give anything which means whenever we are executing it will

demand the input from the user. This parameter will go and sit in databricks notebook input parameter.

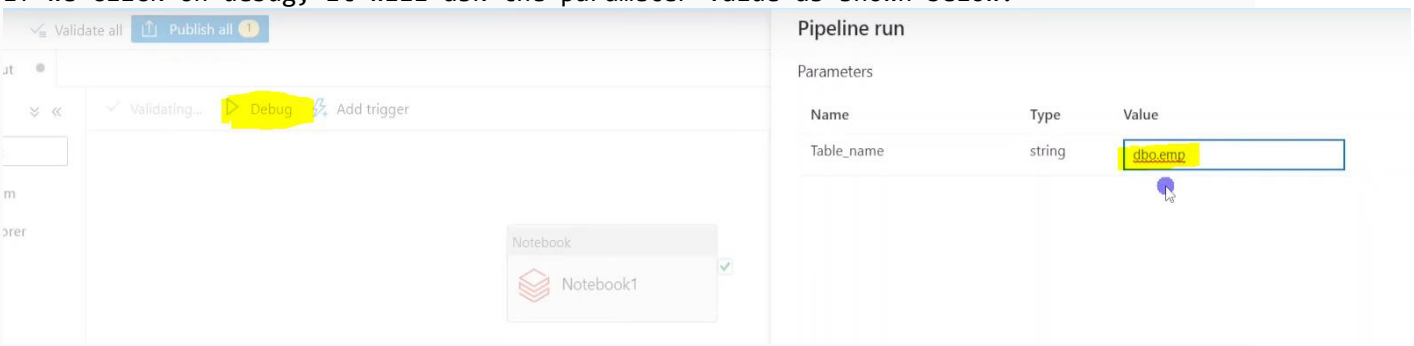


This is how we are passing parameter to databricks notebook. Once we sent value from here, it will call the notebook and the value that was passed there it will come and sit here in databricks notebook and finally it will go to tableName variable



Now we can debug. Debug is mostly used for development purposes. Trigger is used for production purposes for scheduling.

If we click on debug, it will ask the parameter value as shown below.



Basically, this value dbo.emp will be passed to the notebook text widget.

HOW TO GET OUTPUT PARAMETER WITH ADF DATABRICKS NOTEBOOK ACTIVITY?

In order to get output parameter from ADB notebook we have to define the logic to return some value to the caller. Lets say we are calling ADB notebook using ADF. In this case , ADF is the caller and when we execute databricks notebook at the end of execution, it should return some value to the caller using the below syntax.

dbutils.notebook.exit(<value>)

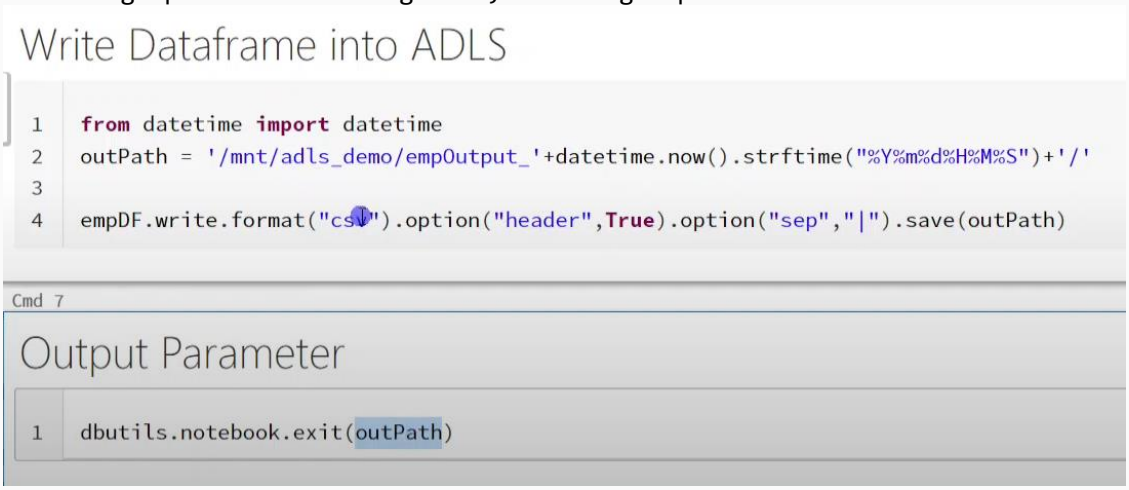
Based on the decision, various other processes can be triggered in the orchestration.so that is the reason this output parameter is important for certain ETL pipelines. So, we need to define output parameter and coming to ADF side we have to collect value from databricks, for that we have to use expression **@activity('ActivityName').output.runOutput** .

Based on these 2 code changes, we can receive output in ADF. So, in real time we will use output value for various other decisions and based on triggering other processes. We need to collect output variable into one of the variable in ADF.

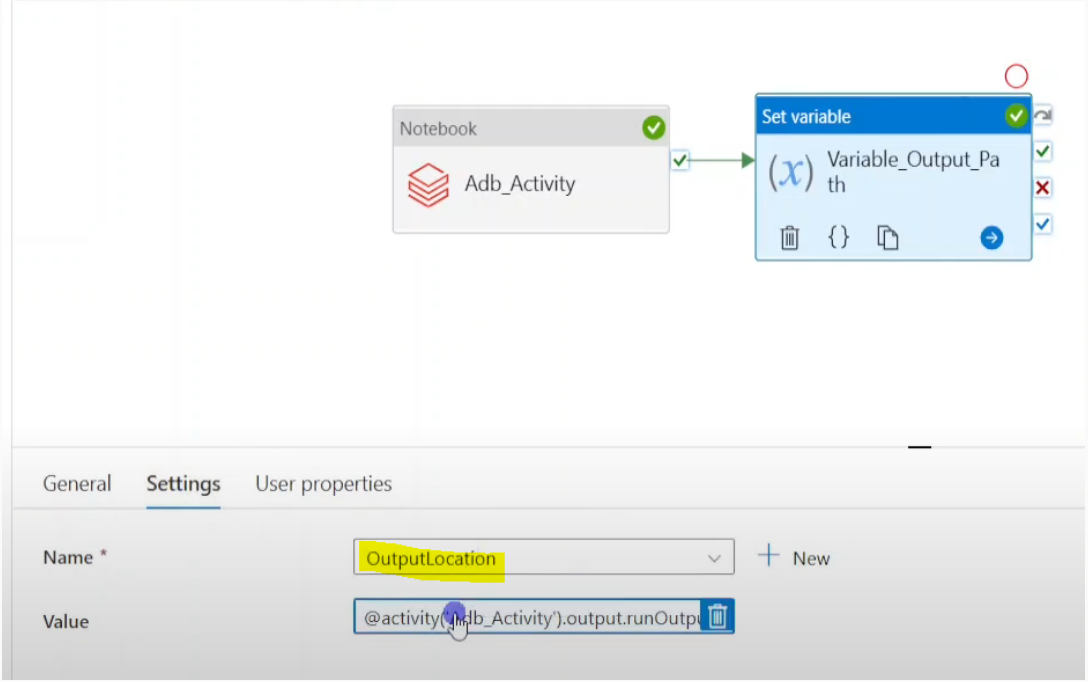
The below screenshot shows this notebook is accepting input parameter table.

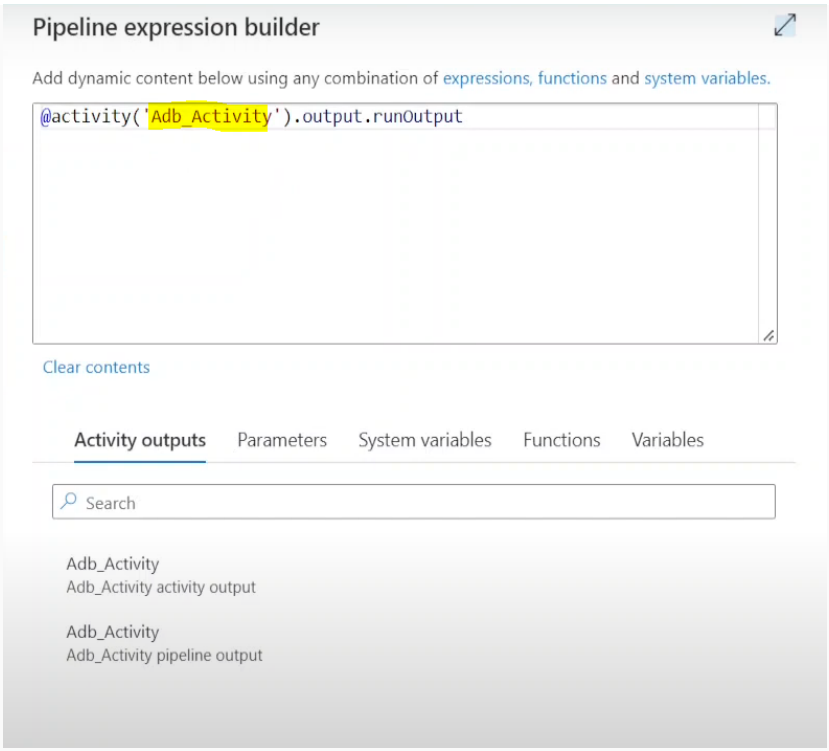
It will read table from Azure SQL DB. Based on jdbc connection and input tableName, df is getting created. Once df is created, we need to write output into ADLS so for that we will create mount point. Once data is returned to output location, in this case

ADLS then we will pass this output parameter to the caller(in this case ADF). So, the output location will be given back to ADF.
Now let's say in ADLS path output location is written. Let's say ADF will read data once again based on adls output location then it will apply some data massage or data cleansing operation removing null, removing duplicate or etc.



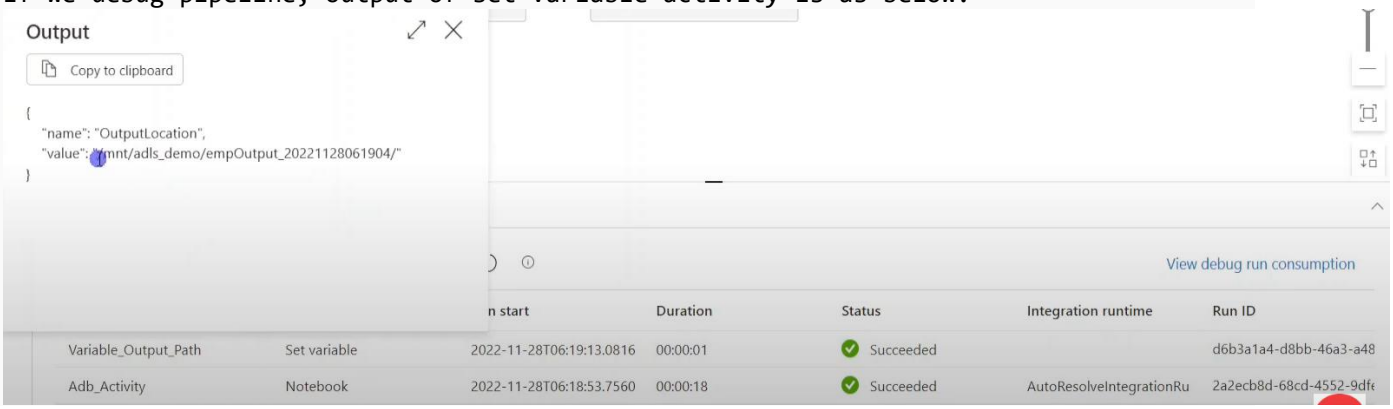
In this case, we will collect this output variable output parameter into one of the variable.Now we will go to ADF, We will create linked service in Notebook Activity and will add input parameter in base parameters. Now we need to collect output from databricks, Once notebook activity is executed in ADB, it is returning some output value so we need to collect that. We can collect that notebook output value using set variable activity. In set variable we are defining one variable that is output location as shown below.





Adb_Activity is previous activity. runOutput from databricks is being captured in the set variable.

If we debug pipeline, output of set variable activity is as below.



This is mount point location. On top of that, we are creating new folder dynamically. This is how we can collect output from databricks activity.

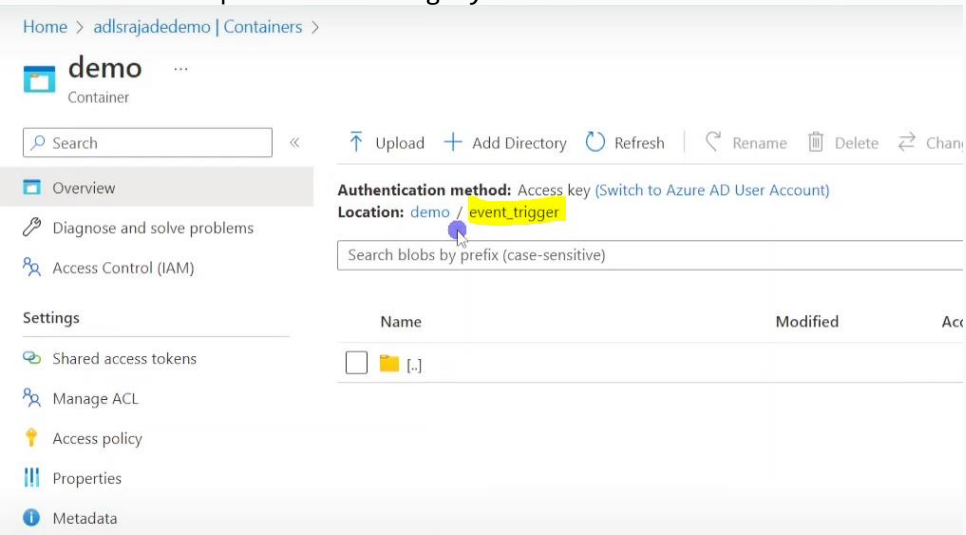
Event based trigger:

Whenever we are going to receive new file within ADLS then by detecting that event , data factory trigger can invoke data factory pipeline. Whenever new file arrives into data lake storage then ADF trigger can detect that event and it can invoke data factory pipeline immediately. Its not only for new file arrival but it could be for file deletion. As soon as event happens then ADF trigger will invoke data factory pipeline. ADF pipeline is calling databricks notebook so finally incoming file will be processed. We are going to receive new file into ADLS so that is the input for databricks notebook

ADB Notebook:

Creating one databricks notebook. That notebook will detect latest or most recent file under a particular data lake storage and from there it will pick latest file then it will create a data frame then it is going to write that output into Azure SQL database. That is the requirement.

We have a data lake storage with container named demo. Within that we have one folder event_trigger. As soon as we receive new file under this folder then immediately that file should be processed using my databricks notebook.

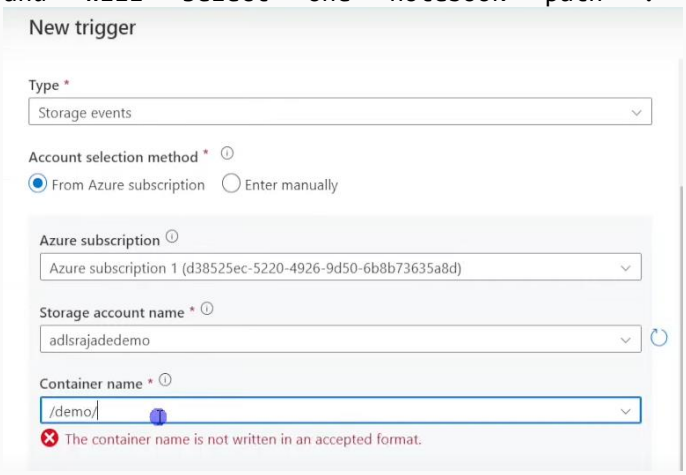


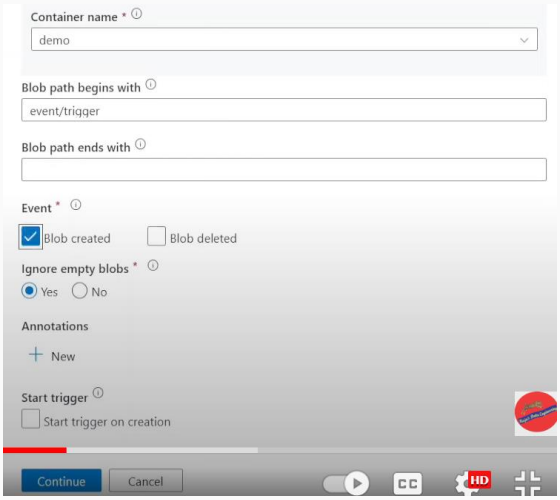
Currently this folder is empty as shown above. Whenever we are going to get new file then my ADF will detect that file that event then it will invoke ADB notebook.

```
#DEFINING THE ROOT FOLDER
root_path = “/mnt/adls_demo/event_trigger/”

#Get Most Recent File
lst = dbutils.fs.ls(root_path)
latest_file = sorted(lst,reverse=True)[0]
print(latest_file)
root_path +=latest_file.name
print(root_path)

#Creating Dataframe for Most Recent File
df = spark.read.option(“sep”,”|”).option(“header”,”true”).csv(root_path)
display(df)
#JDBC CONNECTION DEFINITION
jdbcHostname = dbutils.secrets.get(scope = “akv_secret_demo” , key= ”jdbcHostname”)
jdbcPort = 1433
jdbcDatabase = dbutils.secrets.get(scope= “akv_secret_demo” , key= ”jdbcDatabase”)
jdbcUsername = dbutils.secrets.get(scope =”akv_secret_demo”, key= “jdbcUsername”)
jdbcPassword = dbutils.secrets.get(scope = “akv_secret_demo”, key = “jdbcPassword”)
```

```
jdbcDriver = “com.microsoft.sqlserver.jdbc.SQLServerDriver”
jdbcUrl =
f”jdbc:sqlserver://{jdbcHostname}:{jdbcPort};databaseName={jdbcDatabase};user={jdbcUs
ername};password={jdbcPassword}”

#Writing dataframe into Azure SQL Table
df.write\
      .format(“jdbc”)\
      .option(“url”,jdbcUrl)\
      .option(“dbtable”,”dbo.emp”)\
      .option(“user”,jdbcUsername)\
      .option(“password”,jdbcPassword)\
      .mode(“append”)\
      .save()
```

Now when file arrives, immediately the file will be processed. This is called event-based trigger. Now we go to ADF, we will drag databricks notebook activity to canvas and will select one notebook path . and now we will create trigger

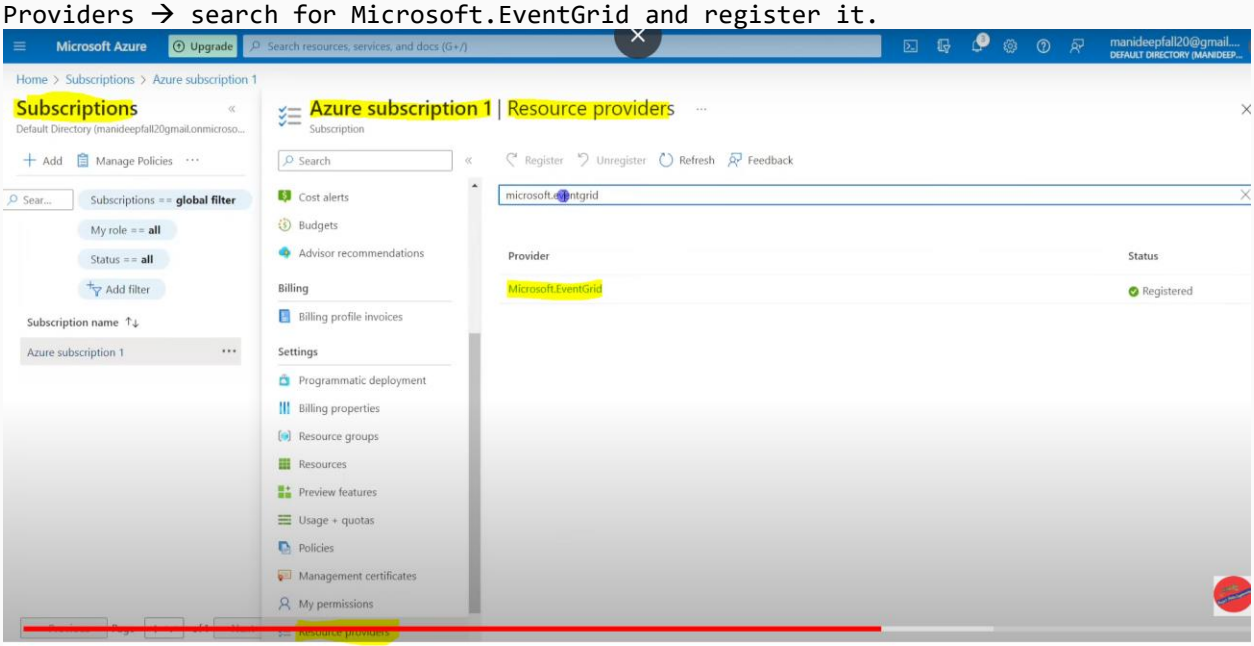




Now event trigger got created.

Before defining this event-based trigger, we have to register event grid within azure subscription.

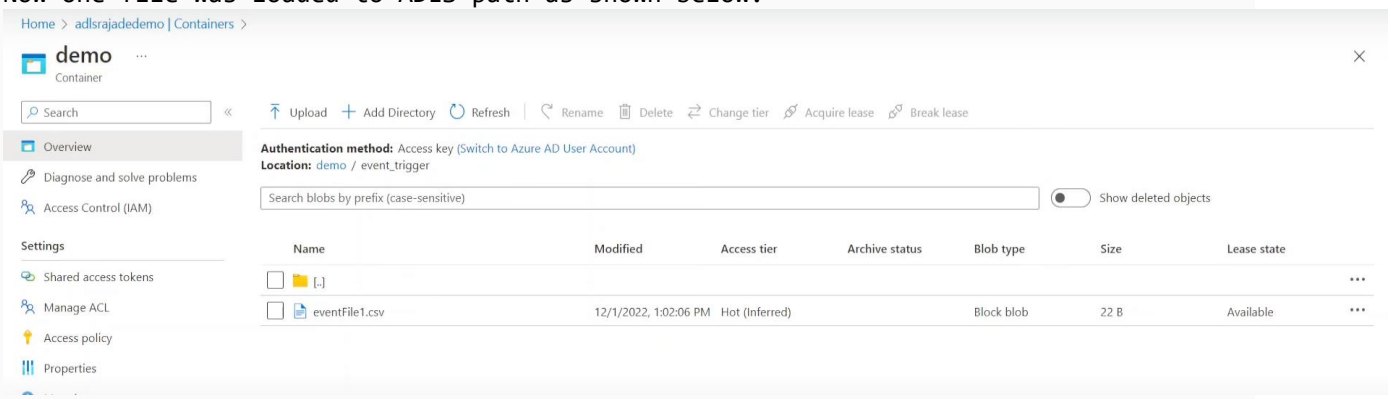
Go to Home → search for subscriptions→ go to our subscription → click on Resource Providers → search for Microsoft.EventGrid and register it.



Now we will attach event-based trigger to our ADF notebook.

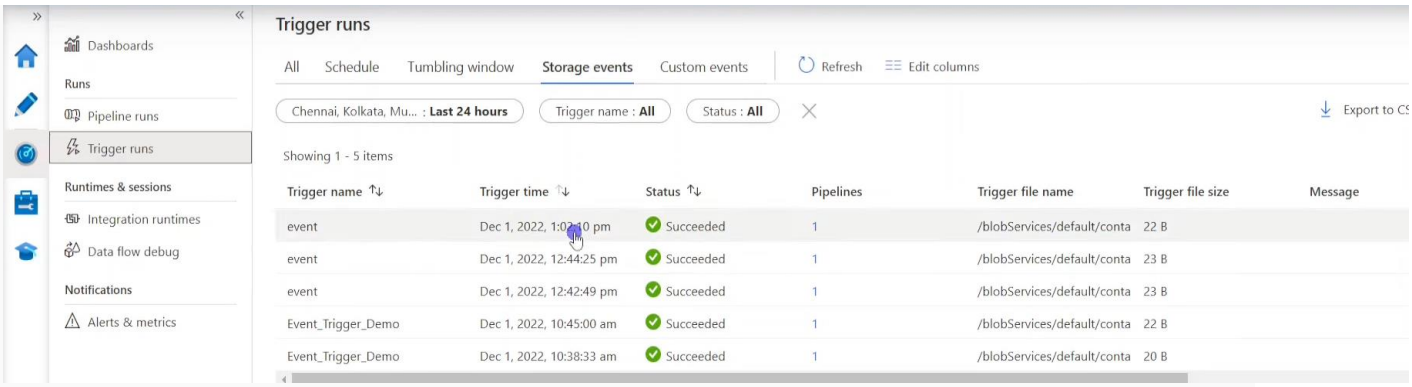
Now we will add one new file in ADLS location, then event will be triggered by ADF pipline where we will have databricks notebook activity which will invoke databricks notebook. This databricks notebook will identify the latest file and it will create data frame and it will write output into Azure SQL database.

Now one file was loaded to ADLS path as shown below.

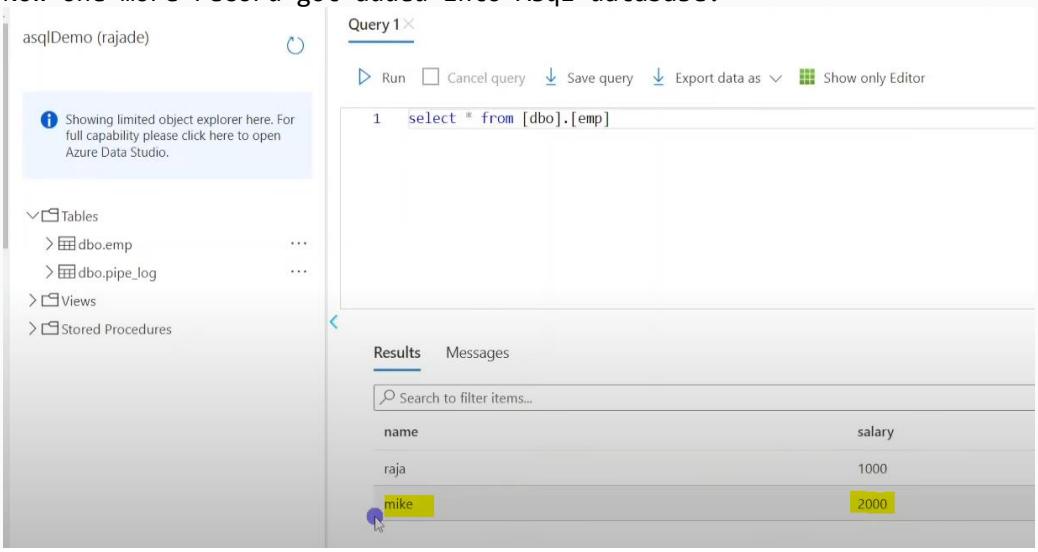


Once file was arrived into ADLS path then ADF trigger would have identified this one then it would have invoked ADF pipeline. ADF pipeline invoked databricks notebook then databricks notebook will read ADLS file even if we are having hundreds of files databricks logic will pick only latest file then that file will be processed and it will write output into Azure SQL database.

Now once file added into ADLS, event got triggered. If we want to monitor it, go to Monitor → Trigger runs → Storage events → event got triggered.



Now one more record got added into Asql database.



Whenever new files arrives into ADLS location, immediately that file will be processed by data bricks and it writes into output location.

HOW TO CREATE DATA FRAME BY READING MULTIPLE EXCEL SHEETS

In some projects, master data is still maintained through excel sheets.

In excel_score_data excel file, we are having 3 sheets.

Sheet1

Id	Name	Score
111	Mike	100
222	David	200

Sheet2:

Id	Name	Score
333	James	300
444	Kevin	400

Sheet3:

Id	Name	Score
555	Nancy	500
666	Blessy	600

Overall excel contains 6 records and 2 records per sheet.
Now we uploaded excel_score_data.xlsx file into DBFS path.

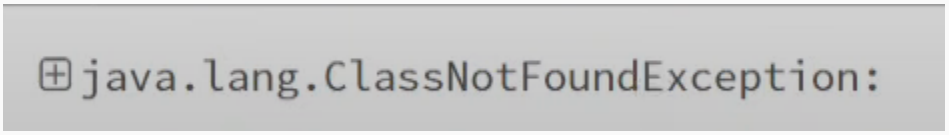
CODE:

```
%fs
ls /FileStore/tables/unit_testing/
```

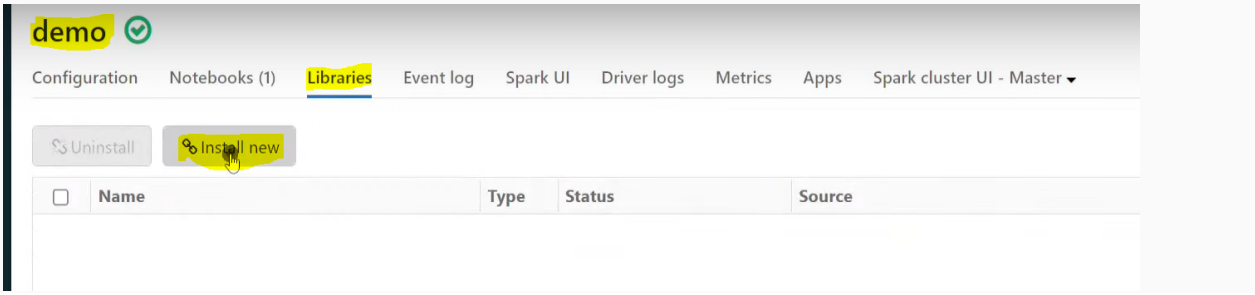
Table +

	path	name	size	modificationTime
1	dbfs:/FileStore/tables/unit_testing/code.csv	code.csv	83	1669350016000
2	dbfs:/FileStore/tables/unit_testing/excel_score_data.xlsx	excel_score_data.xlsx	10464	1669902988000
3	dbfs:/FileStore/tables/unit_testing/product.csv	product.csv	95	1669350016000

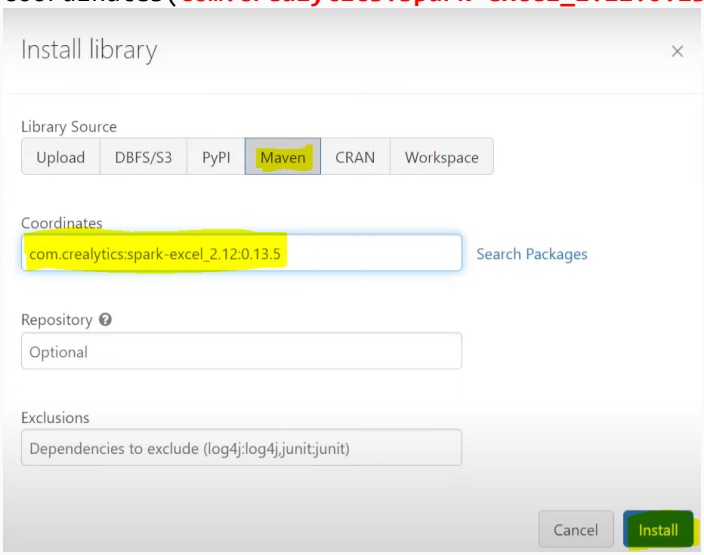
```
#READ SINGLE SHEET OF EXCEL
df =
spark.read.format("com.crealytics.spark.excel").option("inferschema",True).option("header",True).option("dataAddress","sheet3!").load("/FileStore/tables/unit_testing/excel_score_data.xlsx")

display(df)
```



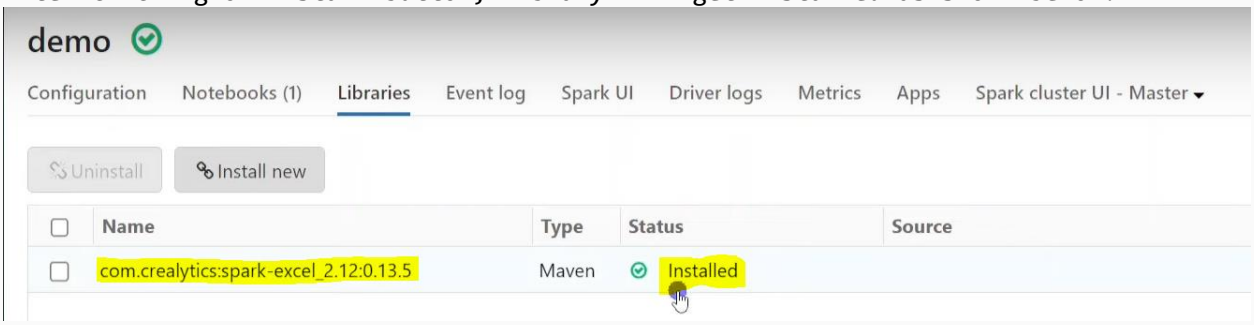
while reading excel file, we can't use this format without library. Before using this format, we have to install some library in our cluster.
We got above error, if we haven't used library in our cluster.
We need to install library **com.crealytics:spark-excel_2.12:0.13.5** at notebook level or at cluster level.
At cluster level, we need to open our created cluster. Go to libraries and click on install new.



Now go to Maven Library source, we will enter that library name in coordinates(com.crealytics:spark-excel_2.12:0.13.5).

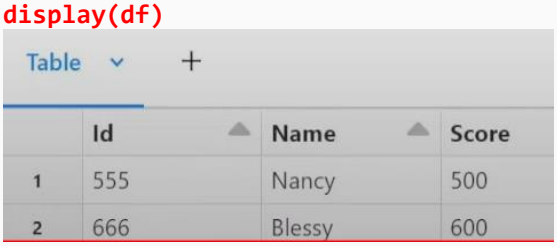


After clicking on install button, library will get installed as shown below.



Now if we run same query again.

```
df = spark.read.format("com.crealytics.spark.excel").option("inferschema",True).option("header",True).option("dataAddress","sheet3!").load("/FileStore/tables/unit_testing/excel_score_data.xlsx")

display(df)
```



If we don't give dataAddress as sheet3, then by default it will give sheet1 data.

```
#DEFINE EXCEL LOCATION AND SHEET NAMES
sheets = ['sheet1','sheet2','sheet3']
path = "/FileStore/tables/unit_testing/excel_score_data.xlsx"
```

In order to get all sheets data , what we can do is creating UDF

```
#UDF TO CREATE DATAFRAME FROM MULTIPLE SHEETS AND PERFORM UNION
def createExcelDataFrame(path,sheets):

  firstSheet = sheets[0]
  df =
spark.read.format("com.crealytics.spark.excel").option("inferschema",True).option("he
ader",True).option("dataAddress",f"{firstSheet}!").load(path)
  schema = df.schema
  for sheet in sheets[1:]:
    sheetDF =
spark.read.format("com.crealytics.spark.excel").schema(schema).option("header",True).
option("dataAddress",f"{sheet}!").load(path)
    df = df.union(sheetDF)
  return df
```

Explanation: we are creating one list that is sheets and also, we are defining the location where we are having that particular excel sheet . these are the parameters we are passing for UDF. Here we are hardcoding no of sheets as sheets = ['sheet1','sheet2','sheet3'] but in real time there could be scenarios where we don't know the no of sheets.so dynamically it could change. In this example, we hardcoded number of sheets. schema=df.schema use is we are inferring schema of first sheet so that we can make sure there is no schema deviation in other sheets.so for that purpose we are just reading schema from the data frame which is created by reading the first sheet. Then we are iterating through the sheets one by one so we have already processed first sheet so that is the reason we have used python slicing list slicing sheets[1:] it starting from 2nd sheet till the end. Iterating sheets one by one and we are hardcoding sheet name in dataAddress then immediately we performed union which means it is combining with the previous result so what happens is whenever we have created initial version we are having 2 records then it is processing 2nd sheet and it is going to combine those 2 records into the previous result so it is performing the union which means we will get 4 records in the 2nd iteration and 3rd iteration it is going to add 2 more records so finally it is going to process all the sheets and its combining all the records into one data frame. Finally, it will return dataframe into caller.

```
#CALL UDF AND DISPLAY OUTPUT DATAFRAME
outDF = createExcelDataFrame(path,sheets)
display(outDF)
```

Table +

	Id	Name	Score
1	111	Mike	100
2	222	David	200
3	333	James	300
4	444	Kevin	400
5	555	Nancy	500
6	666	Blessy	600

DISTINCT VS DROP_DUPLICATES

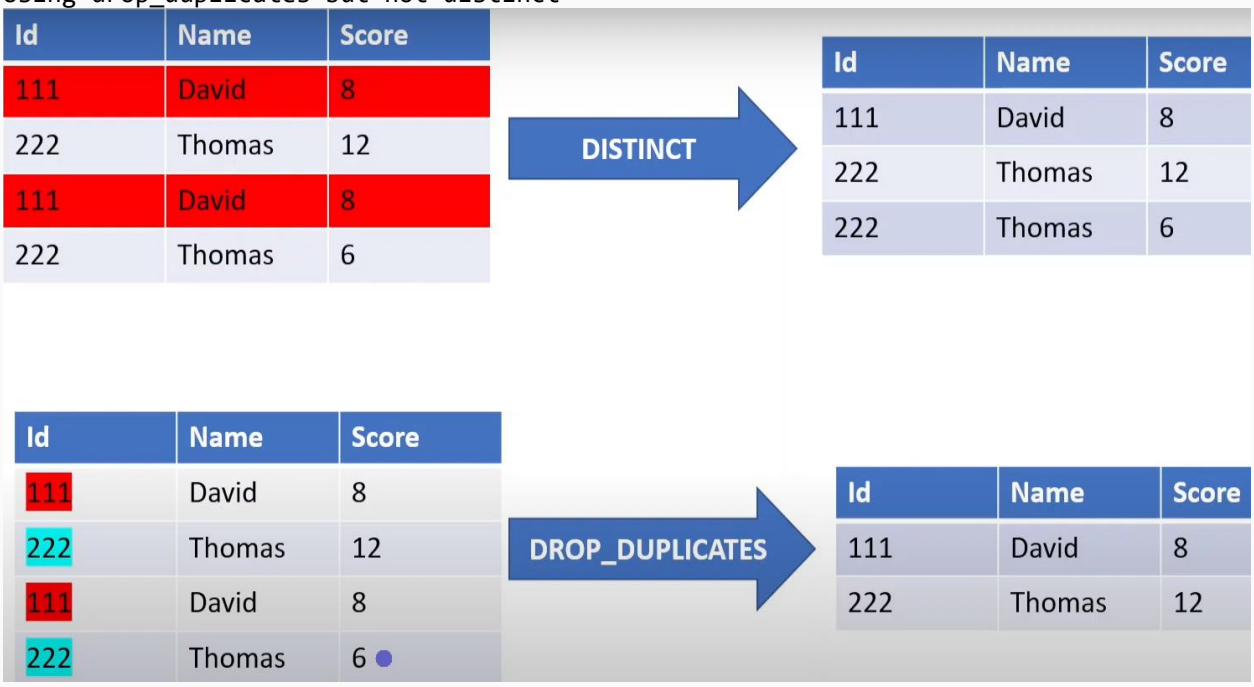
Distinct and drop_duplicates both are used to remove duplicate records out of a dataframe but still there are differences.

- **HOW TO REMOVE DUPLICATE RECORDS IN A DATA FRAME?**

Using distinct and drop_duplicates

- **HOW TO REMOVE DUPLICATE RECORDS IN A DATA FRAME ONLY BASED ON CERTAIN COLUMNS?**

Using drop_duplicates but not distinct



Lets say we are having dataframe with 3 columns and 4 records.

DISTINCT : Whenever we are going to consider complete list of columns then we are having duplicate only for this record 111 at the same time 2 records id with 222 can't be considered as duplicate because it is varying value in the last column Score.

Let's imagine we are going to consider subset of column list which means out of 3 columns , we are going to consider only two columns Id and Name then in that case output will be as 2 records only.

Id	Name
111	David
222	Thomas

Distinct basically doesn't support subset. So, whenever we are going to apply distinct on top of above dataframe, we got 3 records. Coming to drop_duplicates, we can give subset, which means let's say column as Id and Name , then we will get only 2 records in above dataframe. But in the result, it will give all the columns, it is not going to eliminate any column even though we are giving subset of ID and name. coming to Id 222 and Thomas with score 12 and 6, it will randomly pick one data , here it could be either 12 or 6.
So, whenever we are having multiple partitions and data is distributed across multiple partitions so it will pick one value randomly so in this case example it took 12.

DROP: Drop is used mainly to get uniqueness of complete row so no parameter needed so we cannot remove duplicates based on a subset. Still there is a workaround to consider the subset but that is coming with shortcomings as well.
DROP_DUPLICATES: This is similar to distinct functionality by default so whenever we are not giving any parameter then it will act as a distinct.it will identify uniqueness of a complete row by default. This will support subset elimination which means we can give subset of columns , duplicate records can be identified only within that subset of columns.
Syntax: dropDuplicates(subset=None)
CODE:
CREATE SAMPLE DATAFRAME

```
simpleData = ((111,"James",8),\
       (222,"Mike",5), \
       (111,"James",8), \
       (222,"Mike",12) \
)
columns =["id","name","score"]
df = spark.createDataFrame(simpleData,columns)
df.display()
```

	id	name	score
1	111	James	8
2	222	Mike	5
3	111	James	8
4	222	Mike	12

#DISTINCT TO REMOVE DUPLICATE

```
df.distinct().display()
```

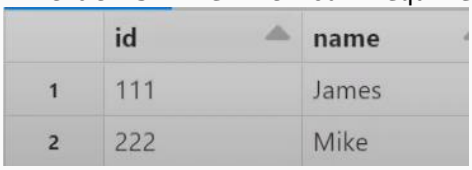
	id	name	score
1	111	James	8
2	222	Mike	5
3	222	Mike	12

DISTINCT WITH SUBSET

Incase if we have to use distinct only for the subset of column then using select statement, we have to select only those columns , we want to remove duplicate for subset of Id and name

df.select("id","name").distinct().display()

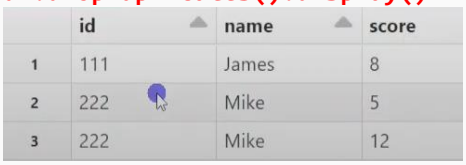
In distinct if we want to apply subset then we have to lose some columns , we need to compromise some columns in the end result. So that's not suitable in all the use cases. If that is fine with our requirement, we can go with this workaround.



	id	name
1	111	James
2	222	Mike

DROP DUPLICATE WITH OUT SUBSET

df.dropDuplicates().display()

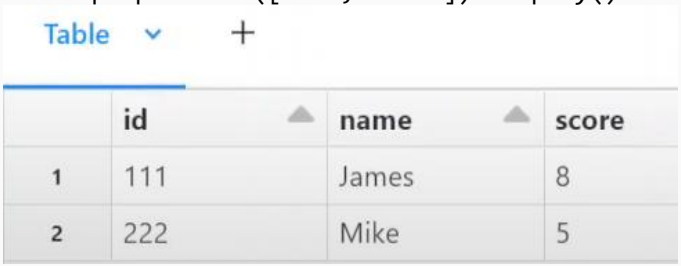


	id	name	score
1	111	James	8
2	222	Mike	5
3	222	Mike	12

distinct and drop_duplicates both are same when we are not giving any input parameter.

DROP DUPLICATE WITH SUBSET

df.dropDuplicates(['id','name']).display()



	id	name	score
1	111	James	8
2	222	Mike	5

Score for 222 id taken randomly and it displayed as 5.

SELECT VS WITHCOLUMN

This is also one of the performance optimization technique. Withcolumn is used to add compute columns on top of existing data frames. Select function is used to select list of columns but using select we can add new computed columns also. So if we got a new requirement to add a new column we can use either withcolumn or select but which function we are going to choose based on that there would be difference in the performance

https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.DataFrame.withColumn.html

calling withColumn multiple times, for instance, via loops in order to add multiple columns can generate big plans which can cause performance issues and

even *StackOverflowException*. To avoid this, use **select()** with the multiple columns at once.

So, if we are going to use withColumn multiple times, it will not only cause performance issues. It might also lead to stack overflow exception error as well.

Let's say there is a data frame which is containing 4 columns, now we have to add 4 more columns so using withColumn we can add 4 more columns. So, if we add 4 more columns using withColumn function, what happens is each withColumn will create different dataframe internally because as per spark architecture, dataframes are immutable which means we cannot change existing data frame.so whenever we are adding new column using withColumn what happens is internally it is creating new data frame even though we are not giving explicit name for the dataframe but internally it will create a new data frame. So, let's say we are adding 4 columns which means 4 different dataframes will be created internally one by one.Let us assume in withColumn we are using certain complex logic. Let's say we are having window function. window function is one of the costliest operation because it is shuffling data across executors.

Whenever we have to add new column based on window function, it is going to shuffle the data across executors but whenever we are going to add data multiple times i.e.; multiple columns let's say in this case 4 columns which means 4 different dataframes will be created which would require 4 times shuffle operation so it will be costly. So, this is the reason for performance bottleneck in withColumn. But on the other hand , if we are adding all 4 columns in one single group using select function which means it is going to create only one dataframe internally. For that it is going to shuffle the data only one time and it will add all the computed columns on top of that in single code so that is the reason we are avoiding unnecessary shuffle operations in select function. That is the reason select performs better when compared to withColumn.so that is the reason we have to go with select instead of withColumn wherever possible. So incase we cannot achieve solution using select then we can go with withColumn but we need to avoid withColumn as much as possible.

SAMPLE DATAFRAME

```
employee_data =
[(111,"Stephen","King",2000),(222,"Philipp","Larkin",8000),(333,"John","Smith",6000)]
employee_schema = ["Id","FirstName","LastName","salary"]
df = spark.createDataFrame(employee_data,employee_schema)
display(df)
```

Table +

	Id	FirstName	LastName	salary
1	111	Stephen	King	2000
2	222	Philipp	Larkin	8000
3	333	John	Smith	6000

ADD NEW COLUMNS WITH MULTIPLE WITHCOLUMN

```
from pyspark.sql.functions import col,concat,lit,current_timestamp
dfWithColumn = df.withColumn("Name",concat(col("FirstName"),lit("
"),col("LastName"))) \
.withColumn("BonusPercent",lit(10)) \
.withColumn("TotalSalary",col("salary")*col("BonusPercent")) \
.withColumn("DateCreated",current_timestamp())

display(dfWithColumn)
```

Table +

	Id	FirstName	LastName	salary	Name	BonusPercent	TotalSalary	DateCreated
1	111	Stephen	King	2000	Stephen King	10	20000	2022-12-13T15:51:34.168+0000
2	222	Philipp	Larkin	8000	Philipp Larkin	10	80000	2022-12-13T15:51:34.168+0000
3	333	John	Smith	6000	John Smith	10	60000	2022-12-13T15:51:34.168+0000

we can see 10ths or 100's of withColumns but that is not good for performance. Whenever we are creating new column for each column we are creating explicitly new data frame. Apart from that we has one more approach as shown below. Whenever we are creating new column for each column we are creating explicitly new data frame as shown below.

ADD NEW COLUMNS WITH MULTIPLE DATAFRAMES

```
from pyspark.sql.functions import col,concat,lit,current_timestamp
dfWithCol = df.withColumn("Name",concat(col("FirstName"),lit(" "),col("LastName")))
dfWithCol = dfWithCol.withColumn("BonusPercent",lit(10))
dfWithCol = dfWithCol.withColumn("TotalSalary",col("salary")*col("BonusPercent"))
dfWithCol = dfWithCol.withColumn("DateCreated",current_timestamp())
```

Table +

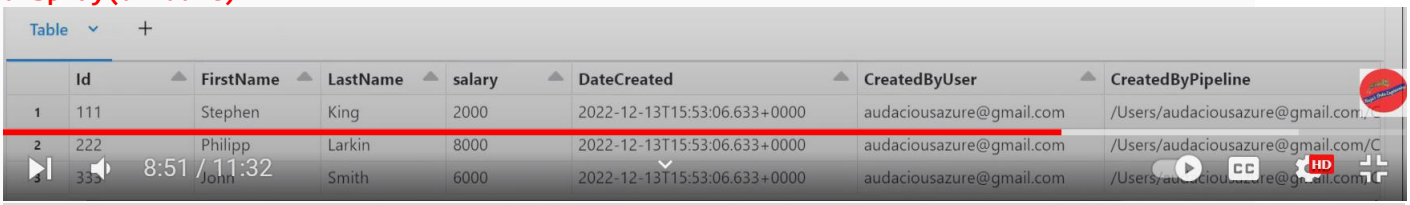
	Id	FirstName	LastName	salary	Name	BonusPercent	TotalSalary	DateCreated
1	111	Stephen	King	2000	Stephen King	10	20000	2022-12-13T15:51:34.168+0000
2	222	Philipp	Larkin	8000	Philipp Larkin	10	80000	2022-12-13T15:51:34.168+0000
3	333	John	Smith	6000	John Smith	10	60000	2022-12-13T15:51:34.168+0000

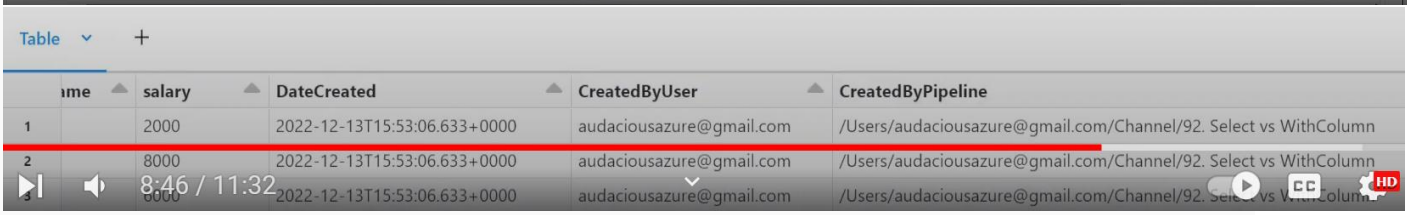
New columns can also be added through UDF. For example, in certain projects we have to add audit columns at the end of each dataframe. Audit column means it is kind of date created.at what time that particular record got created and who created so that information can be taken from dbutility function as shown below and which notebook created that can also be taken from db utility function. These information's are used for tracking or logging purpose later. That's called audit columns. In some projects audit columns are adding using UDF

ADD NEW COLUMNS THROUGH UDF

```
def addAuditCols(df):
  from pyspark.sql.functions import col,lit,current_timestamp
```

```
df = df.withColumn("DateCreated",current_timestamp()) \

.withColumn("CreatedByUser",lit(dbutils.notebook.entry_point.getDbutils().notebook().
getContext().userName().get())) \
.withColumn("CreatedByPipeline",lit(dbutils.notebook.entry_point.getDbutils().noteboo
k().getContext().notebookPath().get()))

return df

dfAudit = addAuditCols(df)
display(dfAudit)
```






```
SELECT VS WITHCOLUMN
from pyspark.sql.functions import col,concat,lit,current_timestamp
dfSelect = df.select("*",concat(col("FirstName"),lit("
"),col("LastName")).alias("Name")
,lit(10).alias("BonusPercent")
,(col("salary")*lit(10)).alias("TotalSalary")
,current_timestamp().alias("DateCreated"))
display(dfSelect)

df.select("*",)→ here * means it will select all existing columns, then on top of
that we are adding 4 columns
#SELECT VS WITHCOLUMN
dfWithColumn = df.withColumn("Name",concat(col("FirstName"),lit("
"),col("LastName"))) \
.withColumn("BonusPercent",lit(10)) \
.withColumn("TotalSalary",col("salary")*col("BonusPercent")) \
.withColumn("DateCreated",current_timestamp())

display(dfWithColumn)
```

Table +

	Id	FirstName	LastName	salary	Name	BonusPercent	TotalSalary	DateCreated
1	111	Stephen	King	2000	Stephen King	10	20000	2022-12-13T15:55:41.277+0000
2	222	Philipp	Larkin	8000	Philipp Larkin	10	80000	2022-12-13T15:55:41.277+0000
3	333	John	Smith	6000	John Smith	10	60000	2022-12-13T15:55:41.277+0000

Showing all 3 rows. | 2.00 seconds runtime

Table +

	Id	FirstName	LastName	salary	Name	BonusPercent	TotalSalary	DateCreated
1	111	Stephen	King	2000	Stephen King	10	20000	2022-12-13T15:55:42.316+0000
2	222	Philipp	Larkin	8000	Philipp Larkin	10	80000	2022-12-13T15:55:42.316+0000
3	333	John	Smith	6000	John Smith	10	60000	2022-12-13T15:55:42.316+0000

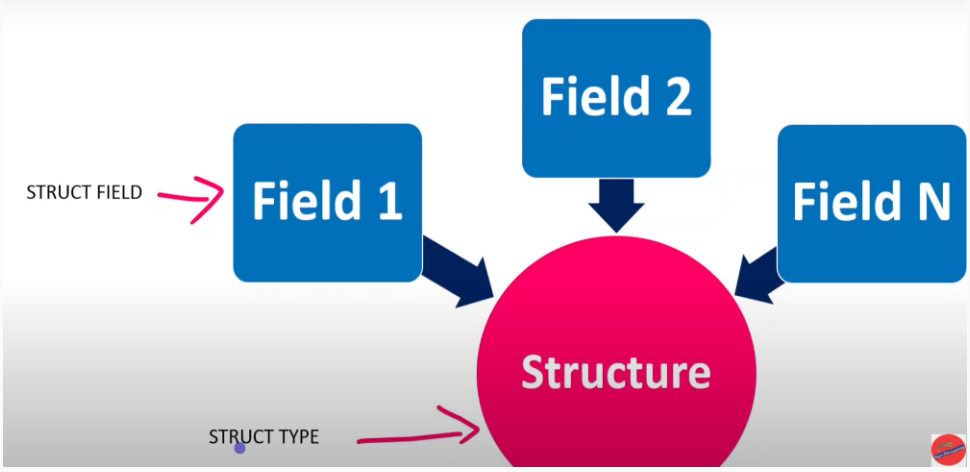
Both select and withColumn producing same output but with huge difference of performance with select statement.

STRUCT TYPE VS STRUCT FIELD

These are used to define schema of a particular dataframe.

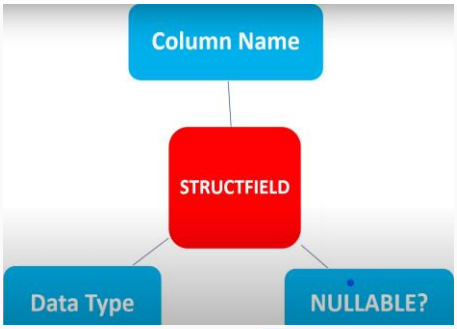
Initially spark provided only one API that is rdd but that is not suitable for high performance because it is not having any schema its not having any proper structure. As a result, it is performing poor so that's the reason, next API dataframe came into picture. Data frame is a schema-based structure so we must define a structure for the data frame then based on that schema, performance optimization can be applied.so there is one engine called catalyst optimizer. Catalyst optimizer not available in rdd.

We need to define structure of a data frame for that we have to use Struct Type. Struct Type is nothing but it is defining one particular dataset so that dataset can be one complete record row in a data frame or nested column



Basically Structtype is defining structure of a data frame and Struct Type is collection of struct fields. Struct Field is within that data structure how to define one particular column. Struct Field is defining one particular field in the row in the data set.

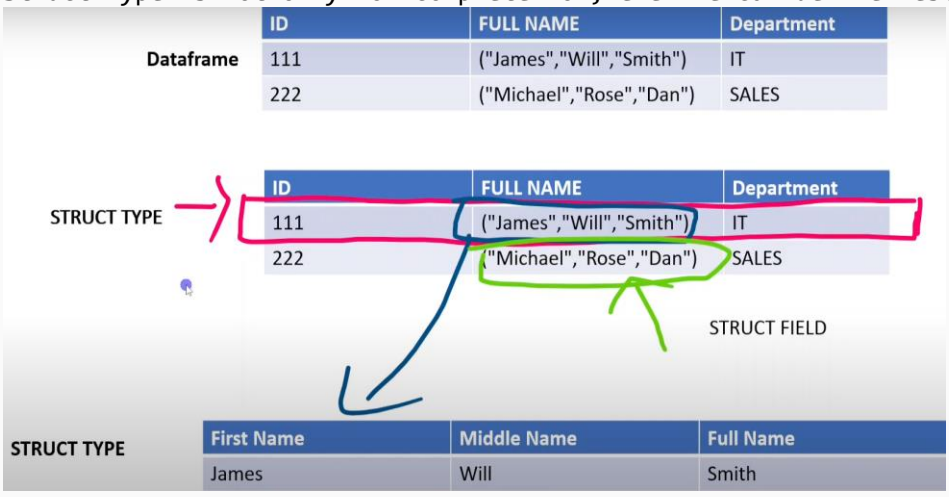
Struct Field will accept 3 parameters, column name and data type and nullable or not?



Dataframe

ID	NAME	Department
111	James	IT
222	Michael	SALES

Struct Type is not only for complete row, even we can define nested columns



CODE:

SAMPLE DATA

```
structureData =
[("James",111,"HR"),("Michael",222,"IT"),("Robert",333,"SALES"),("Maria",444,"IT"),("
Jen",555,"HR")]
```

DEFINE STRUCTURE USING STRUCT TYPE AND STRUCT FIELD

```
from pyspark.sql.types import StructType,StructField,StringType,IntegerType
structureSchema = StructType([
StructField('Name',StringType(),True),
StructField('ID',IntegerType(),True),
StructField('Department',StringType(),True)
])
```

The first parameter is column name, 2nd parameter is data type and 3rd parameter is nullable or not . True means it can accept null values, if we are going to keep false, it will not accept null values.

CREATE DATA FRAME BASED ON STRUCTURE DEFINITION

```
df = spark.createDataFrame(structureData,structureSchema)
df.printSchema()
df.display()
```

```
root
 |-- Name: string (nullable = true)
 |-- ID: integer (nullable = true)
 |-- Department: string (nullable = true)
```

Table +

	Name	ID	Department
1	James	111	HR
2	Michael	222	IT
3	Robert	333	SALES
4	Maria	444	IT
5	Jen	555	HR

NESTED DATA

```
structureData = [
(("James","Will","Smith"),111,"HR"),
(("Michael","Rose","Dan"),222,"SALES"),
(("Robert","Ray","Williams"),333,"IT"),
(("Maria","Anne","Jones"),444,"IT"),
(("Jen","Mary","Brown"),555,"HR")
]
```

DEFINE NESTED STRUCTURE

```
structureSchema = StructType([
StructField('Name',StructType([
      StructField('FirstName',StringType(),False),
      StructField('MiddleName',StringType(),True),
      StructField('LastName',StringType(),True)
])),
StructField('ID',IntegerType(),True),
StructField('Department',StringType(),True)
])
```

CREATE DATAFRAME USING NESTED STRUCTURE

```
dfNested = spark.createDataFrame(structureData,structureSchema)
dfNested.printSchema()
dfNested.display()
```

```
root
 |-- Name: struct (nullable = true)
 |    |-- FirstName: string (nullable = false)
 |    |-- MiddleName: string (nullable = true)
 |    |-- LastName: string (nullable = true)
 |-- ID: integer (nullable = true)
 |-- Department: string (nullable = true)
```

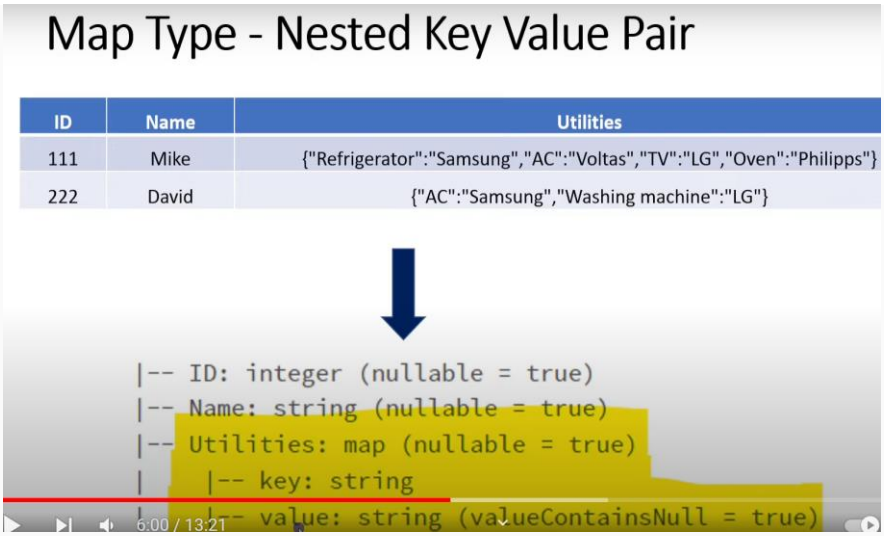
Table +

	Name	ID	Department
1	{"FirstName": "James", "MiddleName": "Will", "LastName": "Smith"}	111	HR
2	{"FirstName": "Michael", "MiddleName": "Rose", "LastName": "Dan"}	222	SALES
3	{"FirstName": "Robert", "MiddleName": "Ray", "LastName": "Williams"}	333	IT
4	{"FirstName": "Maria", "MiddleName": "Anne", "LastName": "Jones"}	444	IT
5	{"FirstName": "Jen", "MiddleName": "Mary", "LastName": "Brown"}	555	HR

STRUCT TYPE vs MAP TYPE

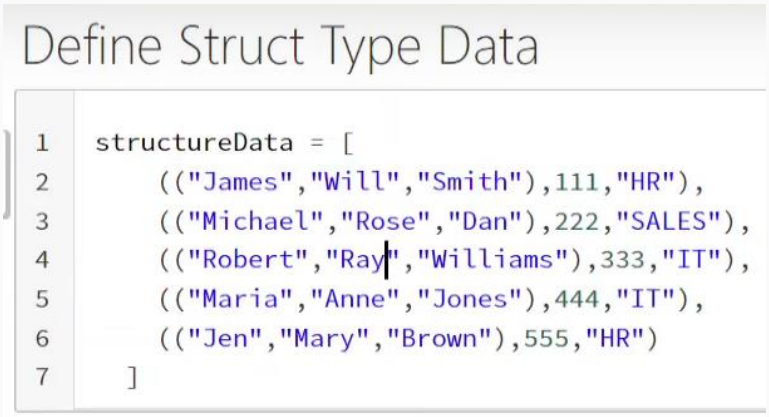
Struct Type is used to define structure of a particular record in the dataframe. Struct Field is used to define structure of a particular field. Collection of struct fields form struct type , that is entire structure of a particular data set, one record. Struct Type is not only used for defining data structure even it is used to define nested columns.

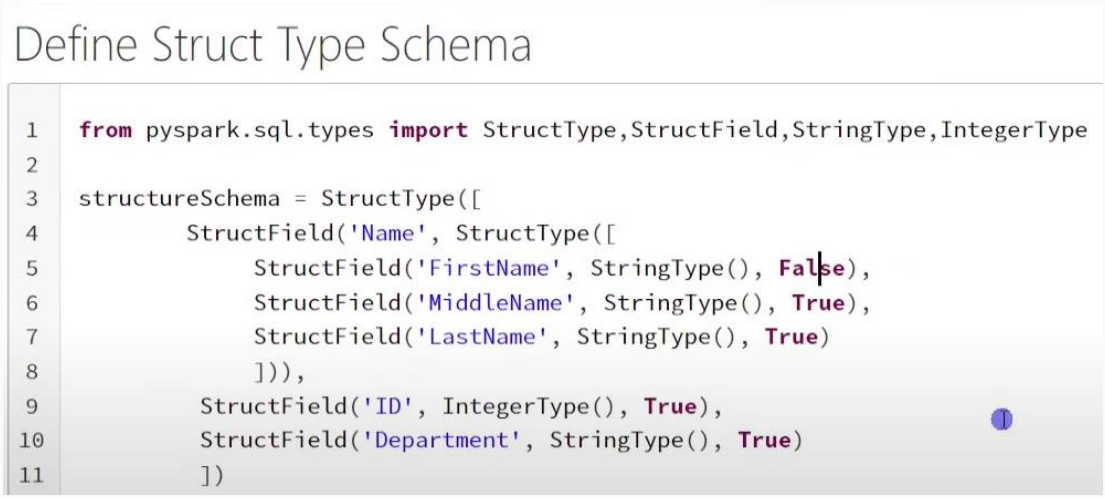
In real time, one record may contain first name and last name. for another record we are missing with middle name. But whenever we are using structure using StructType that's not possible. So, for that we will go with Map Type. In Map Type, we can add any no of key value pairs, it is unbounded and also we can add some extra column or we can remove some extra key value pairs.



StructType: When we know the structure clearly then we can go with StructType and all the records should follow that particular structure. If there is a mismatch then it will throw error. Coming to MapType, simply we are telling key value pair. Any number of key value pairs we can add, we can remove so there is no standard number of keys between the records. This is the major difference between struct type and map type.

CODE:





Create Dataframe With Nested Columns of Struct Type

```
1  df = spark.createDataFrame(data=structureData,schema=structureSchema)
2  df.printSchema()
3  df.display()
```

Cancel Running command...

(3) Spark Jobs

```
root
 |-- Name: struct (nullable = true)
 |    |-- FirstName: string (nullable = false)
 |    |-- MiddleName: string (nullable = true)
 |    |-- LastName: string (nullable = true)
 |-- ID: integer (nullable = true)
 |-- Department: string (nullable = true)
```

Table +

	Name	ID	Department
1	{"FirstName": "James", "MiddleName": "Will", "LastName": "Smith"}	111	HR
2	{"FirstName": "Michael", "MiddleName": "Rose", "LastName": "Dan"}	222	SALES
3	{"FirstName": "Robert", "MiddleName": "Ray", "LastName": "Williams"}	333	IT
4	{"FirstName": "Maria", "MiddleName": "Anne", "LastName": "Jones"}	444	IT
5	{"FirstName": "Jen", "MiddleName": "Mary", "LastName": "Brown"}	555	HR

If we change struct data type with 2nd record as null, then

Define Struct Type Data

```
1  structureData = [
2      (("James","Will","Smith"),111,"HR"),
3      (("Michael","Dan"),222,"SALES"),
4      (("Robert","Ray","Williams"),333,"IT"),
5      (("Maria","Anne","Jones"),444,"IT"),
6      (("Jen","Mary","Brown"),555,"HR")
7    ]
```

Define Struct Type Schema

```
1  from pyspark.sql.types import StructType,StructField,StringType,IntegerType
2
3  structureSchema = StructType([
4          StructField('Name', StructType([
5               StructField('FirstName', StringType(), False),
6               StructField('MiddleName', StringType(), True),
7               StructField('LastName', StringType(), True)
8               ])),
9           StructField('ID', IntegerType(), True),
10          StructField('Department', StringType(), True)
11          ])
```

Create Dataframe With Nested Columns of Struct Type

```
1  df = spark.createDataFrame(data=structureData,schema=structureSchema)
2  df.printSchema()
3  df.display()
```

```
ValueError: field Name: Length of object (2) does not match with length of fields (3)
```

Expected length of fields is 3 but we have only 2 so we got that error.
Using StructType we must specify values whatever we are defining, we are defining 3 fields here, so we need to give 3 values here. even though we don't have any proper value, we have to populate that with null value that can be defined using none.
Now if we execute below steps, it will work.

Define Struct Type Data

```
1  structureData = [
2      (("James","Will","Smith"),111,"HR"),
3      (("Michael",None,"Dan"),222,"SALES"),
4      (("Robert","Ray","Williams"),333,"IT"),
5      (("Maria","Anne","Jones"),444,"IT"),
6      (("Jen","Mary","Brown"),555,"HR")
7    ]
```

Define Struct Type Schema

```
1  from pyspark.sql.types import StructType,StructField,StringType,IntegerType
2
3  structureSchema = StructType([
4          StructField('Name', StructType([
5               StructField('FirstName', StringType(), False),
6               StructField('MiddleName', StringType(), True),
7               StructField('LastName', StringType(), True)
8               ])),
9           StructField('ID', IntegerType(), True),
10          StructField('Department', StringType(), True)
11          ])
```

Now we could see output with no error , if we placed None in input.

```
df = spark.createDataFrame(data=structureData,schema=structureSchema)
df.printSchema()
df.display()
```

(3) Spark Jobs

```
root
 |-- Name: struct (nullable = true)
 |    |-- FirstName: string (nullable = false)
 |    |-- MiddleName: string (nullable = true)
 |    |-- LastName: string (nullable = true)
 |-- ID: integer (nullable = true)
 |-- Department: string (nullable = true)
```

Table +

	Name	ID	Department
1	{"FirstName": "James", "MiddleName": "Will", "LastName": "Smith"}	111	HR
2	{"FirstName": "Michael", "MiddleName": null, "LastName": "Dan"}	222	SALES
3	{"FirstName": "Robert", "MiddleName": "Ray", "LastName": "Williams"}	333	IT
4	{"FirstName": "Maria", "MiddleName": "Anne", "LastName": "Jones"}	444	IT
5	{"FirstName": "Jen", "MiddleName": "Mary", "LastName": "Brown"}	555	HR

When we are using StructType , it should follow standard key value pairs, we should not miss anything,we should not add anything extra, even though we are not having proper value we have to fill that with null value.

MAP TYPE

Coming to map type, there won't be standard number of key value pairs. In map type, we need to mention key and also value. In map type, it can accept any number of key value pairs and it can be completely different from one record to another record.

Define Map Type Data and Schema

```
#Create DataFrame with struct, array & map
from pyspark.sql.types import StructType,StructField,StringType,ArrayType,MapType

data=[(111,"Mike",{'TV':'LG','Refrigerator':'Samsung','Oven':'Philipps','AC':'Voltas'}),
      (222,"David",{'AC':'Samsung','Washing machine':'LG'}),
      (333,"Thomas",{'TV':'Croma'}),
      (444,"Williams",None) ]

schema = StructType([
        StructField('ID', IntegerType(),True),
        StructField('Name', StringType(),True),
        StructField('Utilities', MapType(StringType(),StringType()),True)
    ])
```

Create Dataframe With Map Type

```
df=spark.createDataFrame(data,schema)
df.printSchema()
df.display()
```

▸ (3) Spark Jobs

```
root
 |-- ID: integer (nullable = true)
 |-- Name: string (nullable = true)
 |-- Utilities: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

Table +

	ID	Name	Utilities
1	111	Mike	{"Refrigerator": "Samsung", "AC": "Voltas", "TV": "LG", "Oven": "Philipps"}
2	222	David	{"AC": "Samsung", "Washing machine": "LG"}
3	333	Thomas	{"TV": "Croma"}
4	444	Williams	null

DIFFERENT METHODS OF SCHEMA DEFINITION

FIRST METHOD OF CREATING SCHEMA

STANDATD SCHEMA DEFINITION

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


data1 = [("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)
  ]

schema1 = StructType([ \
    StructField("firstname",StringType(),False), \
    StructField("middlename",StringType(),True), \
    StructField("lastname",StringType(),True), \
    StructField("id", StringType(), True), \
    StructField("gender", StringType(), True), \
    StructField("salary", IntegerType(), True) \
  ])

df = spark.createDataFrame(data=data1,schema=schema1)

df.display()
```

Table +

	firstname	middlename	lastname	id	gender	salary
1	James		Smith	36636	M	3000
2	Michael	Rose		40288	M	4000
3	Robert		Williams	42114	M	4000
4	Maria	Anne	Jones	39192	F	4000
5	Jen	Mary	Brown		F	-1

NESTED SCHEMA DEFINITION

```
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(), False),
             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 = false)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)
```

```
+--------------------+-----+------+------+
|name                |id   |gender|salary|
+--------------------+-----+------+------+
|{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    |
+--------------------+-----+------+------+
```

HOW TO DEFINE SCHEMA OF ARRAY TYPE AND MAP TYPE

Array Type is nothing but we are having 1 column which can accept list of values. This is equivalent of python list. Map Type is used to hold key value pairs that is equivalent of python dictionary.

```
from pyspark.sql.types import *

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)
    ])


arraystructureData = [
    (("James","","Smith"),["music","reading","chess"],{'salary':'3000','dept':'HR'}),
    (("Michael","Rose",""),["music","playing","chess"],{'salary':'3000','dept':'HR'}),
    (("Robert","","Williams"),["music","reading","chess"],{'salary':'3000','dept':'HR'}),
    (("Maria","Anne","Jones"),["music","reading","chess"],{'salary':'3000','dept':'HR'}),
    (("Jen","Mary","Brown"),["music","reading","chess"],{'salary':'3000','dept':'HR'})
  ]

arrayDF = spark.createDataFrame(data=arraystructureData,schema=arrayStructureSchema)
display(arrayDF)
```

Table +

	name	hobbies	properties
1	{"firstname": "James", "middlename": "", "lastname": "Smith"}	["music", "reading", "chess"]	{"salary": "3000", "dept": "HR"}
2	{"firstname": "Michael", "middlename": "Rose", "lastname": ""}	["music", "playing", "chess"]	{"salary": "3000", "dept": "HR"}
3	{"firstname": "Robert", "middlename": "", "lastname": "Williams"}	["music", "reading", "chess"]	{"salary": "3000", "dept": "HR"}

Showing all 5 rows. | 0.86 seconds runtime

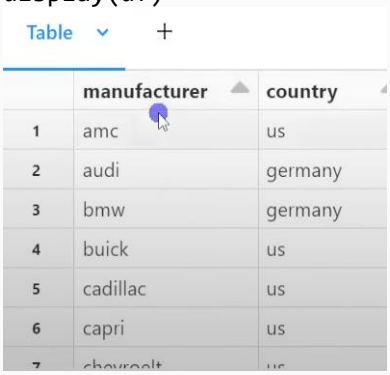
SCHEMA DEFINITION FOR READING BIGDATA FILES

%fs

head /FileStore/tables/manufacturers.csv

```
manufacturer,country
amc,us
audi,germany
bmw,germany
buick,us
cadillac,us
capri,us
chevroelt,us
chevrolet,us
chevy,us
chrysler,us
datsun,japan
dodge,us
fiat,italy
```

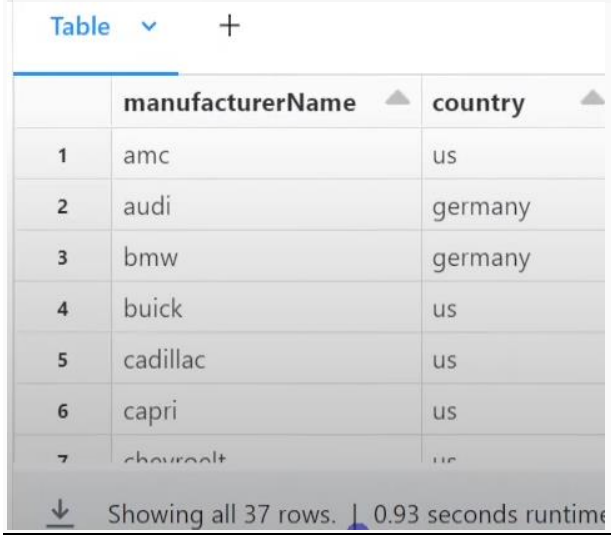
```
mySchema = StructType([ \
    StructField(“manufacturer”,StringType(),False), \
    StructField(“country”,StringType(),True) \
])
df =
spark.read.format(“csv”).option(“header”,True).schema(mySchema).load(“/FileStore/tables/manufacturers.csv”)
display(df)
```



	manufacturer	country
1	amc	us
2	audi	germany
3	bmw	germany
4	buick	us
5	cadillac	us
6	capri	us

SECOND METHOD OF CREATING SCHEMA

```
inlineSchema = "manufacturerName STRING, country STRING"

df = spark.read.format("csv").option("header",True).schema(inlineSchema).load("/FileStore/tables/manufacturers.csv")

display(df)
```



	manufacturerName	country
1	amc	us
2	audi	germany
3	bmw	germany
4	buick	us
5	cadillac	us
6	capri	us

Showing all 37 rows. 0.93 seconds runtime

THIRD METHOD OF CREATING SCHEMA

Here in this method, spark will infer the schema automatically based on the input data for each column

```
data = [
    ("Mike","M",50000,2),
    ("David","F",45000,3),
    ("Thomas","M",47000,2),
    ("William","M",40000,4),
    ("Steve","F",35000,5)
  ]

schm=["Name of employee","Gender","Salary","Years of experience"]

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

df.display()
```

Table +

	Name of employee	Gender	Salary	Years of experience
1	Mike	M	50000	2
2	David	F	45000	3
3	Thomas	M	47000	2
4	William	M	40000	4
5	Steve	F	35000	5

Even though we haven't defined nullable property or datatype , execution is successful.

FOURTH METHOD OF CREATING SCHEMA-CREATE SCHEMA ALONG WITH DATAFRAME

```
df = spark.createDataFrame([
  ("Mazda RX4",21,4,4),
  ("Hornet 4 Drive",22,3,2),
  ("Merc 240D",25,4,2),
  ("Lotus Europa",31,5,2),
  ("Ferrari Dino",20,5,6),
  ("Volvo 142E",22,4,2)
],["Car Name","mgp","gear","carb"])

df.display()
```

3) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [Car Name: string, mgp:

able +

	Car Name	mgp	gear	carb
	Mazda RX4	21	4	4
	Hornet 4 Drive	22	3	2
	Merc 240D	25	4	2
	Lotus Europa	31	5	2
	Ferrari Dino	20	5	6
	Volvo 142E	22	4	2

```
df.printSchema()
```

```
root
 |-- Car Name: string (nullable = true)
 |-- mgp: long (nullable = true)
 |-- gear: long (nullable = true)
 |-- carb: long (nullable = true)
```

Command took 0.08 seconds -- by audaciousazure@gmail.com at 2/2/2023, 8:28:05 PM on demo

Cmd 14

```
print(df.schema)
```

Python

```
StructType([StructField('Car Name', StringType(), True), StructField('mgp', LongType(), True), StructField('gear', LongType(), True), StructField('carb', LongType(), True)])
```

If we want to get schema output in the form of json, then we can go with df.schema.json() which means output of df.schema.json will be bundled within json

SCHEMA COMPARISION- HOW TO COMPARE SCHEMA BETWEEN MULTIPLE DATAFRAMES

Creating 3 sample data frames

Create First Dataframe

```
empData1 = [(111,"Stephen","King",2000),
    (222,"Philipp","Larkin",8000),
    (333,"John","Smith",6000)
  ]
empSchema1 = ["Id","FirstName","LastName","salary"]

df1 = spark.createDataFrame(data=empData1, schema = empSchema1)

display(df1)
```

Cmd 2

Create Second Dataframe with Same Schema as First Dataframe

```
empData2 = [(444,"Thomas","Frank",4000),
    (555,"Stephen","Fleming",3000),
    (666,"William","Pending",7000)
  ]
empSchema2 = ["Id","FirstName","LastName","salary"]

df2 = spark.createDataFrame(data=empData2, schema = empSchema2)

display(df2)
```

df1 output

Table +

	Id	FirstName	LastName	salary
1	111	Stephen	King	2000
2	222	Philipp	Larkin	8000
3	333	John	Smith	6000

df2 output

Table +

	Id	FirstName	LastName	salary
1	444	Thomas	Frank	4000
2	555	Stephen	Fleming	3000
3	666	William	Pending	7000

--

For df3 there is schema difference when compared to df1 and df2.

In df3 FirstName and LastName is missing , in df1 and df2 Name column is missing.

Create Third Dataframe with Different Schema from First Dataframe

```
empData3 = [(777,"David",4000),
    (888,"Mike",3000),
    (999,"Winsten",7000)
  ]
empSchema3 = ["Id","Name","salary"]

df3 = spark.createDataFrame(data=empData3, schema = empSchema3)

display(df3)
```

▸ (3) Spark Jobs

▸ df3: pyspark.sql.dataframe.DataFrame = [Id: long, Name: string ... 1 more field]

Table +

	Id	Name	salary
1	777	David	4000
2	888	Mike	3000
3	999	Winsten	7000

Let’s assume we have a project requirement to compare data frame between first and second data frame, in case if the schema is matching, we will do certain operation. In case if schema is not matching, then we will perform certain other operation.

Compare Schema of First and Second Dataframe

```
1  if df1.schema == df2.schema:
2    print("Schema Matches")
3  else:
4    print("Schema Does Not Match")
```

```
Schema Matches
```

Compare Schema of First and Third Dataframe

```
1  if df1.schema == df3.schema:
2    print("Schema Matches")
3  else:
4    print("Schema Does Not Match")
```

```
Schema Does Not Match
```

List of Columns Missing in Third Dataframe

```
1  print(list(set(df2.columns) - set(df3.columns)))
```

```
['LastName', 'FirstName']

Command took 0.05 seconds -- by audaciousazure@gmail.com at 2/3/2023, 7:09:14 P
```

Cmd 7

List of Columns Missing in Second Dataframe

```
1  print(list(set(df3.columns) - set(df2.columns)))
```

```
['Name']

Command took 0.10 seconds -- by audaciousazure@gmail.com at 2/3/2023, 7:09:44 P
```

Collect All Possible Columns in a List

```
1  allColumns= df1.columns+df3.columns
2  uniqueColumns = list(set(allColumns))
3  print(uniqueColumns)
```

```
['Id', 'Name', 'LastName', 'salary', 'FirstName']
```

Add Missing Columns

```
1  from pyspark.sql.functions import lit
2  for col in uniqueColumns:
3      if col not in df1.columns:
4        df1 = df1.withColumn(col, lit(None))
5      if col not in df3.columns:
6        df3 = df3.withColumn(col, lit(None))
7
8  display(df1)
9  display(df3)
```

Table +

	Id	FirstName	LastName	salary	Name
1	111	Stephen	King	2000	null
2	222	Philipp	Larkin	8000	null
3	333	John	Smith	6000	null

Showing all 3 rows. | 1.30 seconds runtime

Table +

	Id	Name	salary	LastName	FirstName
1	777	David	4000	null	null
2	888	Mike	3000	null	null
3	999	Winsten	7000	null	null

We can also write query in the below way also.

```
def addMissingColumns(df1,df2):
  from pyspark.sql.functions import lit
  allColumns= df1.columns+df2.columns
  uniqueColumns = list(set(allColumns))
  for col in uniqueColumns:
      if col not in df1.columns:
        df1 = df1.withColumn(col, lit(None))
      if col not in df3.columns:
        df2 = df2.withColumn(col, lit(None))
  return df1,df2
```

```
df1, df3 = addMissingColumns(df1,df3)
display(df1)
display(df3)
```

(6) Spark Jobs

df1: pyspark.sql.dataframe.DataFrame = [Id: long, FirstName: string ... 3 more fields]

df3: pyspark.sql.dataframe.DataFrame = [Id: long, Name: string ... 3 more fields]

Table +

	Id	FirstName	LastName	salary	Name
1	111	Stephen	King	2000	null
2	222	Philipp	Larkin	8000	null
3	333	John	Smith	6000	null

Showing all 3 rows. | 0.94 seconds runtime

Table +

	Id	Name	salary	LastName	FirstName
1	777	David	4000	null	null
2	888	Mike	3000	null	null
3	999	Winston	7000	null	null

DATA SECURITY- COLUMN LEVEL DATA ENCRYPTION

WHY DATA SECURITY IS IMPORTANT?

We need to protect sensitive information. Databricks is used to store and process large volumes of data some of which might contain sensitive information such as financial data, personal information or intellectual property. So, it's crucial to ensure that this information is protected from unauthorized access theft or misuse.

Coming to compliance, most of the industries are following certain regulatory compliances such as GDPR that is global data production regulation. Those regulations

mandate production of sensitive data. Complaints with these regulations is necessary to avoid any legal penalties.

Coming to preventing data breaches, data breaches can have serious consequences to any business including financial losses, computational damage, or legal liabilities. Proper data security measures can help prevent data breaches and minimize their impact.

Finally maintaining trust with customers is one of the important area for any business customers and clients trust organizations with their data ensuring that their data is secure and critical for maintaining the trust and fostering long term relationship.

DIFFERENT SECURITY FEATURES

Databricks provides several security features such as Encryption at Transit/Rest, RBAC, Audit Logging and MFA.

Whenever we are building data lake or building DWH solution within databricks, how we can encrypt data at column level?

Create Sample Delta Table

```
1  from delta.tables import *
2
3  DeltaTable.create(spark) \
4    .tableName("dimEmployee") \
5    .addColumn("empId", "INT") \
6    .addColumn("empName", "STRING") \
7    .addColumn("SSN", "STRING") \
8    .execute()
```

▸ (4) Spark Jobs

```
Out[1]: <delta.tables.DeltaTable at 0x7f8803be75e0>
```

Insert PII data

```
1  %sql
2  insert into dimEmployee values (100, 'Mike', '2345671');
3  insert into dimEmployee values (200, 'David', '5632178');
4  insert into dimEmployee values (300, 'Peter', '1456782');
```

View Data

```
%sql
select * from dimEmployee
```

(3) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [empId

Table +

	empId	empName	SSN
1	200	David	5632178
2	300	Peter	1456782
3	100	Mike	2345671

Now we need to install library Cryptography to generate encryption or decryption of data.

Cryptography Library Installation

Python

```
pip install cryptography
```

```
Python interpreter will be restarted.
Requirement already satisfied: cryptography in /databricks/python3/lib/python3.9/site-packages (3.4.8)
Requirement already satisfied: cffi>=1.12 in /databricks/python3/lib/python3.9/site-packages (from cryptography) (1.14.6)
Requirement already satisfied: pycparser in /databricks/python3/lib/python3.9/site-packages (from cffi>=1.12->cryptography) (2.20)
Python interpreter will be restarted.
```

Generate Encryption/Decryption Key

```
from cryptography.fernet import Fernet

key = Fernet.generate_key()
f = Fernet(key)
```

Command took 0.19 seconds -- by audaciousazure@gmail.com at 3/2

Encrypt Sample Data

Python

```
PIIData =b"testmail@gmail.com"
TestData = f.encrypt(PIIData)
print(TestData)
```

```
b'gAAAAABkHEYqx2jRJuxkPScvuSgukIKAej_DbLJwAIyhqGLUDMu80vqcBFmOoEbYmYiLuEqDczfudgb4nnj7zfmfpCHq6ZFoyGo-C8hM9HEmwN6K6rRsPy8='
```

Decrypt Sample Data

```
1  print(f.decrypt(TestData))
```

```
b'testmail@gmail.com'
```

Define UDF to Encrypt Data

```
1  def encrypt_data(data,KEY):
2      from cryptography.fernet import Fernet
3      f = Fernet(KEY)
4      dataB=bytes(data, 'utf-8')
5      encrypted_data = f.encrypt(dataB)
6      encrypted_data = str(encrypted_data.decode('ascii'))
7      return encrypted_data
```

Command took 0.06 seconds -- by audaciousazure@gmail.com at 3/23/2023, 6:01:03 PM

Cmd 9

Define UDF to Decrypt Data

```
1  def decrypt_data(encrypted_data,KEY):
2      from cryptography.fernet import Fernet
3      f = Fernet(KEY)
4      decrypted_data=f.decrypt(encrypted_data.encode()).decode()
5      return decrypted_data
```

We will register UDF using udf().

Register UDF's

```
1  from pyspark.sql.functions import udf, lit, md5
2  from pyspark.sql.types import StringType
3
4  encryption = udf(encrypt_data, StringType())
5  decryption = udf(decrypt_data, StringType())
```

Encrypt the data

```
df = spark.table("dimEmployee")
encryptedDF = df.withColumn("ssn_encrypted", encryption("SSN",lit(key)))
display(encryptedDF)
```

(2) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [empId: integer, empName: string ... 1 more field]

encryptedDF: pyspark.sql.dataframe.DataFrame = [empId: integer, empName: string ... 2 more fields]

Table

	empId	empName	SSN	ssn_encrypted
1	200	David	5632178	gAAAAABkHEblepuyFxbyrTFbIkzGHVtZXjICuItIcaXMyJFNuVpNkXLSWxwe6J9FFTDpM4gJRACMDzEr2V8TasUmSFC_50
2	300	Peter	1456782	gAAAAABkHEbl9AS5JITiwmr3i82zJRp_9jKe5hzTLGtPGv4_C_pV3kXbvq82KO2jBOwyTzOi2WbB7c1Vo1LYL6CH_wacp4HI
3	100	Mike	2345671	gAAAAABkHEblYOJUC0Vad0-dTOcm6hco9JYV-EMFxVKTxVGpmjAuy-mBml14ZwGg1l5y54UNbrqV-w6ubAMYbtN4KeI ynOA==

Here we kept ssn and ssn_encrypted but in real time, once we have encrypted we will remove actual data from this table. We can give only column without original data to the users.

--

Later for some users they should be able to see original data then they can decrypt using same key as shown below.

Decrypt the data

```
decryptedDF = encryptedDF.withColumn("ssn_decrypted", decryption("ssn_encrypted",lit(key)))
display(decryptedDF)
```

(2) Spark Jobs

decryptedDF: pyspark.sql.dataframe.DataFrame = [empId: integer, empName: string ... 3 more fields]

Table

	SSN	ssn_encrypted	ssn_decrypted
1	5632178	gAAAAABkHEcexvUf4DVKiqCGBgohb0DitELoZYdoSbp4xfzQB17Ctj7IFC9ASf7BL_hW-BdTDSwe4QxWoRnN8N2tAdG4IJZw==	5632178
2	1456782	gAAAAABkHEcfcnHPtQbl0THT5MF6857deyVZXm40kLk7gLTlK_jBudmXrBkqf3zyIMqvEmxb2B3oXpe1QyeuYkJX-YNbAdk5eQ==	1456782
3	2345671	gAAAAABkHEcfw_6QTGEQx-3SKwpZ7Zye9Q-NzNL5392OEeEMAZJbjtKTGDuo6-lXvlUAtP48WlBe9uZXl8OOa5uEd79GSkhU1Q==	2345671

DATAFRAME- PANDAS VS PYSPARK

Pandas and pyspark used for bigdata processing. Even though they are python libraries, both are used for big data still there are significant differences between two libraries in terms of processing data volume or processing data speed.

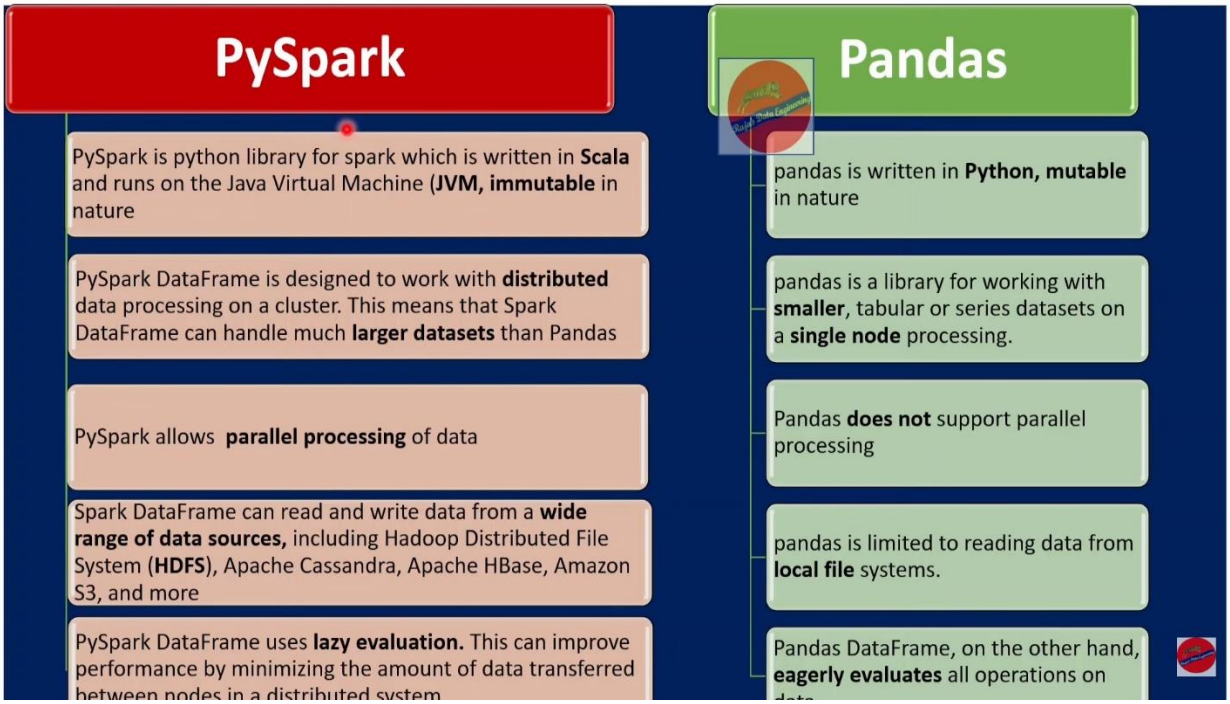
Pandas is one of the open-source python library that provides high performance data manipulation and analysis tools which means in bigdata processing whenever we are processing huge amount of data, we can use pandas library to create data frame and apply data manipulation , data transformation or any data analysis on top of that. This is built on top of numpy library. In pandas, we can process single dimensional

array that is similar to list in python and also dataframe that is 2 dimensional matrix its nothing but tabular structure.
Pyspark is mostly used for data processing and data engineering side, coming to pandas it mostly suitable for ML and AI.
Pyspark is python API or python library that is mainly provided for apache spark

PySpark is the Python API for Apache Spark, an open-source distributed computing system used for big data processing and analytics.

It provides a high-level interface for programming Spark with Python, enabling data scientists and engineers to write distributed data processing code in Python.

PySpark allows users to leverage the power of Spark's distributed computing engine for data processing, machine learning, and analytics tasks.



HOW TO GENERATE HUGE AMOUNT OF TEST DATA FOR TESTING SCENARIO?
Once we have developed our solution, we normally go with testing. For testing we usually used to get only limited amount of data in the development phase or in the testing phase but once we move the solution to production, actually that solution should be scalable, we should be able to handle huge amount of data also so for that we have to create test data during the test phase.

Array_repeat is one of the pyspark function which is creating an array containing a value from one of the dataframe column with repeated count times.

Let's say we have a dataframe which has 2 columns ID and name with 1 record. Let's say we want to create repeated value of one of the column ID, we have to create repeated array value, let's say this term should be repeated 5 times, 10 times, 1000 times or could be any number of times.so we want to create one new array column which would contain repeated values of one of the column so this can be achieved using array repeat.

Let's say if we are going to give array_repeat function of 5 times, then this ID value will be repeated 5 times and result will be array.

This is used for performance testing. Like in order to process 1000 records how much time it takes. In order to process 1 million records how much time it takes, In order to process 1 billion records how much time it takes, in order to compare performance, we need to generate a huge amount of test data so for that this process can be used.

CODE:

Create Sample Delta Table

```
1  empData = [(111,"Stephen","King",2000),
2             (222,"Philipp","Larkin",8000),
3             (333,"John","Smith",6000) ]
4
5  empSchema = ["Id","FirstName","LastName","salary"]
6
7  df = spark.createDataFrame(data=empData, schema = empSchema)
8
9  display(df)
```

(3) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [Id: long, FirstName: string ... 2 more fiel

Table +

	Id	FirstName	LastName	salary
1	111	Stephen	King	2000
2	222	Philipp	Larkin	8000
3	333	John	Smith	6000

```
1  from pyspark.sql.functions import explode,array_repeat,col
2  repeatDf= df.withColumn("key_col",array_repeat(col("Id"),10) )
3  display(repeatDf)
```

▸ (3) Spark Jobs

▸ repeatDf: pyspark.sql.dataframe.DataFrame = [Id: long, FirstName: string ... 3 more fields]

Table +

	Id	FirstName	LastName	salary	key_col
1	111	Stephen	King	2000	▸ [111, 111, 111, 111, 111, 111, 111, 111, 111, 111]
2	222	Philipp	Larkin	8000	▸ [222, 222, 222, 222, 222, 222, 222, 222, 222, 222]

3 rows | 0.71 seconds runtime

In order to convert this single elements into separate rows we have to apply explode function.

```
1  df1= df.withColumn("key_col",explode(array_repeat(col("Id"),10) ))
2  display(df1)
```

▸ (3) Spark Jobs

▸ df1: pyspark.sql.dataframe.DataFrame = [Id: long, FirstName: string ... 3 more fields]

Table +

	Id	FirstName	LastName	salary	key_col
1	111	Stephen	King	2000	111
2	111	Stephen	King	2000	111
3	111	Stephen	King	2000	111
4	111	Stephen	King	2000	111
5	111	Stephen	King	2000	111
6	111	Stephen	King	2000	111
7	111	Stephen	King	2000	111

30 rows | 0.57 seconds runtime

```
from pyspark.sql.window import Window
from pyspark.sql.functions import lit, row_number

win =Window().orderBy(lit('A'))
df2=df1.withColumn("RowN", row_number().over(win))
df3 =df2.withColumn("NewID",col("Id")+col("RowN"))
display(df3)
```

(2) Spark Jobs

df2: pyspark.sql.dataframe.DataFrame = [Id: long, FirstName: string ... 4 more fields]

df3: pyspark.sql.dataframe.DataFrame = [Id: long, FirstName: string ... 5 more fields]

Table +

	Id	FirstName	LastName	salary	key_col	RowN	NewID
1	111	Stephen	King	2000	111	1	112
2	111	Stephen	King	2000	111	2	113
3	111	Stephen	King	2000	111	3	114
4	111	Stephen	King	2000	111	4	115
5	111	Stephen	King	2000	111	5	116
6	111	Stephen	King	2000	111	6	117
7	111	Stephen	King	2000	111	7	118

1,000 rows | Truncated data | 1.32 seconds runtime

Here data is truncating to 1000 rows so totally 3000 rows will be there.

```
outputDF = df3.drop("Id","key_col","RowN").select(col("NewID").alias("Id"),"FirstName","LastName","salary")
display(outputDF)
```

(2) Spark Jobs

outputDF: pyspark.sql.dataframe.DataFrame = [Id: long, FirstName: string ... 2 more fields]

Table +

	Id	FirstName	LastName	salary
1	112	Stephen	King	2000
2	113	Stephen	King	2000
3	114	Stephen	King	2000
4	115	Stephen	King	2000
5	116	Stephen	King	2000
6	117	Stephen	King	2000
7	118	Stephen	King	2000

1,000 rows | Truncated data | 0.89 seconds runtime

We are summarizing all the steps in one UDF.

```
def multiplyInputData(df, keyCol,multiplyFactor):

  from pyspark.sql.functions import explode,array_repeat,col,lit, row_number
  from pyspark.sql.window import Window

  df1= df.withColumn(f"{keyCol}",explode(array_repeat(col(f"{keyCol}"),multiplyFactor) ))

  win =Window().orderBy(lit('A'))
  df2=df1.withColumn("RowN", row_number().over(win))

  df3 =df2.withColumn(f"{keyCol}",col("Id")+col("RowN")).drop("RowN")
  return df3
```

```
1  finalDF =multiplyInputData(df,"Id",10)
2  display(finalDF)
```

▸ (2) Spark Jobs

▸ finalDF: pyspark.sql.dataframe.DataFrame = [Id: long, FirstName: string

Table +

	Id	FirstName	LastName	salary
1	112	Stephen	King	2000
2	113	Stephen	King	2000
3	114	Stephen	King	2000
4	115	Stephen	King	2000
5	116	Stephen	King	2000
6	117	Stephen	King	2000
7	118	Stephen	King	2000

↓ 30 rows | 0.87 seconds runtime