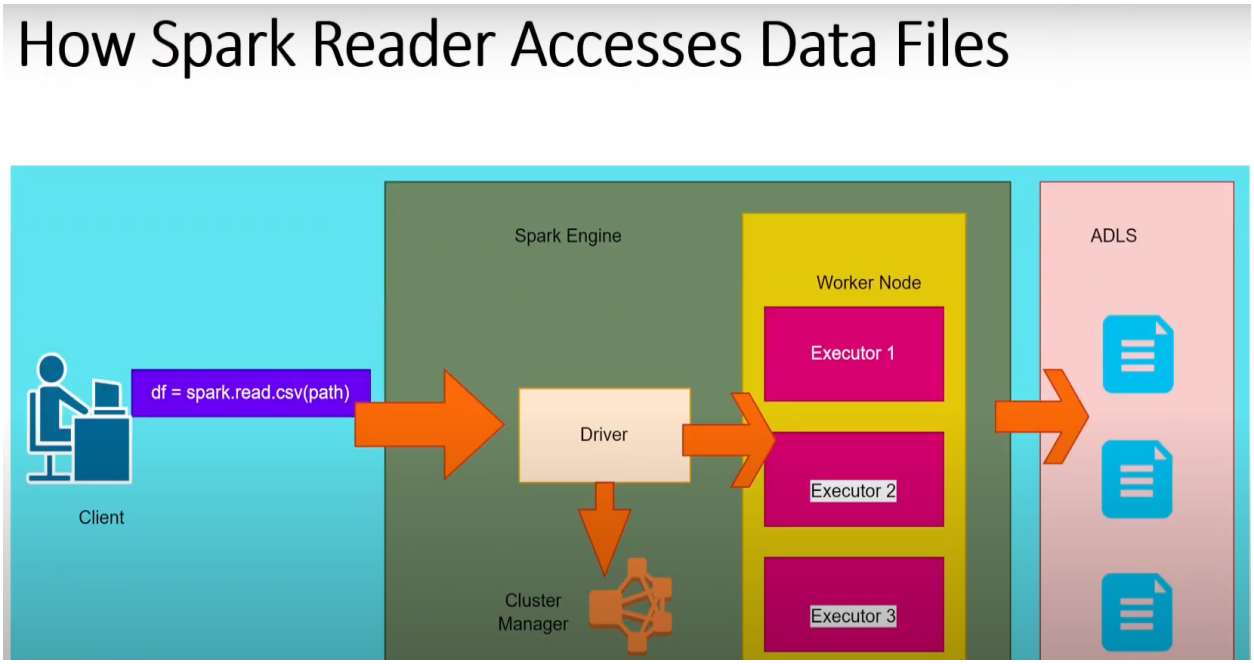
INTERNALS OF PARTITION CREATION

Whenever we have to process huge amount of data, let's say tera bytes, peta bytes or exabytes, the data should be splitted in the form of partitions and it will be distributed across multiple nodes then multiple cores from multiple nodes can work in parallel to process the data.

DIFFERENT COMPONENTS INVOLVED WHILE CREATING PARTITION

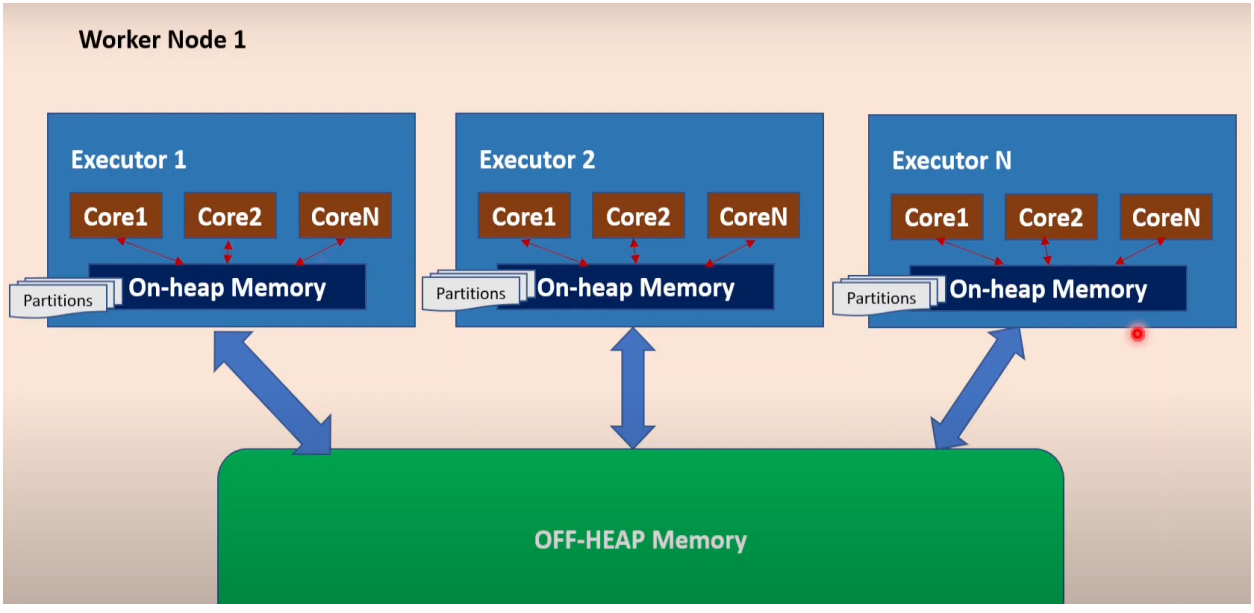


Let's say one of the application is contacting spark engine to create a dataframe.client application giving code like df= spark.read.csv(path) , the path could be from ADLS or HDFS location or even DBFS. In order to connect ADLS, data lake storage should be mounted. Let's say data lake storage is mounted and in one of the location we are having certain csv files, now the dataframe df is created by reading csv file from ADLS. Whenever this piece of code is submitted to spark engine what happens is spark context would be created and first this particular code will be submitted to driver. Driver will understand this code and it will optimize the code, it will split entire process into multiple smaller tasks inside. Basically, it is creating lineage graph, lazy evolution. Once we are calling an action then entire code is executed by calling previous transformations.

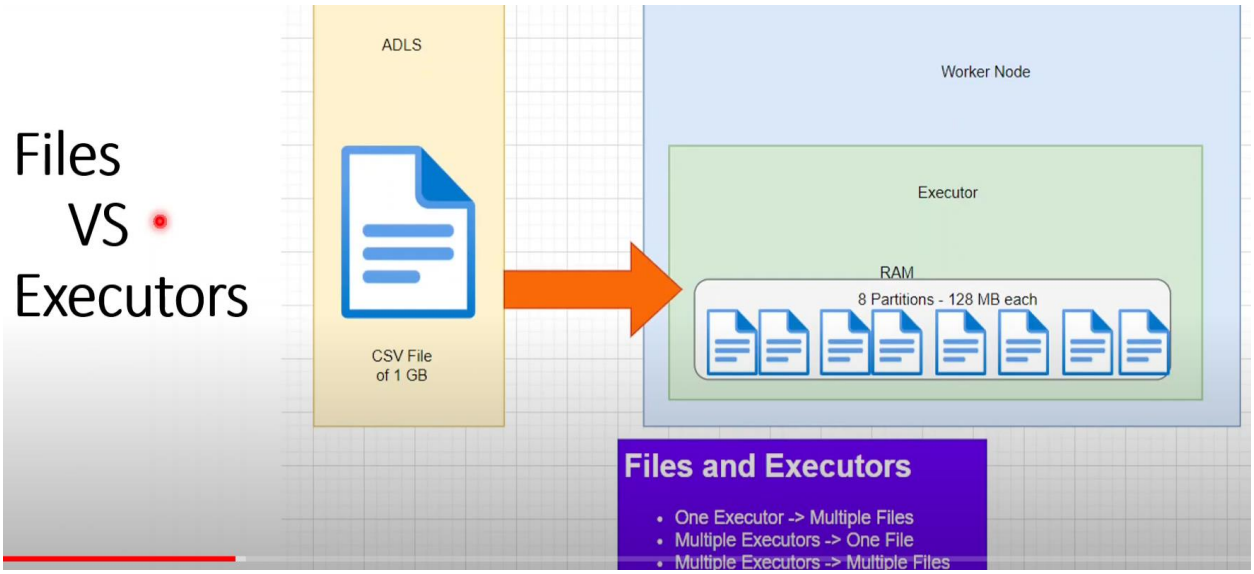
So, whenever we are submitting some piece of code into driver, it will optimize code, it will create best execution plan and it will divide entire task into multiple smaller steps and then it will submit those smaller steps into different executors. One worker

node might contain one or more than one executors. Now whenever driver is receiving program from client application, then it is optimizing, then it is submitting one small portion of the work to worker node into the executor1, then executor1 will connect to ADLS and it will start reading the data in parallel so each executor will consist of number of cores so each core will start consuming the data from external storage in parallel as long as file is in splitable format so for this parallel processing or parallel reading, important concept is file should be in the form of splitable format. even though file is compressed, few compressed file formats are splitable. But coming to traditional sip or tar these compressed formats are non-splitable. So, let's assume we have kept our file in splitable format then worker nodes can connect to external storage and then based on that it can pull the data from ADLS and it will distribute the data across multiple executor in the form of partitions

Summary: Driver is getting application from one of the client application or developer then it is optimizing the code, dividing the task into multiple smaller steps then each and every step will be given to executors, so all the cores within the executor will start working in parallel. Let's say we have to connect to ADLS that's one of the external storage for that we need access key so storage account was already mounted so the access key would be already part of driver so driver will give those credentials to all the executors when it is submitting piece of code. So, through that credentials, all executors can establish the connection with external storage and it can start processing the data.



Cluster is combination of worker node and driver node. Worker node is responsible to execute any piece of logic. Worker node that is logically divided into multiple executors as shown above. And each executor will consist of multiple cores. Let's say we are having one worker node which is having 12 cores and 12GB RAM. Let's assume what happens is we are going to divide that worker node into 2 executors, it will be divided equally 6 cores will go to 1 executor and 6 cores will go to another executor. Similarly, RAM also be splitted into 2 parts and internally we have to assign some cores for operating system and also, we have to assign memory for OS. So, on this example we didn't consider that. So just for this example, we are giving 6 cores plus 6 GB RAM but apart from that we have off heap memory also. Whenever some piece of code is given by driver to the executor then all the cores within executor that is responsible to pull the data from external storage and it will store those data in the form of partitions within on heap memory that is nothing but RAM memory.

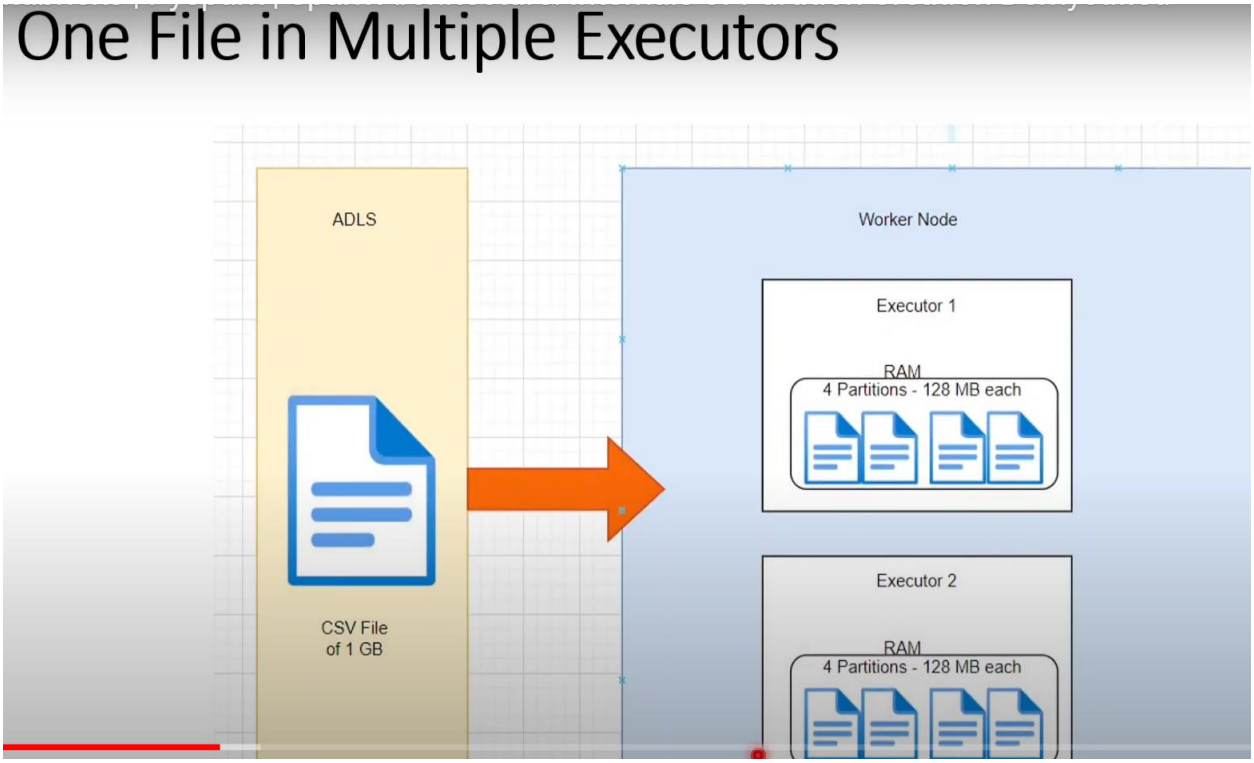


Let us assume we are having 1 GB csv file in ADLS location and we are having only one worker node. Within that we are having only one executor. Let us assume we are having partition size as 128MB. In this case 1GB file will be divided logically into 8 pieces then while consuming data from external storage into spark environment then it will come in the form of partition. Those logical divisions will be converted into physical partitions within executor on heap memory.

In this example, we are having only 1 csv file and 1 executor. In real time we might have multiple files under a particular folder and multiple executors, in cluster we will have multiple worker nodes. Within one worker node, we will have multiple executors so multiple executors can work on one particular csv file which is splitable then it

can distribute the file across the executors even multiple executors can access one single file so

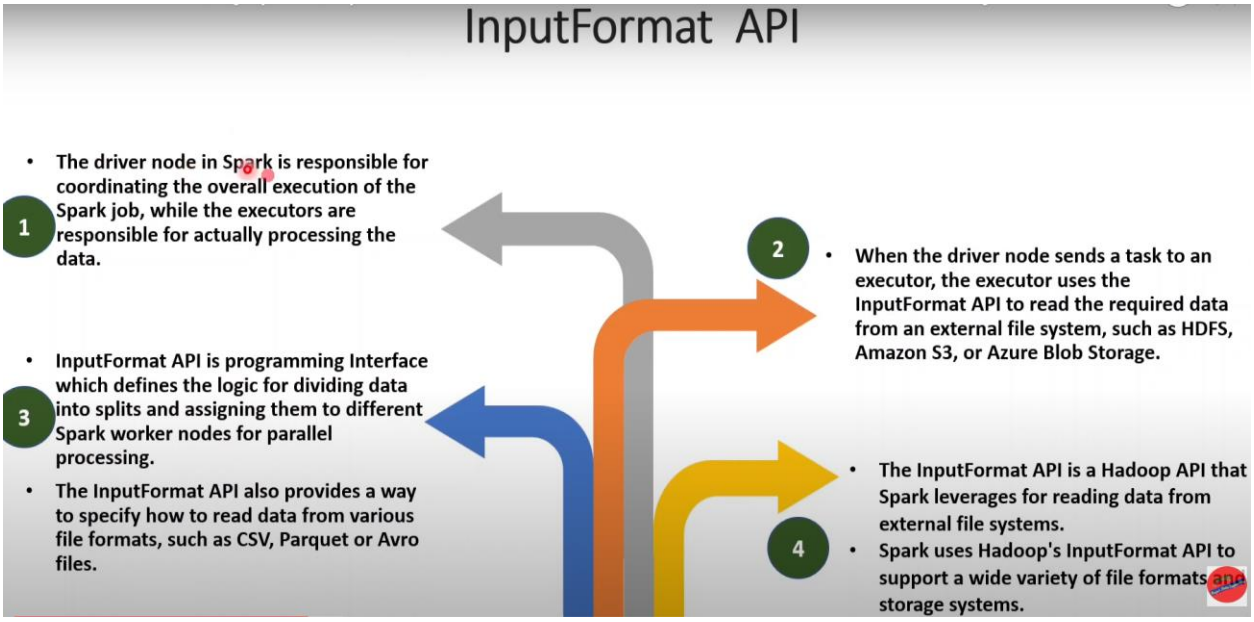
- One executor can access multiple files or multiple executors can access one file or multiple executors can access multiple files.

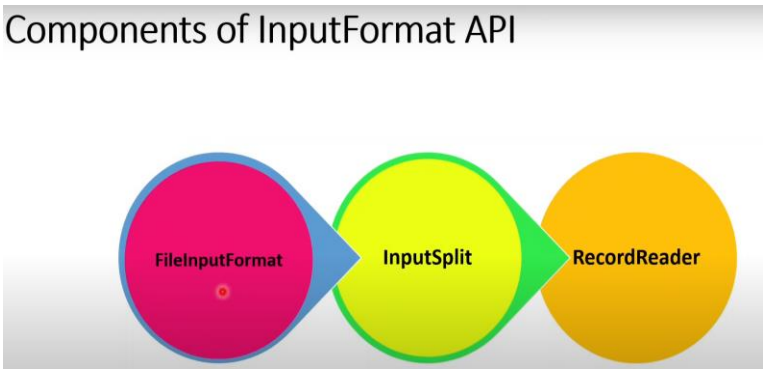


Let’s say we are having only 1 csv file with 2 executors in the worker node so both executors will start consuming the data from csv file, logically it was divided so all the cores within executor can consume the data in parallel so finally 8 partitions got created so it is uniformly distributed across the executors which means 4 partitions will go to executor1 and 4 partitions will go to executor2.

INTERNAL MECHANISM OF HOW WORKER NODE IS CONNECTING WITH ADLS

Within spark executors, there is a component called input format API. This component is responsible to connect with external storage and pull the data from external storage to spark environment in the form of partitions.



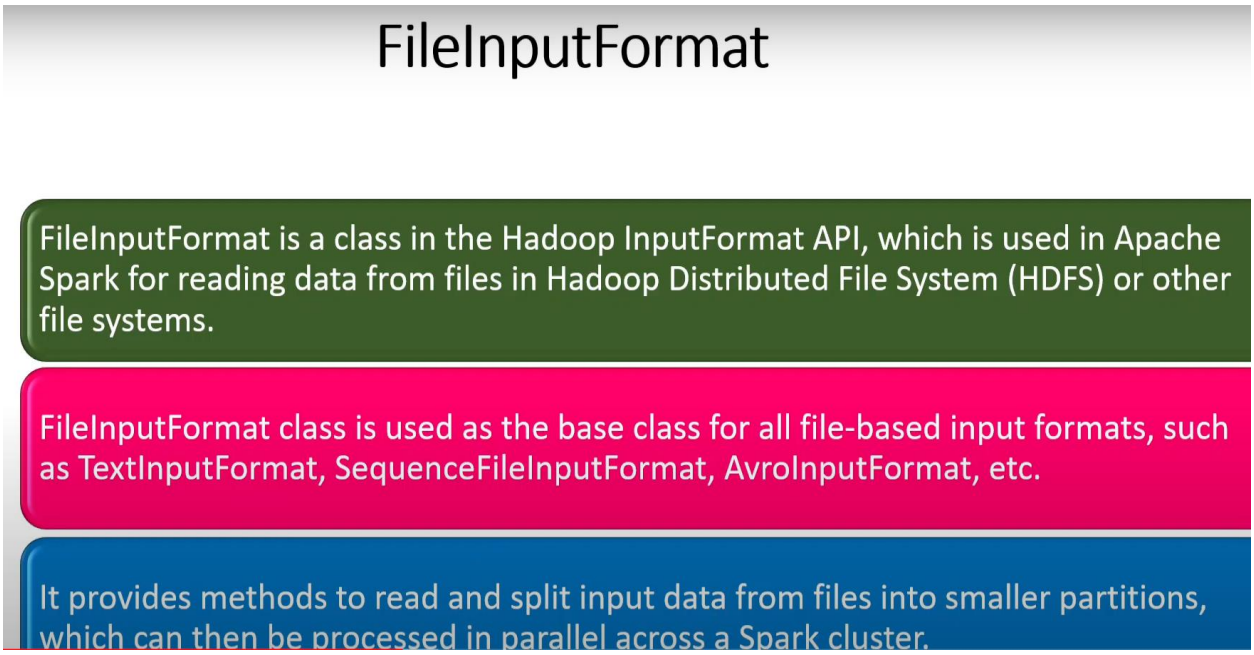


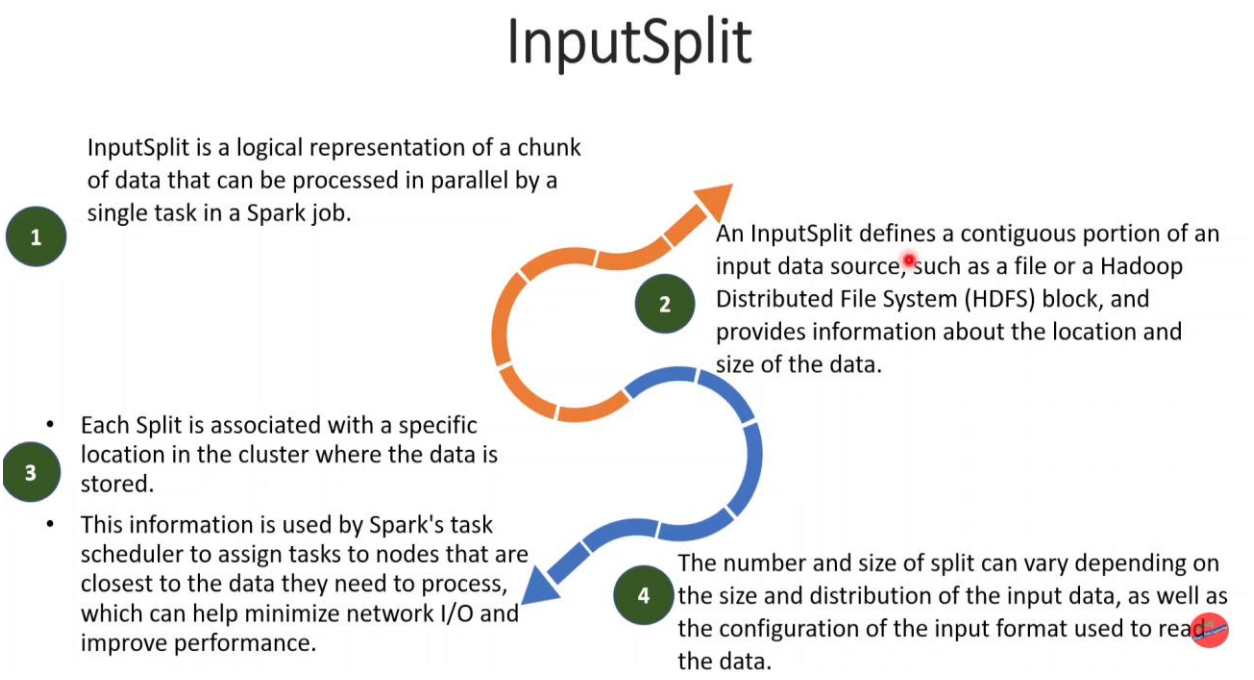
FileInputFormat : This component is responsible to handle different file formats in a different way let's say csv, parquet , Avro we are having different file formats, how we have to handle those different file formats that will be decided by FileInputFormat.

InputSplit: Input split is responsible to divide the data in the external storage logically. For e.g. we are having 1GB data that should be logically divided into 8 pieces. Basically, this InputSplit will access the external file then logically it will split. Logical split means it will note down address, starting memory address and end memory address for each split. That split will be converted as a partition within spark environment. So, this input split that is happening within external storage itself.

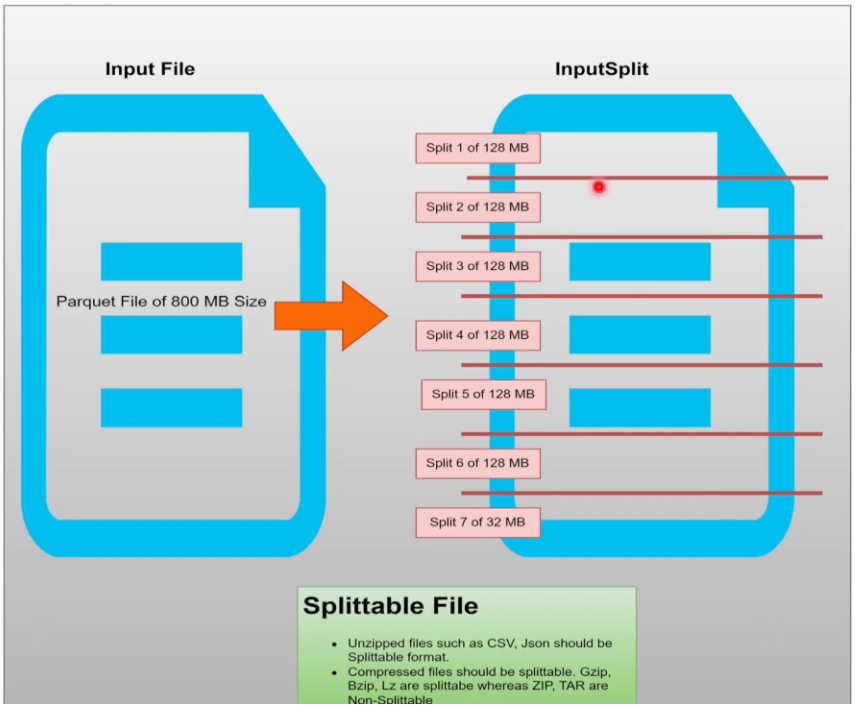
Record Reader: Once input split has splitted physical file into logical divisions which means logical splits then record reader will start consuming the data from external

file system and it will move the data to spark executors in the form of partitions. These are the major 3 components.

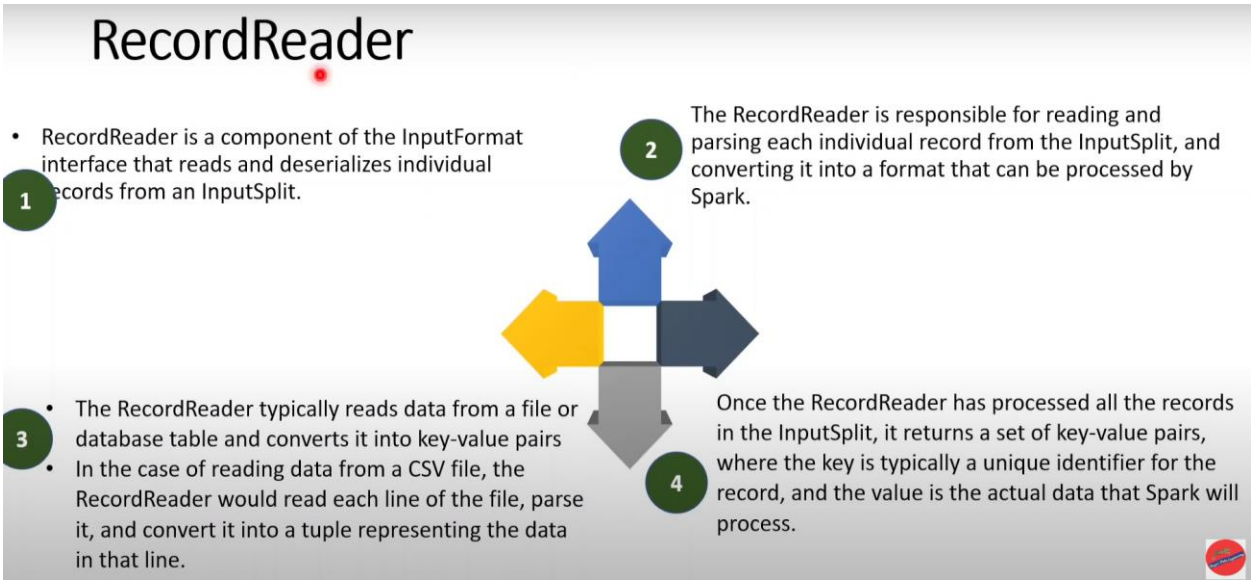




Logical Split
=
Input Split



Let's say in ADLS location we are having one input file which is parquet size of 800MB. This 800 MB should be divided into maximum of 128MB. So, what happens is this component input split is a part of input format API that is part of executor so executor will connect to this ADLS storage using these components, then it will start dividing entire file into logical divisions so basically first split will take 128MB from this parquet file then remaining it will have 672 bytes will be there. Again, it is higher than 128 MB again it will be splitted so it will keep on splitting. Finally last piece will have only 32MB. So, this 800MB is splitted into 7 partitions. In parquet file we will be having header and footer which will have data about this parquet file which means metadata(schema's information). Now those information will be available for all the splits. In order to split the parquet file, important condition is any file such as parquet,csv or json should be in splitable format but incase these files are corrupted then it cannot be splitted. Coming to compressed files Gzip, Bzip, Lz are splitable but ZIP and TAR are non-splitable files. Incase we are having non-splitable files then this entire 800MB will be consumed by only one core. This entire file will be considered as one logical piece so that it is going to hit the performance because we are impacting parallel processing. Its very important to use splitable file formats and compression formats while working in bigdata especially in spark.



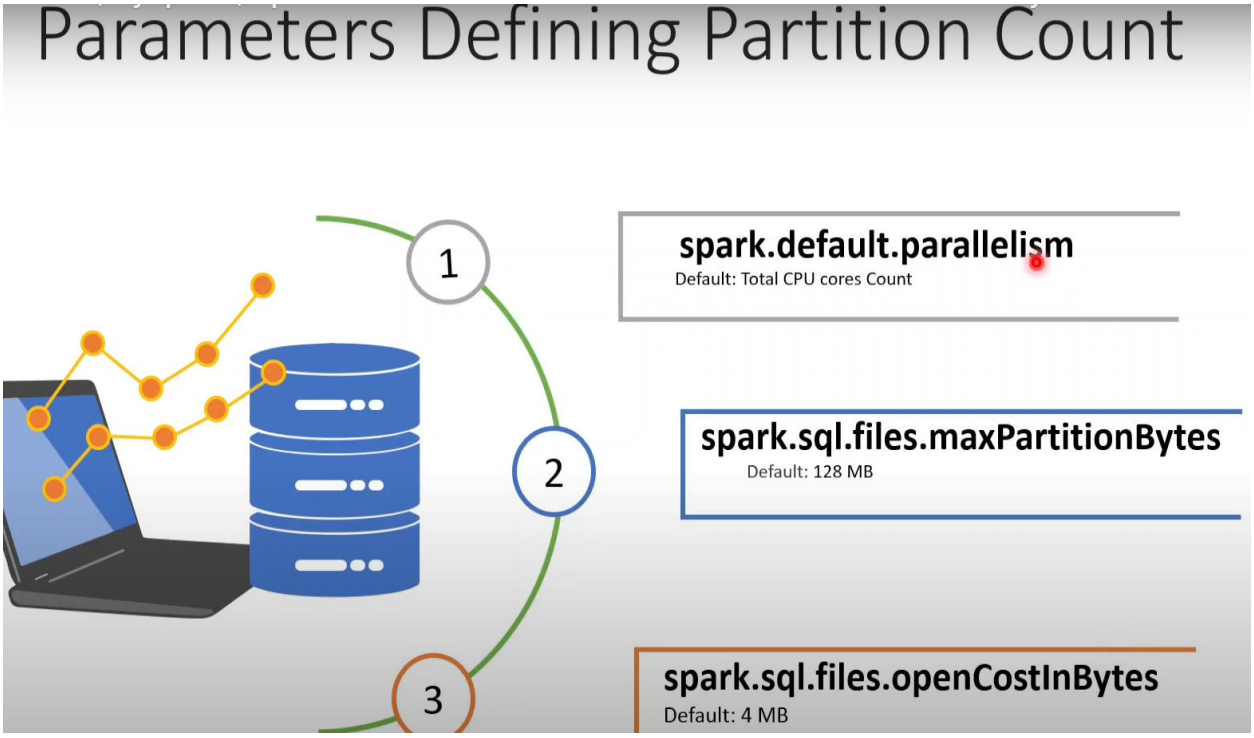
Once data is logically divided into multiple pieces then record reader will start consuming. So basically, this is transporting data from external file system to the spark environment so this is the entry point where data is flowing from external file system to spark in-memory and this record reader .

Summary: Whenever we are submitting some piece of code to spark engine to cluster, what happens is driver is accepting that code then that is dividing into multiple smaller tasks then each task is given to executors then executor will establish connection with external storage with the help of Input Format API. With an Input Format API we are having 3 components. File Format API that is responsible to handle different files in a different way and 2nd one is input split that is responsible to split actual physical file into multiple logical chunks and 3rd one is record reader that is responsible to transport each splitted input splitted data into spark in the form of partitions.

HOW NO OF PARTITIONS ARE DECIDED ?

In order to decide no of partitions there are 3 major parameters involved as shown below.

spark.default.parallelism → This is one of the important spark configuration and default value is total CPU core count. Let's say we are having one worker node within that one executor which is having 10 cores which means this parameter can be customized we can change to 20, 30 etc. but by default value would be total CPU cores count.

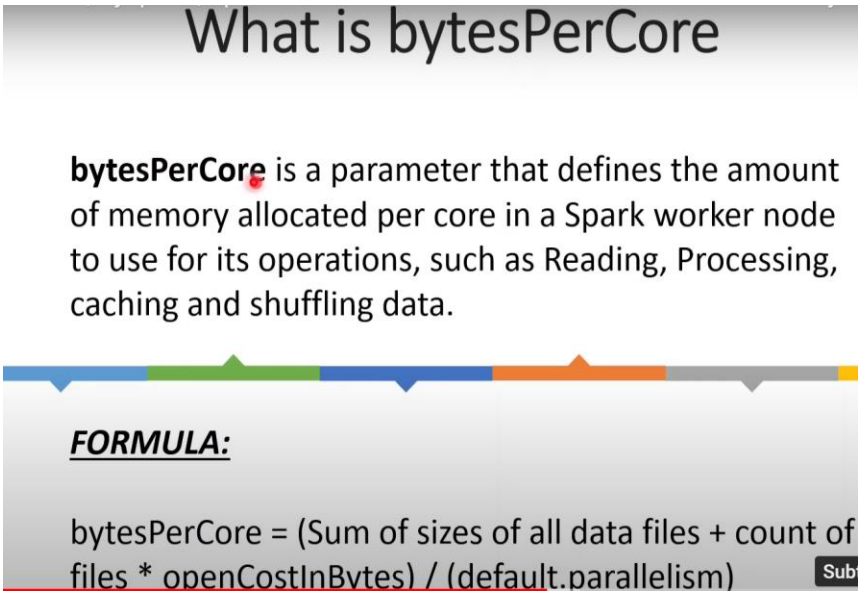


2nd important parameter is spark.sql.files.maxPartitionBytes → This will decide block size of each partition. By default, it will be 128MB. Again, depending on the use case we can change this one.

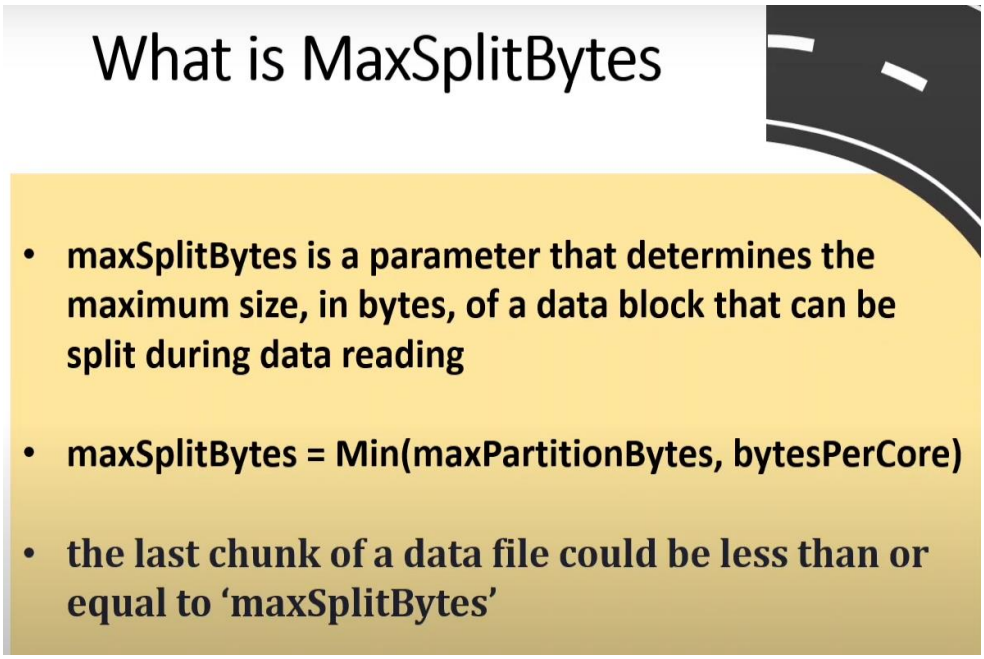
3rd important parameter is spark.sql.files.openCostInBytes → default value is 4MB but this is changeable. This is also playing important role in deciding number of partitions in spark environment, openCostInBytes means lets say we are having 800MB of parquet file which is logically divided into 8 pieces. After dividing those file into multiple pieces, incase if we have to pack 2 logical divisions into single file then whenever we have to pack 2 different files into one partition then we have to use 4MB as a overhead processing because this 4MB will be used as a buffer to merge 2 different files into one partition. Let's say we are having 20MB json file and one more 30MB json file , now we have to combine these two and convert into one single partition. In that process what happens is in first file 20MB will be taken and 4MB of this openCostInBytes will be used and then top of that 2nd file 30MB json file will be added so overall it will be 54MB. This 4MB is nothing but process overhead or buffer memory that is needed to pack multiple files into single partition.

These are the major parameters which are used to define partition count in spark environment.

In order to understand how partitions are being decided within a spark environment, we need to know 2 different formuals , 1) is bytesPerCore and 2) is MaxSplitBytes

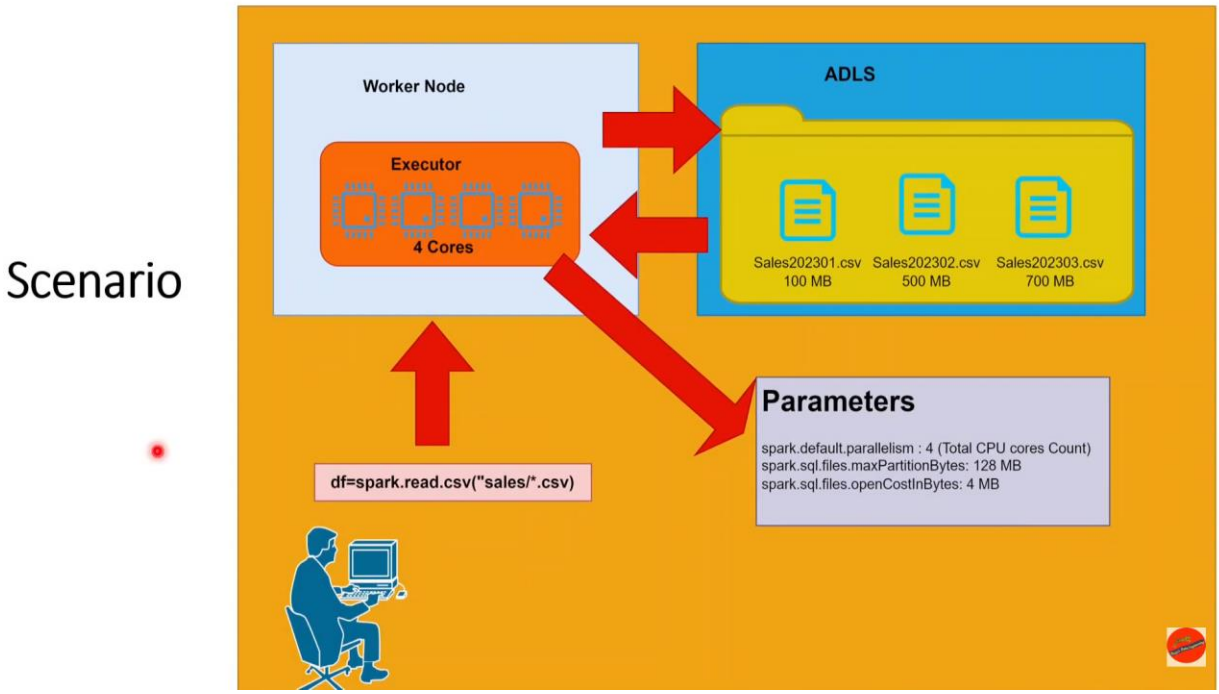


Let's say we are having 10 cores in our worker node and we has 1GB file, then what happens is this 1GB file will be divided by number of cores which will be approximately 100MB which means each core within the cluster can process 100MB approximately . bytesPerCore is nothing but whenever we are going to distribute total number of files across available number of cores or an available executors, how it can be distributed uniformly .



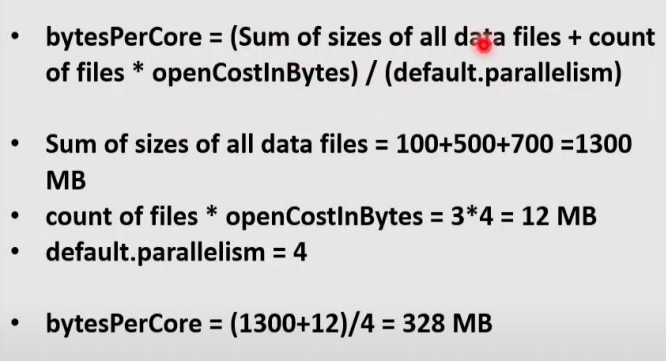
MaxSplitBytes is nothing but actually partition size. Let's assume in our storage we are having 1000's of files under a folder. Now we have to create a data frame by reading all the thousands of files then in order to convert those 1000 of files into physical partitions within spark environment. Based on maxSplitBytes value, final partition size will be decided.

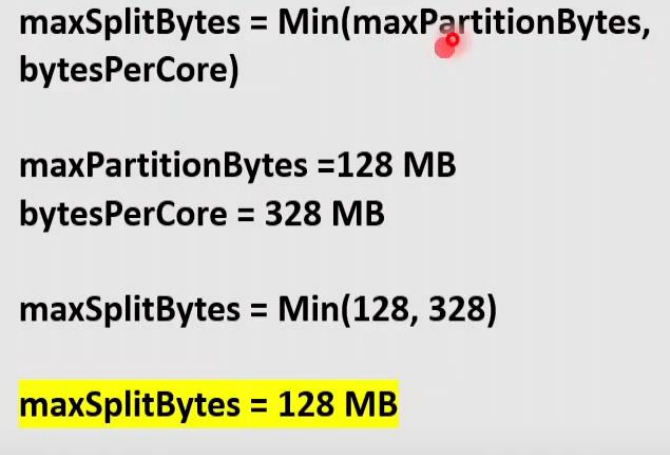
Let's say max partition size as 128MB but calculated bytes per core is coming as 80, then min of those 2 will be 80. 80 will be the partition size for that particular data frame reader.



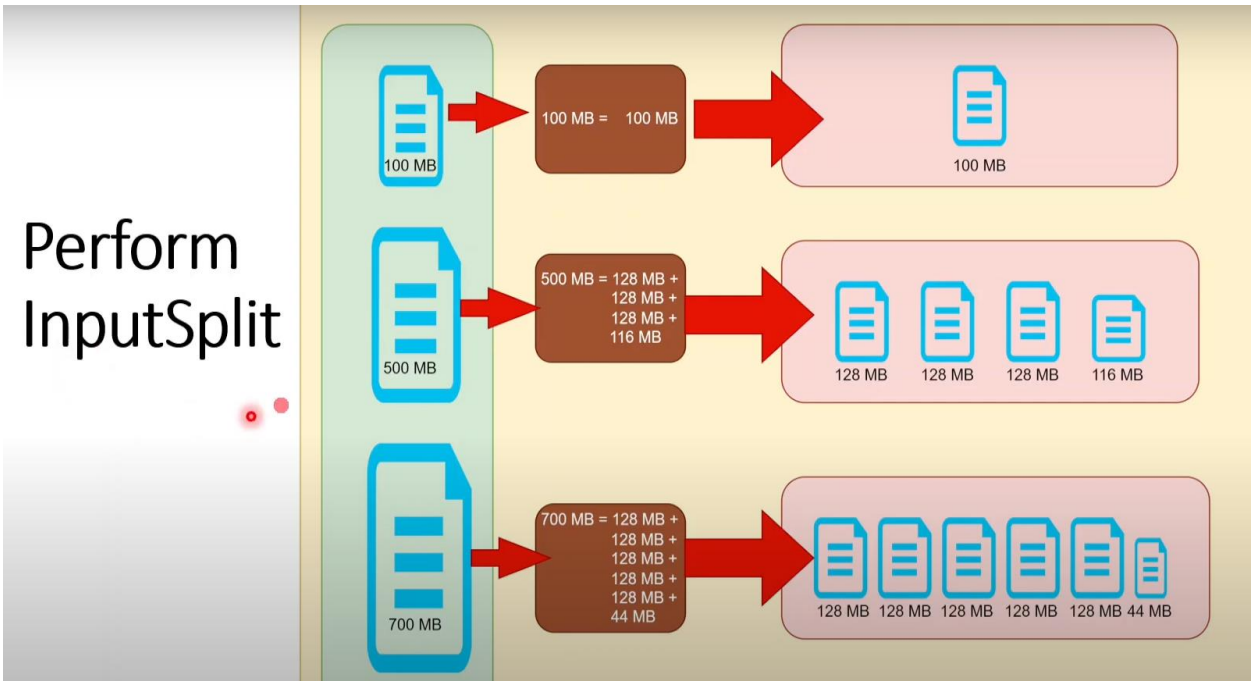
Let's assume one of the developer in development environment is passing spark application to the cluster, df = spark.read.csv("sales/*.csv). Basically, we are having 3 csv files under sales folder. We are having one sales folder under ADLS location. Within that folder we are having 3 files for each month (202301,202302,202303). These files are different in sizes, 1st file 100MB, 2nd file 500MB , 3rd file 700MB so now we have to create a data frame df by reading all the files under sales folder. So whenever particular code is given to cluster then driver is responsible to optimize and it will create multiple tasks and task will be submitted to executors. Let us assume in executor we are having 4 cores. And we are taking default parameters in this case. Not customizing any of the spark parameter. So, the default parallelism will be 4. Having only 4 cores. maxPartitionBytes will be 128MB and openCostInBytes is 4MB.

Whenever we are submitting spark application to the cluster which will contact ADLS, within ADLS we are having three files so for this use case first of all we need to calculate the Spark engine which will start calculating bytesPercore as shown below. In this use case in the first step Spark engine will calculate bytes per core.





In second step we have to calculate max split bytes.so for this particular data frame reader in this example we have given one of the spark calculation we are submitting on spark application to the spark cluster so finally the Spark engine has decided to create partitions of maximum size of 128 megabytes. Once value is decided as 128 megabits .



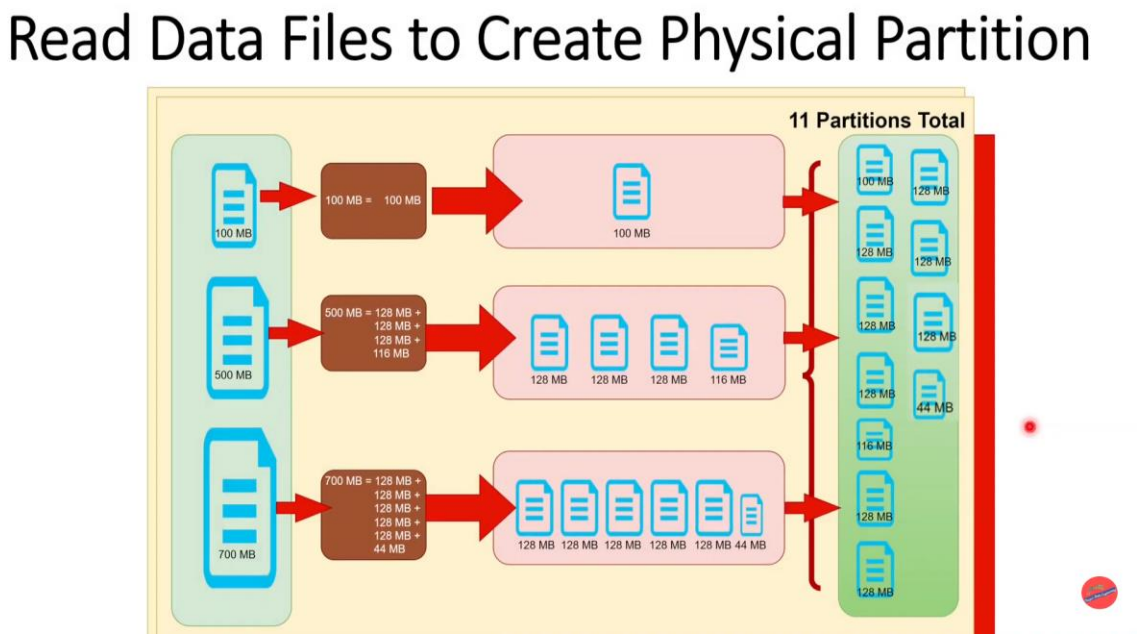
input split is one of the component in input format API and it will start splitting the files in the external storage logically so the first file it's already 128 megabytes but it can be splitted up to 128 megabits but it is already only 100MB so which is lesser than that so there is no point in splitting this file into multiple pieces so it will go as it is and coming to the 500 megabytes this is more than 128 megabytes so that's the reason it will be divided into three 128 megabytes then the last remaining part which will have only one 116 megabytes so this 500 megabytes will be logically divided into four pieces three 128 pieces and the last piece it will be 116 because now 500 megabytes last piece will contain only 116. and coming to the last file 700 megabytes again this 700MB is bigger than 128 megabits and it can be splitted so dividing five 128 megabytes and the final piece it will have only 44 megabytes so basically 700MB is divided into six pieces so the input split it is splitting all the files within the Azure data Lake storage and finally it has come to this conclusion the first file that is not splitted it is it is going as it is yes this is already less than the maximum partition size that is 128 megabytes and

coming to this 500 megabytes it is splitted into largely splitted into four pieces and this 700 megabits that is also logically splitted into six pieces so basically the logical division it will be like this 100 megabytes it is going as it is and these four files basically it will take the starting address and the end address for each file within this 500 megabytes ,basically this is being stored in one of the disk Cutters so basically it will take the address of starting byte and the end of the

byte also. So basically, this is divided into these many pieces so here we are having one + four + six = eleven pieces.

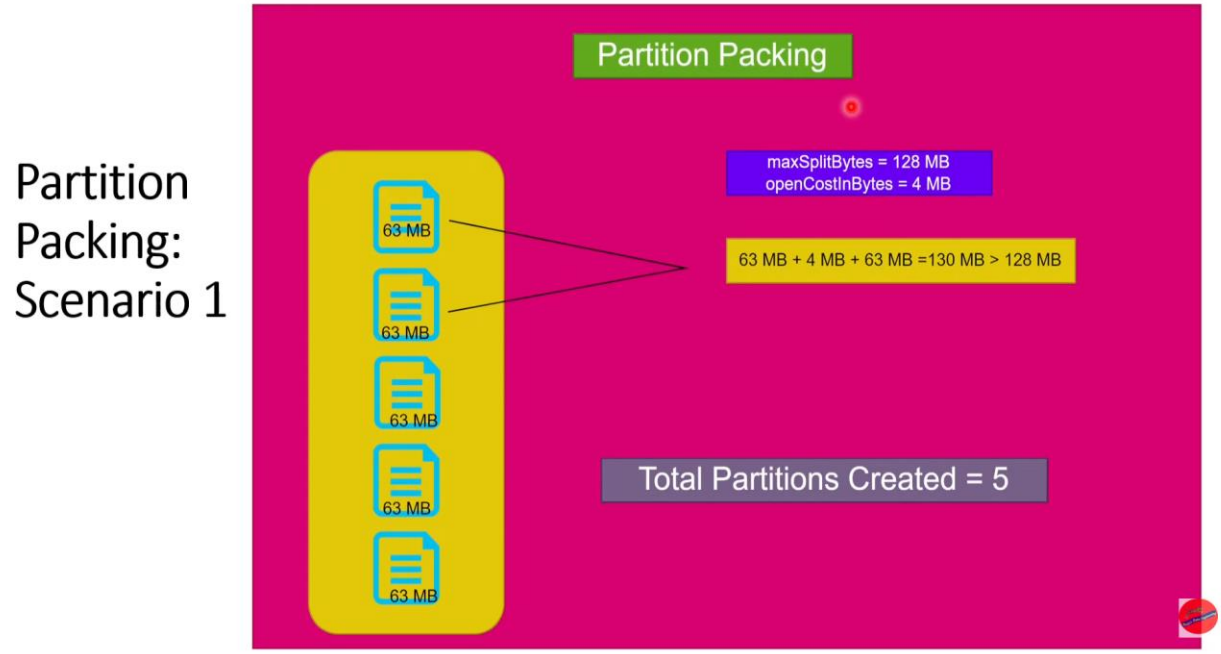
In the Next Step ,record reader will start packing all these files all these logical divisions into partitions so while doing that it will start ,it will try to combine some files in case it can be combined so that it cannot exceed the maximum size so for example in this case first of all this file 100 megabytes that is less than 128 megabytes still 128 megabytes can be added to this split then it will check if you are having any files Which is less than 28 megabytes so that it can be added but we don't have anything because in this particular file the smallest one is 116 but if you are going to add this 116(500MB file one) with 100(100MB file) it will be 216 it will exceed the maximum partition size of 128 so it cannot be combined with this one and coming to this one(700MB file 44MB) the smallest one is 44 even this 44 cannot be merged with this 100(100MB file) because it will become 144 which is again greater than 128 MB so this 100 megabytes will be considered as single partition and it will be transported to spark environment as it is and coming to this 500MB again these files will be start checking it is having the maximum bytes or not i.e.; 128 megabyte these are maximum and coming to the last one 116 it can be combined with any other file so that it can be a maximum of 128. but again, we don't have any other smaller file to combine with this one(100MB) so even this 128 MB,116 MB will go as one separate partition.

Similarly coming to this 44MB we don't have any other data file to combine with this(44MB) one so that we can increase the size within 128 megabytes we don't have any other file so even this 44MB will be considered as one of the single partition.



In spark environment so whenever above particular data frame is created so, that data frame will contain 11 partition and the first partition will contain 100 megabytes and in a few other partitions will have 128 megabytes and one of the partition will contain 116 and one more partition will have 44 megabytes so whenever you are creating data frame for this particular use case this is what we will see within a spark UI within spark memory .

PARTITION PACKING:



let's say in our Azure data Lake storage we are having five files CSV files each one of the file is having 63 megabytes in real time we are not going to have files of same size but just for this explanation just for this example we are considering having five files in all are of 63 megabytes.

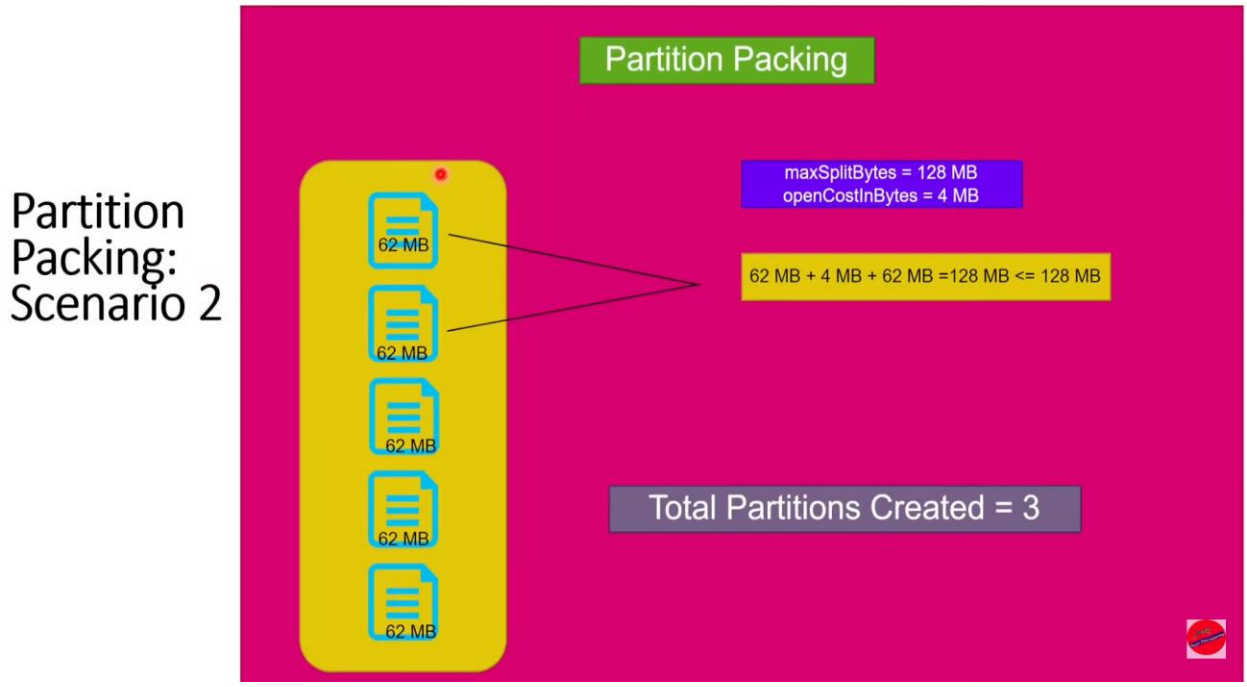
let us assume in our Spark engine it has calculated maximum split bytes that is 128 megabytes which means one partition can be up to 128 megabytes, that is a formula. now Spark engine will start combining multiple files in order to create a partition

first of all, it will try with 63MB that is already less than 128 megabytes so it will try to add one more file along with that in order to create maximum partition size of 128 megabytes but here whenever we are adding

63 megabytes with another 63 megabytes what happens is maximum size will be 126 which is coming less than 128 we might calculate in that way but that is not the right calculation because whenever we have to combine multiple files, Spark engine needs

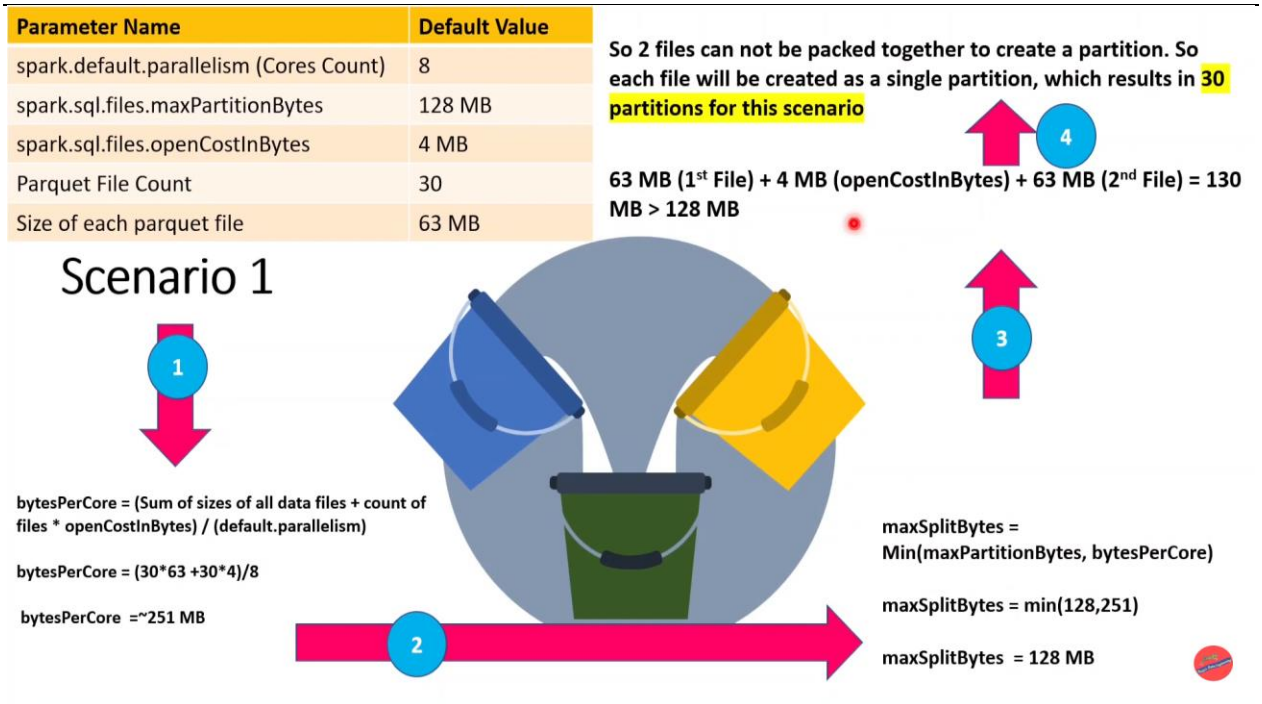
some buffer size that is openCostInBytes that is four megabytes so in this example Spark engine will take the first file that is 63 megabytes then in order to combine another file it will start adding the overhead or buffer memory that is four megabytes then it will start adding the second file 63. then the combination of these three files, in order to combine these two files(63+63+4MB), we need 130 megabytes so which is greater than 128 megabytes so in this case we cannot combine two different files this 63 file cannot be combined with anything so in this case 63 megabytes a file will go as one single partition even though the maximum partition size for this scenario is calculated as 128 megabytes but still this 63 megabytes file cannot be merged with another file in order to create single partition so each file will create single partition within a spark environment, so for this use case overall it will create five partitions so now we are having five files it cannot be merged with another file so each file will create separate partition so overall we will have five different partitions for this scenario. this is a partition packing.

--



let me take a different scenario for this partition packing . instead of 63 megabytes we are going to keep 62 megabyte. now we know what happens is we can assume the maximum split size is same as 128 megabytes and openCostInBytes that is 4 megabytes Now The Spark engine will look at the first file 63 megabytes that is less than 128 megabytes now it will try to add any of the other files then it will start trying to add the second file then the first file 62 megabytes in order to add the second file, it will add four megabytes(openCostInBytes) then after that it will add the second file 62

megabytes. now whenever we are calculating this 62+4+62 =128MB that is lesser than or equal to this 128 megabyte in our partition then in this use case two files can be combined into single partition so in this use case the first two files will be combined to create a single partition, another two files will be created to create another second partition and the third one it does not have any more files to combine so it will go as it is so overall in this use case three partitions will be created.



now we can see different scenarios.

so, the first scenario we can go with the default parallelism we are having eight cores so the parallelism that is eight and maximum partition bytes 128 megabytes and openCostInBytes that is four .

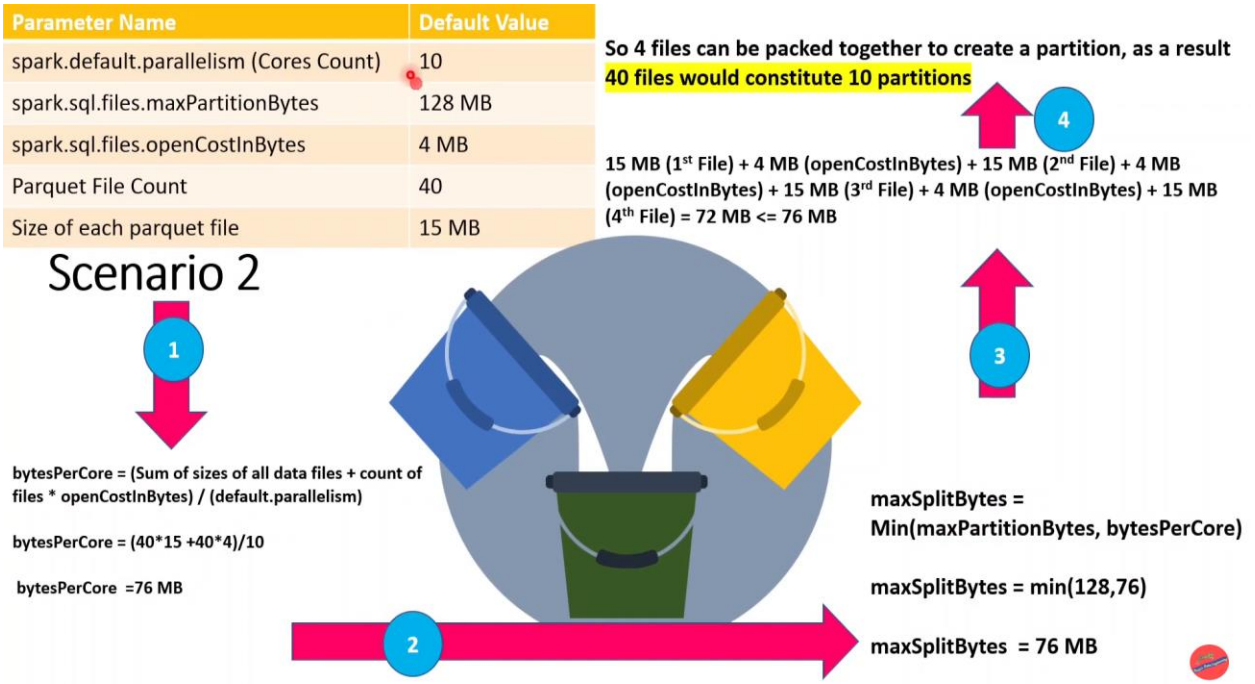
Let us assume in external storage we are having 30 parquet files, each is of 63MB. So spark engine will calculate bytesPerCore as shown above. In 2nd step it will calculate maxSplitBytes as shown in above screenshot.

This scenario has 30 partitions as shown above.

Scenario-2

Let us assume we have number of cores as 10 , maxPartitionBytes as 128MB by default, openCostInBytes as 4MB, parquet file count as 40 and size of each parquet file is of 15 MB. For this use case first spark engine will start calculating byesPerCore. Next step is creating maxSplitBytes . maxSplitBytes in this case is 76MB. Spark engine has decided to get partition with maximum size of 76MB.

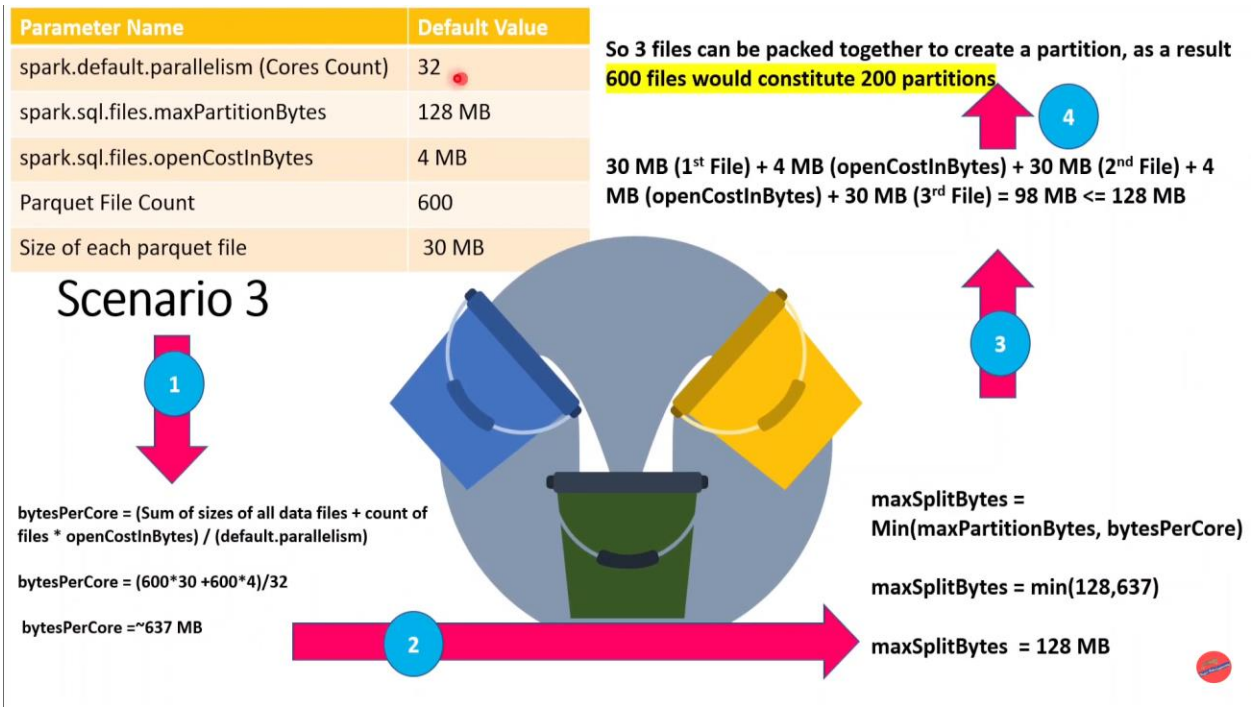
Parameter Name	Default Value
spark.default.parallelism (Cores Count)	10
spark.sql.files.maxPartitionBytes	128 MB
spark.sql.files.openCostInBytes	4 MB
Parquet File Count	40
Size of each parquet file	15 MB



now in this case Spark engine will consider the first file that is 15 megabytes then it will add openCostInBytes as 4 megabyte then it will start adding the second file 15 megabytes now it will check (15+4+15)it is coming 34. but still 34<=76 then it will try to add the one more file for that it has to add openCostInBytes as four megabytes then it will add one more file third file 15 megabytes again this will come around uh 53 bytes 53<=76 then it will try to add one more file for that it will

add four megabytes openCostInBytes then the fourth file 15 megabytes which is coming around 72 which is closer to 76. now we cannot add one more file fifth file we cannot add because it will exceed 76 megabyte so Spark engine will start at packing out of these 40 files every four files will be packed together to create a partition of maximum size 76. so, in this case four files will create single partition so 40 files will create 10 partition.

SCENARIO 3:



DATA TYPE ISSUE WHILE WRITING DATA FRAME DATA INTO AZURE SQL OR DWH

While writing data frame data into any DWH there are 2 modes, append and overwrite. In overwrite mode table structure would be trapped in the DWH and it would be recreated based on the structure from the data frame. Coming to append mode, it will retain table structure along with data in the DWH but on top of that it will just insert new records from the data frame. So, when we are performing overwrite what happens is Let's say we have a table in the DWH and it is having data types such as integer, string , date, and combination of these data types. In ADB in data frame we have all the columns in string type so what happens is when we are writing this data frame into DWH with overwrite mode what happens is it will drop the table structure in DWH and it will recreate. While recreating it will recreate with data types only varchar because in the data frame all the columns are string type. As a result, we will be recreating table in the DWH, it will create all the columns with varchar but this is not ideal, expected result. Let us assume after writing data frame data into DWH table there are certain procedures which will consume data from this table and it will do certain operations. While doing certain calculations it might fail because all the columns even integer columns are string type now. So, it might fail entire pipeline. Coming to append mode, append mode will not write the data successfully into azure synapse analytics if there is a mismatch between the data type between data frame and DWH. So,

while writing itself it will be failed. So, these are common data type issues while writing data frame data into any DWH or database.

In order to handle these scenarios, we have created an automated solution.

In Azure SQL database, we created one database. Within that we created one table as shown below. Its combination of various data types for this table.

```
CREATE TABLE [SalesLT].[Product_new](
    [ProductID] [int] NULL,
    [Name] [nvarchar](max) NULL,
    [ProductNumber] [nvarchar](max) NULL,
    [Color] [nvarchar](max) NULL,
    [StandardCost] [decimal](19, 4) NULL,
    [ListPrice] [decimal](19, 4) NULL,
    [Size] [nvarchar](max) NULL,
    [Weight] [decimal](8, 2) NULL,
    [ProductCategoryID] [int] NULL,
    [ProductModelID] [int] NULL,
    [SellStartDate] [datetime] NULL,
    [SellEndDate] [datetime] NULL,
    [DiscontinuedDate] [datetime] NULL,
    [ThumbNailPhoto] [nvarchar](max) NULL,
    [ThumbnailPhotoFileName] [nvarchar](max) NULL,
    [rowguid] [nvarchar](max) NULL,
    [ModifiedDate] [datetime] NULL
)
```

We just created table structure with no data in it. Now let’s say we have to populate data from data bricks data frame.

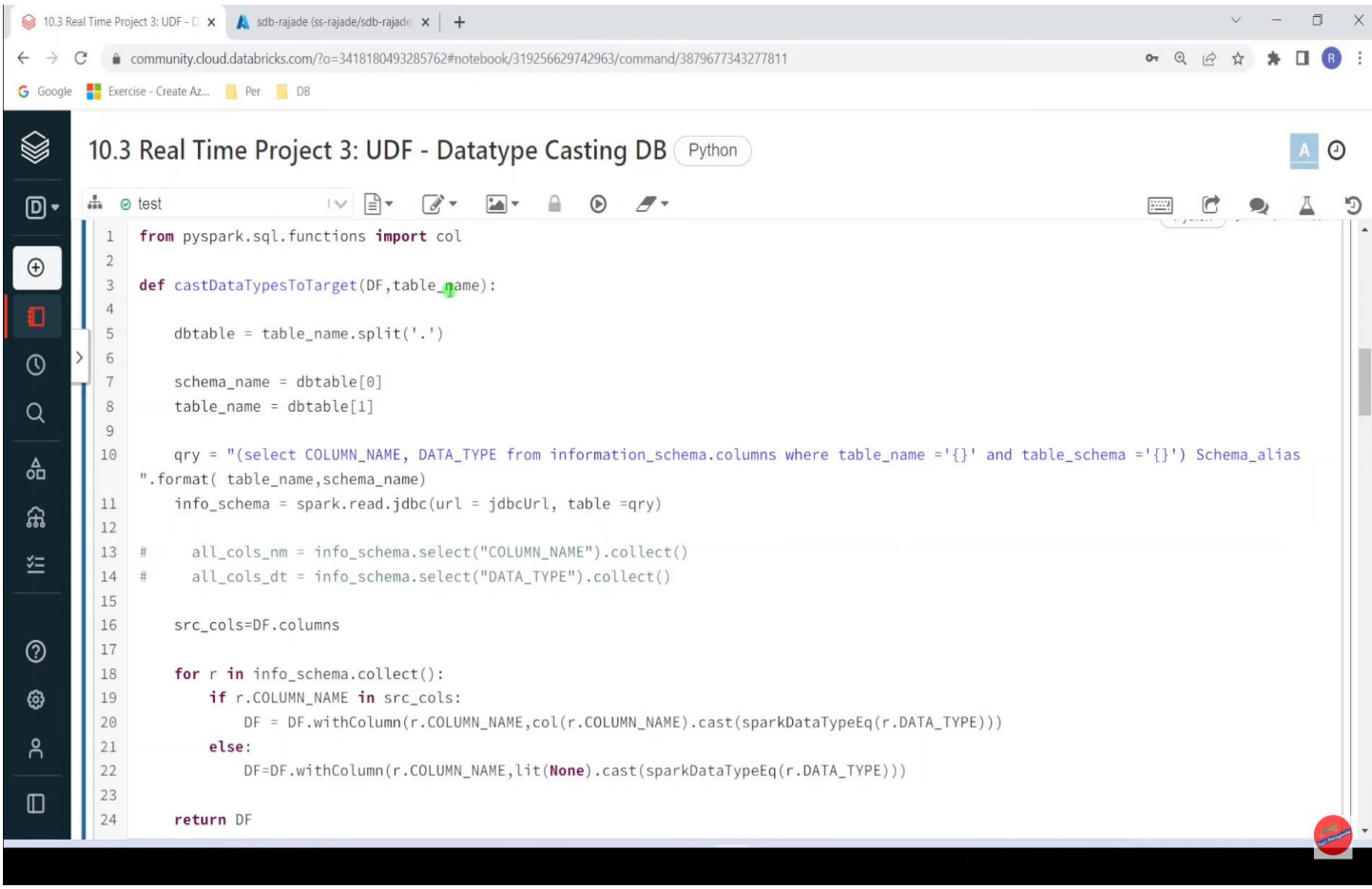
In ADB notebook we created one UDF sparkDataTypeEq (Eq=Equivalent). This UDF will take one datatype as input so this datatype is actually in database standard or azure synapse standard. It will take data type as input and it will compare and based on that it will return data bricks equivalent datatype. For example, if input is char or varchar then the output would be string type. In this UDF function, it will take data bricks datatype as input and it will give output of data bricks equivalent datatype. Finally, it will return output datatype that is equivalent of data frame.

```
from pyspark.sql.types import DateType,ByteType,LongType,StructType, IntegerType, StringType, TimestampType, DecimalType, DoubleType,
ShortType, BooleanType

def sparkDataTypeEq(dt):
    if "char" in dt:
        otpt_dt = StringType()
    elif "varchar" == dt:
        otpt_dt = StringType()
    elif "nvarchar" == dt:
        otpt_dt = StringType()
    elif "date" == dt:
        otpt_dt = DateType()
    elif "datetime" == dt:
        otpt_dt = DateType()
    elif "datetime2" == dt:
        otpt_dt = DateType()
    elif "bit" == dt:
        otpt_dt = BooleanType()
    elif "int" == dt:
        otpt_dt = IntegerType()
    elif "tinyint" == dt:
```

```
    elif "tinyint" == dt:
        otpt_dt = IntegerType()
    elif "smallint" == dt:
        otpt_dt = IntegerType()
    elif "bigint" == dt:
        otpt_dt = IntegerType()
    elif "decimal" == dt:
        otpt_dt = DecimalType(38,10)
    elif "numeric" == dt:
        otpt_dt = DecimalType(38,10)
    elif "float" == dt:
        otpt_dt = DoubleType()
    else:
        otpt_dt = StringType()
    return otpt_dt
```

--

In the 2nd UDF, we are going to give input of data frame which we are planning to write into DWH and also table name on which table we want to write.



Let's assume in our target table there are 10 columns but in our data frame we have only 8 columns even it will fail while appending because there are missing columns so as a result, we have to handle this scenario so for that purpose we have added if else condition . If any of the column is missing then it will populate that column with NULL value even casting will be done for that NULL value. So, at the end of this operation it will return one data frame that would be equivalent to DWH table.

```
jdbcHostname = "ss-rajade.database.windows.net"
jdbcPort = 1433
jdbcDatabase = "sdb-rajade"
jdbcUsername = "rajade"
jdbcPassword = "Training1"
jdbcDriver = "com.microsoft.sqlserver.jdbc.SQLServerDriver"

jdbcUrl = f"jdbc:sqlserver://{jdbcHostname}:{jdbcPort};databaseName={jdbcDatabase};user={jdbcUsername};password={jdbcPassword}"
```

```
productDF = spark.read.format("csv").option("header","true").option("inferschema","true").load("/FileStore/tables/advworks_product")
prodDF = productDF.select([col(c).cast("string") for c in productDF.columns])
prodDF.display()
```

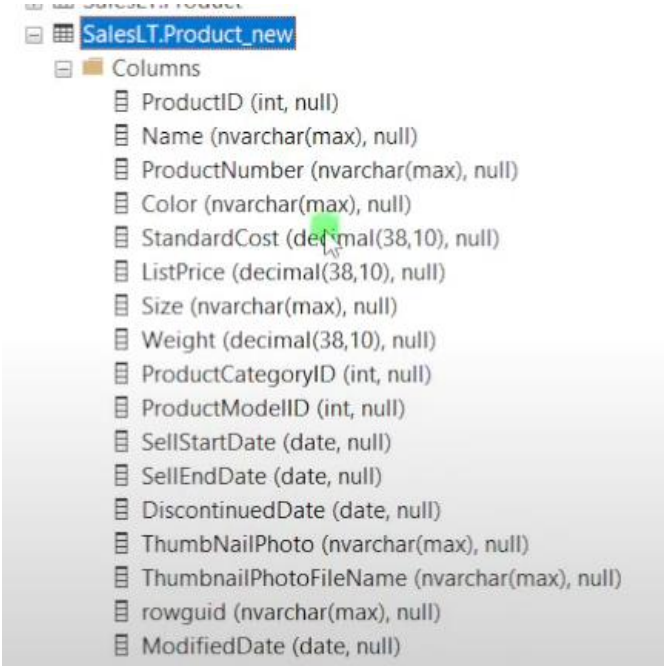
	ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size	Weight	Produc
1	680	HL Road Frame - Black, 58	FR-R92B-58	Black	1059.31	1431.5	58	1016.04	18
2	706	HL Road Frame - Red, 58	FR-R92R-58	Red	1059.31	1431.5	58	1016.04	18
3	707	Sport-100 Helmet, Red	HL-U509-R	Red	13.0863	34.99	null	null	35
4	708	Sport-100 Helmet, Black	HL-U509	Black	13.0863	34.99	null	null	35
5	709	Mountain Bike Socks, M	SO-B909-M	White	3.3963	9.5	M	null	27
6	710	Mountain Bike Socks, L	SO-B909-L	White	3.3963	9.5	L	null	27
7	711	Sport-100 Helmet, Blue	HL-U509-B	Blue	13.0863	34.99	null	null	35

```
1  prodDF.printSchema()
```

```
root
 |-- ProductID: string (nullable = true)
 |-- Name: string (nullable = true)
 |-- ProductNumber: string (nullable = true)
 |-- Color: string (nullable = true)
 |-- StandardCost: string (nullable = true)
 |-- ListPrice: string (nullable = true)
 |-- Size: string (nullable = true)
 |-- Weight: string (nullable = true)
 |-- ProductCategoryID: string (nullable = true)
 |-- ProductModelID: string (nullable = true)
 |-- SellStartDate: string (nullable = true)
 |-- SellEndDate: string (nullable = true)
 |-- DiscontinuedDate: string (nullable = true)
 |-- ThumbNailPhoto: string (nullable = true)
 |-- ThumbnailPhotoFileName: string (nullable = true)
 |-- rowguid: string (nullable = true)
 |-- ModifiedDate: string (nullable = true)
```

All the columns are of string data type.

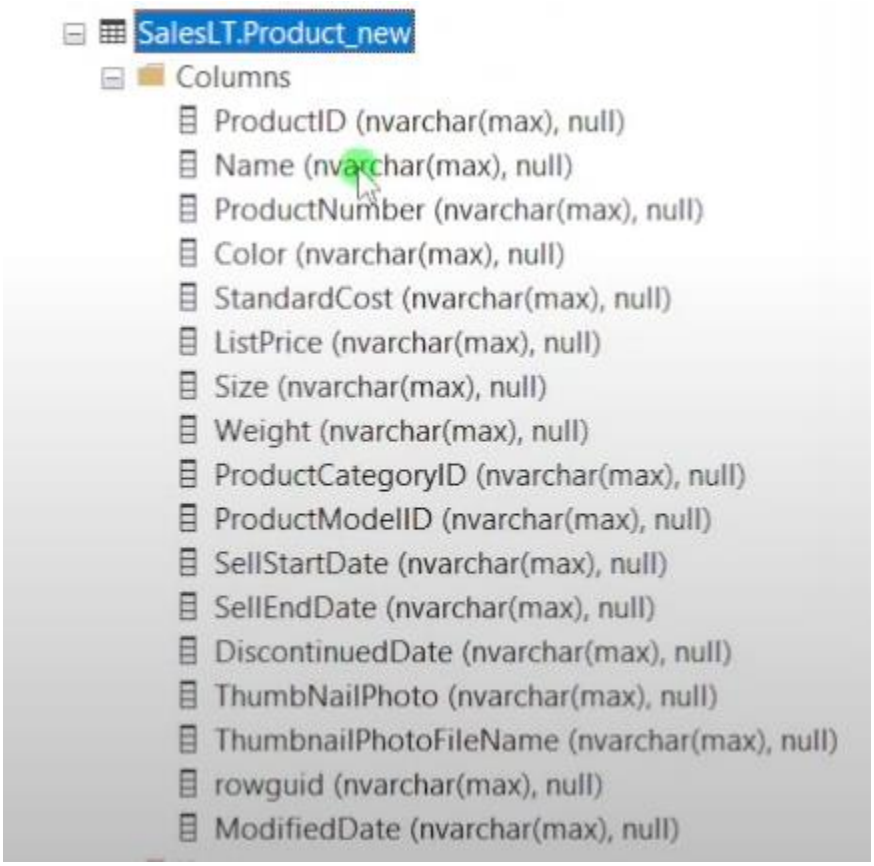
In Azure SQL Database, we are having the structure as below.



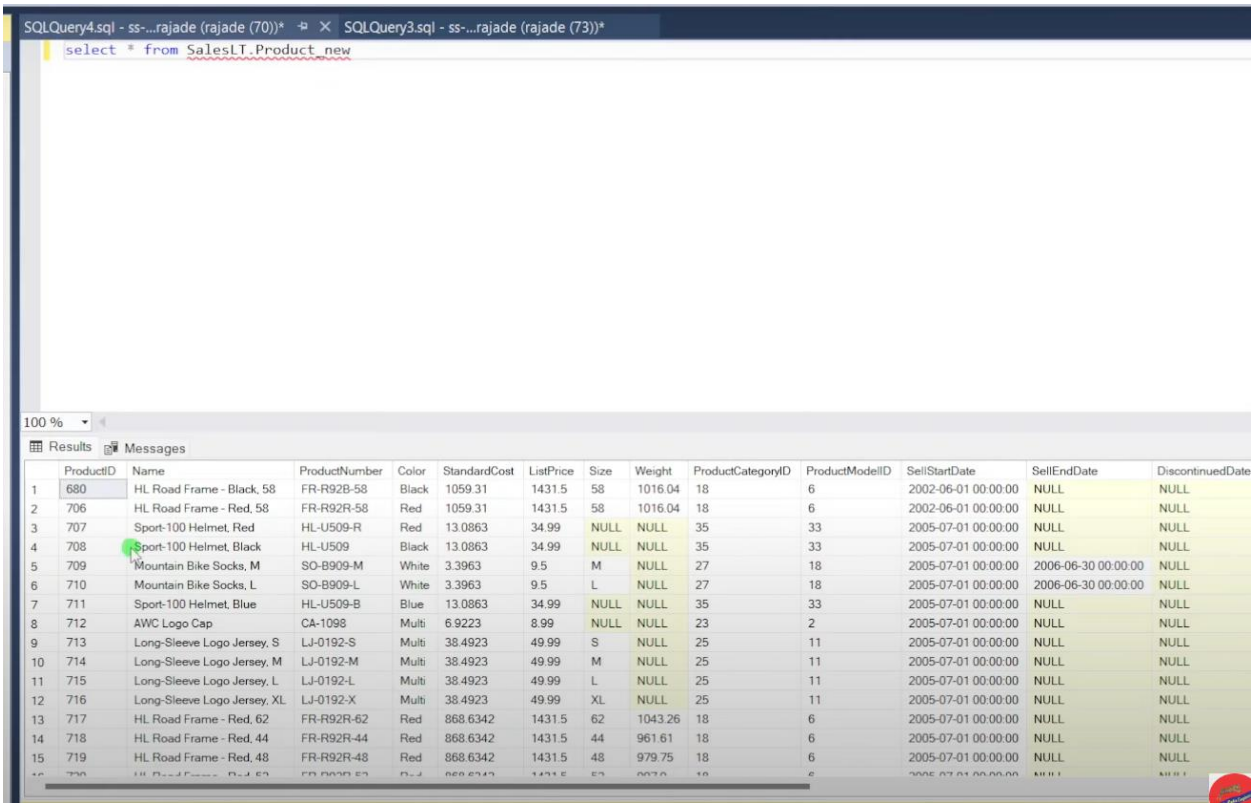
Data types in Azure SQL DB is of proper format where as in ADB data frame is of not in proper format.

```
prodDF.write\
    .format("jdbc")\
    .option("url", jdbcUrl)\
    .mode("overwrite")\
    .option("dbtable", "SalesLT.Product_new")\
    .save()
```

We are using mode as overwrite, so it will drop table in Azure SQL DB in DWH and it will recreate. While recreating, it will just follow the data type of this databricks data frame. So here in databricks everything is string so it will convert datatypes to varchar in DWH.



All columns converted into varchar which is not expected one.



	ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size	Weight	ProductCategoryID	ProductModelID	SellStartDate	SellEndDate	DiscontinuedDate
1	680	HL Road Frame - Black, 58	FR-R92B-58	Black	1059.31	1431.5	58	1016.04	18	6	2002-06-01 00:00:00	NULL	NULL
2	706	HL Road Frame - Red, 58	FR-R92R-58	Red	1059.31	1431.5	58	1016.04	18	6	2002-06-01 00:00:00	NULL	NULL
3	707	Sport-100 Helmet, Red	HL-U509-R	Red	13.0863	34.99	NULL	NULL	35	33	2005-07-01 00:00:00	NULL	NULL
4	708	Sport-100 Helmet, Black	HL-U509	Black	13.0863	34.99	NULL	NULL	35	33	2005-07-01 00:00:00	NULL	NULL
5	709	Mountain Bike Socks, M	SO-B909-M	White	3.3963	9.5	M	NULL	27	18	2005-07-01 00:00:00	2006-06-30 00:00:00	NULL
6	710	Mountain Bike Socks, L	SO-B909-L	White	3.3963	9.5	L	NULL	27	18	2005-07-01 00:00:00	2006-06-30 00:00:00	NULL
7	711	Sport-100 Helmet, Blue	HL-U509-B	Blue	13.0863	34.99	NULL	NULL	35	33	2005-07-01 00:00:00	NULL	NULL
8	712	AWC Logo Cap	CA-1098	Multi	6.9223	8.99	NULL	NULL	23	2	2005-07-01 00:00:00	NULL	NULL
9	713	Long-Sleeve Logo Jersey, S	LJ-0192-S	Multi	38.4923	49.99	S	NULL	25	11	2005-07-01 00:00:00	NULL	NULL
10	714	Long-Sleeve Logo Jersey, M	LJ-0192-M	Multi	38.4923	49.99	M	NULL	25	11	2005-07-01 00:00:00	NULL	NULL
11	715	Long-Sleeve Logo Jersey, L	LJ-0192-L	Multi	38.4923	49.99	L	NULL	25	11	2005-07-01 00:00:00	NULL	NULL
12	716	Long-Sleeve Logo Jersey, XL	LJ-0192-X	Multi	38.4923	49.99	XL	NULL	25	11	2005-07-01 00:00:00	NULL	NULL
13	717	HL Road Frame - Red, 62	FR-R92R-62	Red	868.6342	1431.5	62	1043.26	18	6	2005-07-01 00:00:00	NULL	NULL
14	718	HL Road Frame - Red, 44	FR-R92R-44	Red	868.6342	1431.5	44	961.61	18	6	2005-07-01 00:00:00	NULL	NULL
15	719	HL Road Frame - Red, 48	FR-R92R-48	Red	868.6342	1431.5	48	979.75	18	6	2005-07-01 00:00:00	NULL	NULL

So now in azure SQL DB, will drop table and recreate with proper data types.

```
drop table [SalesLT].[Product_new]

CREATE TABLE [SalesLT].[Product_new](
    [ProductID] [int] NULL,
    [Name] [nvarchar](max) NULL,
    [ProductNumber] [nvarchar](max) NULL,
    [Color] [nvarchar](max) NULL,
    [StandardCost] [decimal](19, 4) NULL,
    [ListPrice] [decimal](19, 4) NULL,
    [Size] [nvarchar](max) NULL,
    [Weight] [decimal](8, 2) NULL,
    [ProductCategoryID] [int] NULL,
    [ProductModelID] [int] NULL,
    [SellStartDate] [datetime] NULL,
    [SellEndDate] [datetime] NULL,
    [DiscontinuedDate] [datetime] NULL,
    [ThumbNailPhoto] [nvarchar](max) NULL,
    [ThumbnailPhotoFileName] [nvarchar](max) NULL,
    [rowguid] [nvarchar](max) NULL,
    [ModifiedDate] [datetime] NULL
)
```

Now data type is in proper format for all columns,

```
df =castDataTypesToTarget(prodDF,"SalesLT.Product_new")
display(df)
```

(2) Spark Jobs

	ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size	Weight
1	680	HL Road Frame - Black, 58	FR-R92B-58	Black	1059.3100000000	1431.5000000000	58	1016.040000000
2	706	HL Road Frame - Red, 58	FR-R92R-58	Red	1059.3100000000	1431.5000000000	58	1016.040000000
3	707	Sport-100 Helmet, Red	HL-U509-R	Red	13.0863000000	34.9900000000	null	null
4	708	Sport-100 Helmet, Black	HL-U509	Black	13.0863000000	34.9900000000	null	null
5	709	Mountain Bike Socks, M	SO-B909-M	White	3.3963000000	9.5000000000	M	null
6	710	Mountain Bike Socks, L	SO-B909-L	White	3.3963000000	9.5000000000	L	null
7	711	Sport-100 Helmet, Blue	HL-U509-B	Blue	13.0863000000	34.9900000000	null	null

Now it has converted the data type.

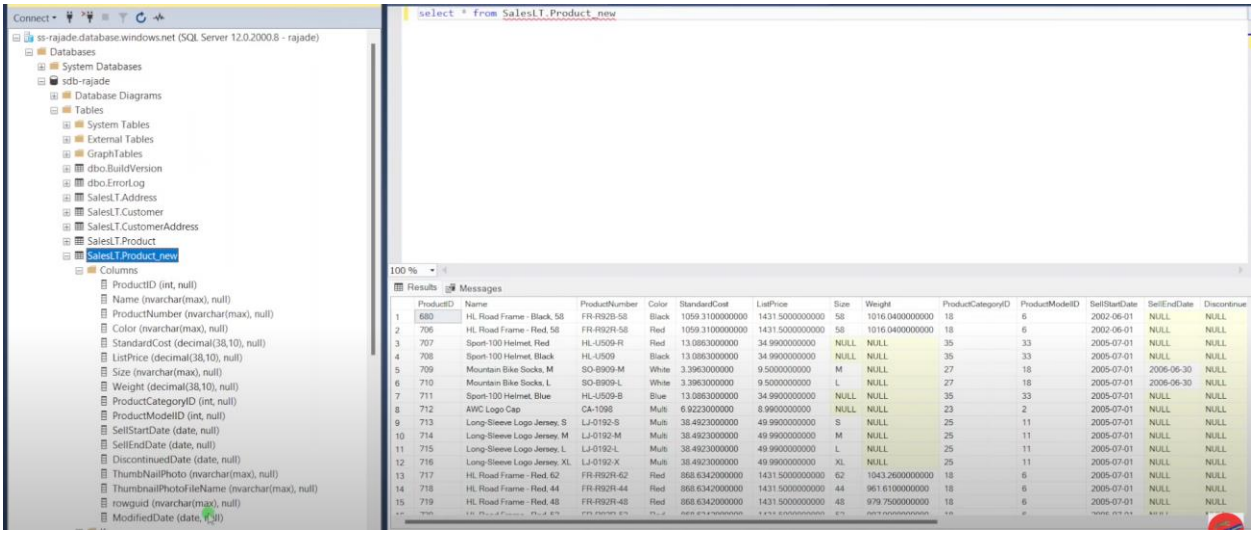
```
df.printSchema()
```

```
root
 |-- ProductID: integer (nullable = true)
 |-- Name: string (nullable = true)
 |-- ProductNumber: string (nullable = true)
 |-- Color: string (nullable = true)
 |-- StandardCost: decimal(38,10) (nullable = true)
 |-- ListPrice: decimal(38,10) (nullable = true)
 |-- Size: string (nullable = true)
 |-- Weight: decimal(38,10) (nullable = true)
 |-- ProductCategoryID: integer (nullable = true)
 |-- ProductModelID: integer (nullable = true)
 |-- SellStartDate: date (nullable = true)
 |-- SellEndDate: date (nullable = true)
 |-- DiscontinuedDate: date (nullable = true)
 |-- ThumbNailPhoto: string (nullable = true)
 |-- ThumbnailPhotoFileName: string (nullable = true)
 |-- rowguid: string (nullable = true)
 |-- ModifiedDate: date (nullable = true)
```

Previously it has all with same data type as string , now after passing data frame into user defined function castDataTypesToTarget, it has converted all the columns into proper data type.

Now we will write this data into DWH using below code as shown below,

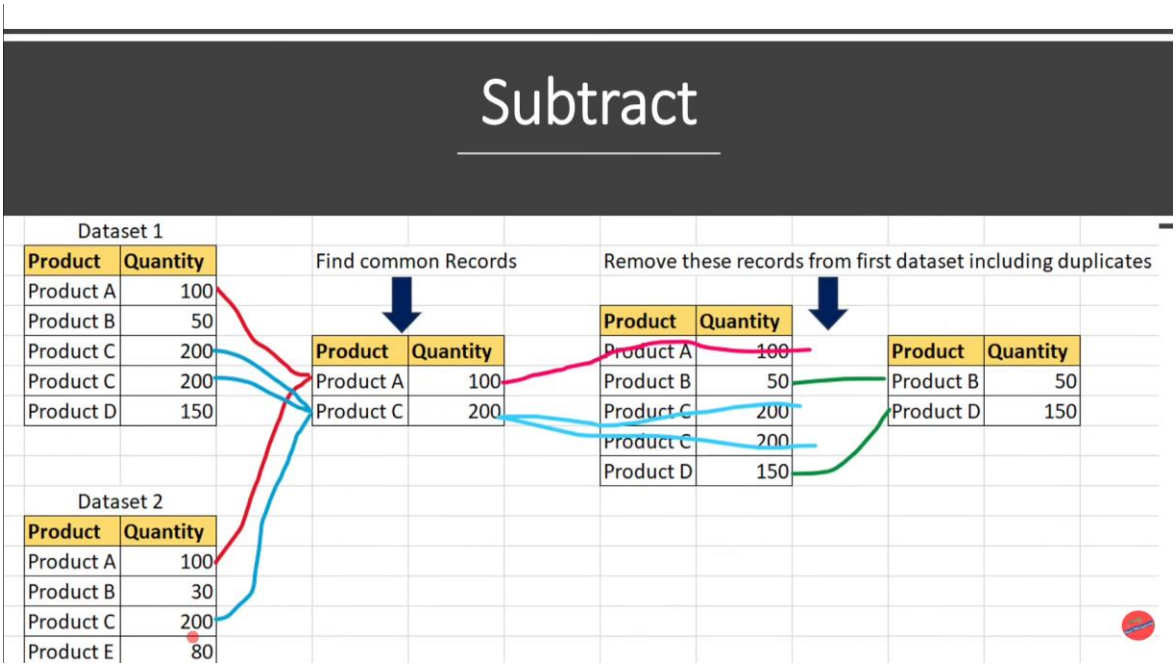
```
df.write\
    .format("jdbc")\
    .option("url", jdbcUrl)\
    .mode("overwrite")\
    .option("dbtable", "SalesLT.Product_new")\
    .save()
```

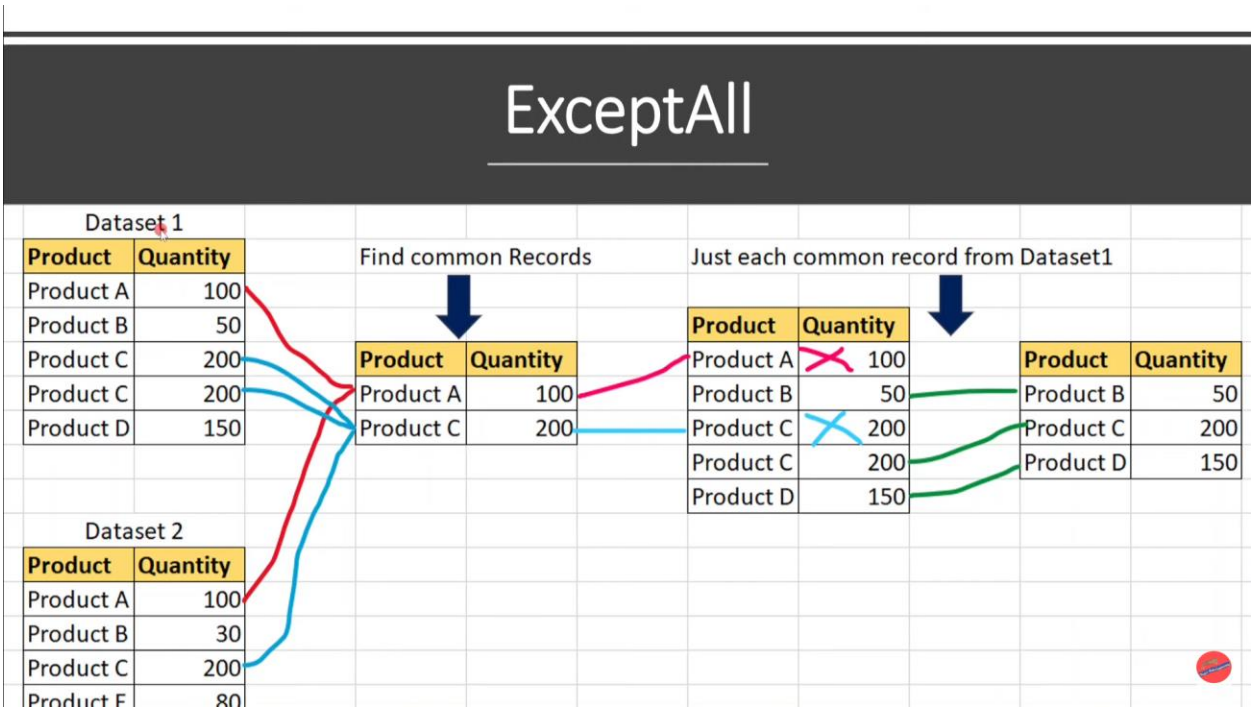


Now data got stored in Azure SQL DB with proper data types.

SUBTRACT VS EXCEPTALL

Subtract and ExceptAll are quite similar to each other but there is a little difference. Subtract and ExceptAll transformations create new data frame by containing rows from one data frame which are not present in second data frame. Basically, these transformations are used to identify difference between two data frames. Coming to the difference , while creating new data frame by retaining only the records from the first data frame , it will not consider the duplicates, so it will remove even if we are having duplicate records which is common between two data frames. But coming to ExceptAll, it will preserve duplicates. This is the major difference between these 2 functions.





If there is a duplicate , it will retain that duplicate in ExceptAll.

Create Sample Dataframes

```
1
2   # Sample sales data
3   data1 = [("Product A", 100),
4            ("Product B", 50),
5            ("Product C", 200),
6            ("Product C", 200),
7            ("Product D", 150)]
8
9   data2 = [("Product A", 100),
10           ("Product B", 30),
11           ("Product C", 200),
12           ("Product E", 80)]
13
14  # Create DataFrames from the sample data
15  sourceDF = spark.createDataFrame(data1, ["Product", "Quantity"])
16  targetDF = spark.createDataFrame(data2, ["Product", "Quantity"])
```

```
sourceDF.display()
targetDF.display()
```

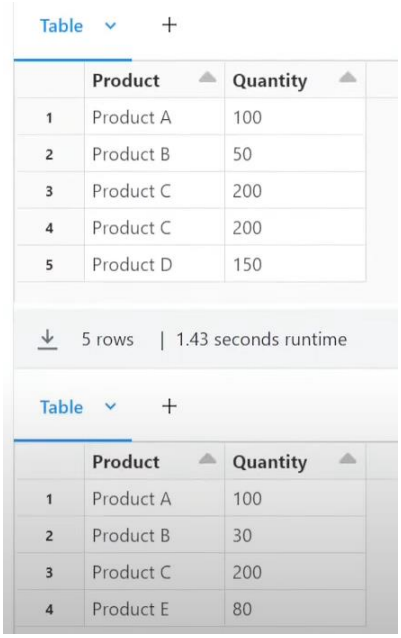


Table +

	Product	Quantity
1	Product A	100
2	Product B	50
3	Product C	200
4	Product C	200
5	Product D	150

5 rows | 1.43 seconds runtime

Table +

	Product	Quantity
1	Product A	100
2	Product B	30
3	Product C	200
4	Product E	80

Subtract - Not Preserving Duplicates

```
1  resultDF = sourceDF.subtract(targetDF)
2  resultDF.display()
```

(5) Spark Jobs

resultDF: pyspark.sql.dataframe.DataFrame = [Product: string, C

Table +

	Product	Quantity
1	Product B	50
2	Product D	150

2 rows | 2.51 seconds runtime

ExceptAll - Preserves Duplicates

```
1  resultDF = sourceDF.exceptAll(targetDF)
2  resultDF.display()
```

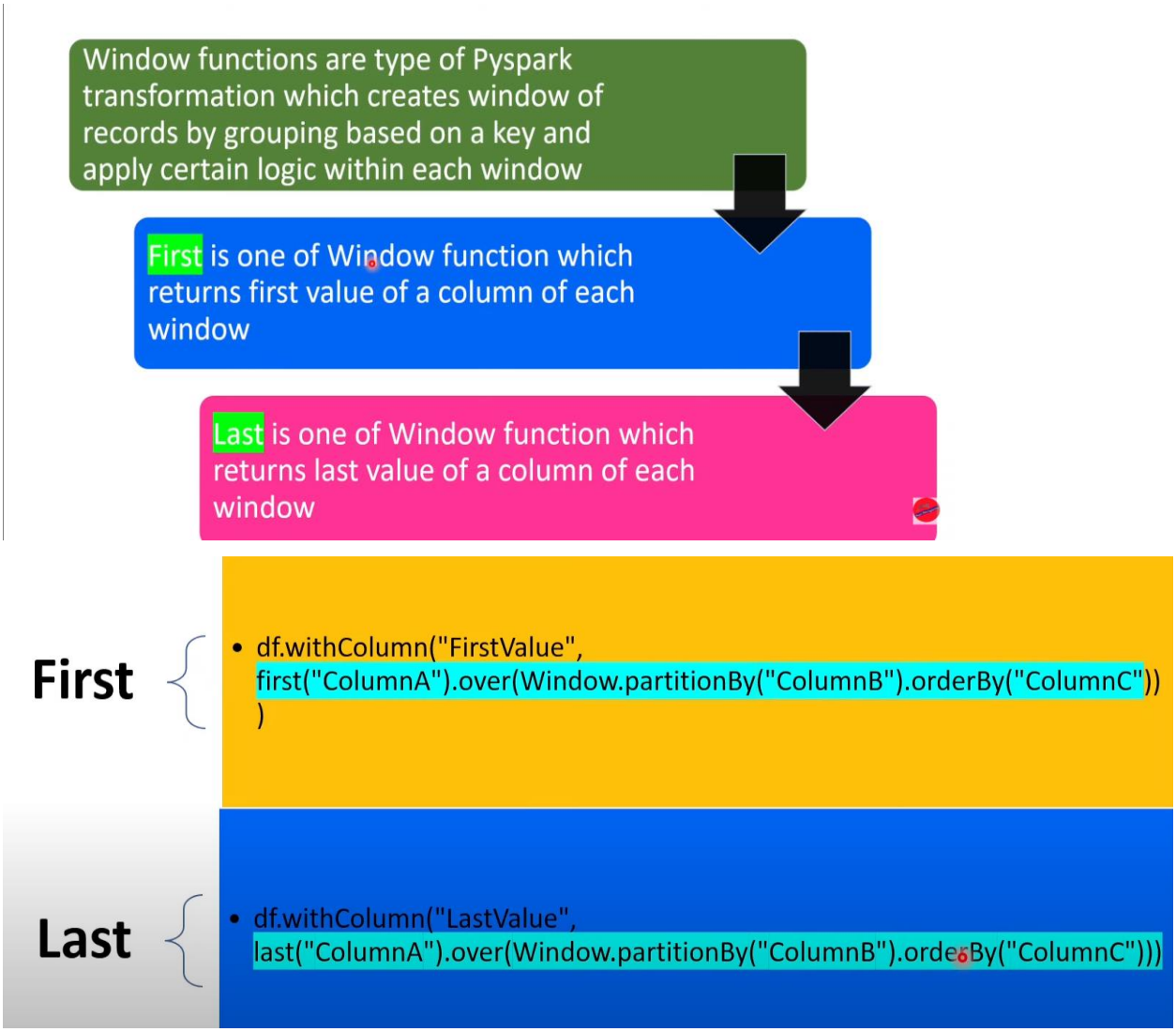
(2) Spark Jobs

resultDF: pyspark.sql.dataframe.DataFrame = [Product: s

Table

	Product	Quantity
1	Product B	50
2	Product C	200
3	Product D	150

WINDOW FUNCTION : FIRST AND LAST




```
#CREATE SAMPLE DATAFRAME
from pyspark.sql.functions import *
data =[
      (“C1”,”2023-06-01”,100.0),
      (“C1”,”2023-06-02”,150.0),
      (“C1”,”2023-06-03”,200.0),
      (“C2”,”2023-06-01”,50.0),
      (“C2”,”2023-06-02”,75.0),
      (“C2”,”2023-06-03”,100.0)
]
df = spark.createDataFrame(data, [“customer_id”,”transaction_date”,”amount”])
#convert transaction_date column to date type
```

```
df = df.withColumn("transaction_date",to_date(col("transaction_date")))
df.display()
```

	customer_id	transaction_date	amount
1	C1	2023-06-01	100
2	C1	2023-06-02	150
3	C1	2023-06-03	200
4	C2	2023-06-01	50
5	C2	2023-06-02	75
6	C2	2023-06-03	100

```
# Using PySpark Window functions
from pyspark.sql.window import Window

windowSpec = Window.partitionBy("customer_id")

result_df = df.withColumn("first_transaction_date", first("transaction_date").over(windowSpec)) \
              .withColumn("last_transaction_date", last("transaction_date").over(windowSpec))

result_df.display()
```

	customer_id	transaction_date	amount	first_transaction_date	last_transaction_date
1	C1	2023-06-01	100	2023-06-01	2023-06-03
2	C1	2023-06-02	150	2023-06-01	2023-06-03
3	C1	2023-06-03	200	2023-06-01	2023-06-03
4	C2	2023-06-01	50	2023-06-01	2023-06-03
5	C2	2023-06-02	75	2023-06-01	2023-06-03
6	C2	2023-06-03	100	2023-06-01	2023-06-03

Find First and Last Transaction date for Each Customer

Python

```
result_df = df.withColumn("first_transaction_date", first("transaction_date").over(windowSpec)) \
              .withColumn("last_transaction_date", last("transaction_date").over(windowSpec)).drop("transaction_date",
              "amount").distinct()

result_df.display()
```

(2) Spark Jobs

result_df: pyspark.sql.dataframe.DataFrame = [customer_id: string, first_transaction_date: date ... 1 more field]

Table +

	customer_id	first_transaction_date	last_transaction_date
1	C1	2023-06-01	2023-06-03
2	C2	2023-06-01	2023-06-03

How to Perform same transformation using Spark SQL?

Cmd 5

Convert Dataframe to View

```
# Register the DataFrame as a temporary view
df.createOrReplaceTempView("transactions")
```

Cmd 6

SQL Window Function

```
%sql

    SELECT distinct customer_id,
        FIRST(transaction_date) OVER (PARTITION BY customer_id) AS first_transaction_date,
        LAST(transaction_date) OVER (PARTITION BY customer_id) AS last_transaction_date
    FROM transactions
    ORDER BY customer_id
```

	customer_id	first_transaction_date	last_transaction_date
1	C1	2023-06-01	2023-06-03
2	C2	2023-06-01	2023-06-03

HOW TO READ FIXED LENGTH TEXT FILE IN DATABRICKS DEVELOPMENT

In most of the projects, still some portion of data is coming through text file, the data could be metadata, some master data management or some configuration data . These are some examples of data coming through text file. fixed length text file means we are not going to have any delimiter in the text file. Instead of that each and every column in that particular text file is going to be fixed. So we are going to get fixed character of data for each column as a result we will have fixed number of characters for each record in the text file.

CODE:

```
%fs
ls /FileStore/
```

Table +

	path	name	size	modificationTime
1	dbfs:/FileStore/city_temperature_data.txt	city_temperature_data.txt	207	1688542731000
2	dbfs:/FileStore/jars/	jars/	0	0
3	dbfs:/FileStore/tables/	tables/	0	0
4	dbfs:/FileStore/twin1.json	twin1.json	14286	1670495356000
5	dbfs:/FileStore/twin1.stx	twin1.stx	14286	1670495465000

--

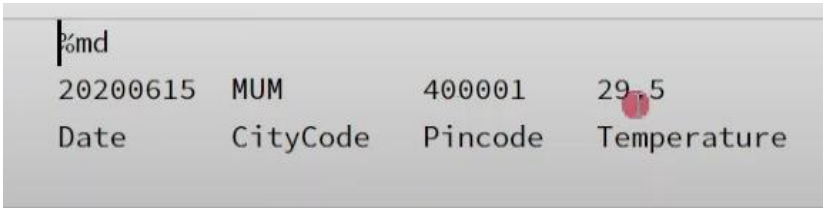
%fs

head dbfs:/FileStore/city_temperature_data.txt

output:

```
20200615MUM40000129.5
20200715MUM40000128.5
20200815MUM40000129.2
20200615DEL11002034.3
20200715DEL11002033.5
20200815DEL11002035.0
20200615CHN60000131.5
20200715CHN60000130.5
20200815CHN60000130.9
```

If we see this data, it is divided into multiple columns. First 8 characters is part of field date YYYYMMDD. Next 3 characters MUM is part of city code. Next 6 characters is pin code. Final 4 characters including dot is temperature as shown below



This is how data divided into multiple columns.

#Read Text File

#Read fixed length file

df = spark.read.text("dbfs:/FileStore/city_temparature_data.txt")

display(df)

	value
1	20200615MUM40000129.5
2	20200715MUM40000128.5
3	20200815MUM40000129.2
4	20200615DEL11002034.3
5	20200715DEL11002033.5
6	20200815DEL11002035.0
7	20200615CHN60000131.5

Now our requirement is we need to split this data into multiple columns according to our schema. So, for that we are going to create one dictionary. This is a python dictionary. So, we have to use curly braces. In curly braces, we have to keep key value pairs. Key is column name and coming to value its going to have combination of starting position of each column and what is the length of that particular column.

#Define schema and field positions using a dictionary

fieldDetails = {“Date” :(1,8), “CityCode” :(9,3) ,“PinCode” :(12,6) ,“Temperature” : (18,4)}

Define Schema and field positions using a dictionary

```
1
2  fieldDetails = {"Date": (1, 8),"CityCode": (9, 3),"PinCode": (12, 6),"Temperature": (18, 4)}
3
```

Now we defined schema and field position.

--

Now we will use for loop. Using for loop, we are going to iterate all the fields one by one using dictionary and then we are going to add a new column .

Splitting the Fixed Length Text Into Columns

```
1  from pyspark.sql.functions import substring
2
3  for field, (start, length) in fieldDetails.items():
4      df = df.withColumn(field, substring(df["value"], start, length))
5
6  df =df.drop("value")
7
8  # Show the DataFrame
9  df.display()
```

	Date	CityCode	PinCode	Temperature
1	20200615	MUM	400001	29.5
2	20200715	MUM	400001	28.5
3	20200815	MUM	400001	29.2
4	20200615	DEL	110020	34.3
5	20200715	DEL	110020	33.5
6	20200815	DEL	110020	35.0
7	20200615	CHN	600001	31.5

9 rows | 0.70 seconds runtime

If we are having this type of requirement in our project, We can create UDF and we can execute this.

SKIP FIRST N RECORDS IN CSV FILE

CSV data is generated through various source systems. It is converted into csv from excel sheet. While generating those data in csv file in certain scenarios junk records are getting inserted in the first few records of the csv file. As a result, if we are going to read entire data, it is going to have corrupted data within data frame. So, in order to avoid that, we need to skip first few records.

Review Data of CSV File

```
%fs
head dbfs:/FileStore/tables/baby_names_id.csv
```

```
Id,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
8,2007,ZOE,NEW YORK,F,55
9,2007,ZOE,NASSAU,F,15
10,2007,ZOE,ERIE,F,6
11,2007,ZOE,SUFFOLK,F,14
12,2007,ZOE,KINGS,F,34
13,2007,ZOE,MONROE,F,9
14,2007,ZOE,QUEENS,F,26
15,2007,ZOE,ALBANY,F,5
16,2007,ZISSY,ROCKLAND,F,5
```

Having total of 250 records.

Read CSV File and Display Data and Record Count

```
df =( spark.read.format("csv")
.option("inferSchema", True)
.option("header", True )
.option("sep",",")
.load("dbfs:/FileStore/tables/baby_names_id.csv"))

display(df)
```

Table +

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

250 rows | 2.97 seconds runtime

```
print(df.count())
```

output : 250

--

Skip First 4 Records While Reading CSV File and Display Data and Record Count

```
skipDF =( spark.read.format("csv")
.option("inferSchema", True)
.option("header", True )
.option("sep",",")
.option("skipRows", 4)
.load("dbfs:/FileStore/tables/baby_names_id.csv"))

display(skipDF)

print(skipDF.count())
```

Table +

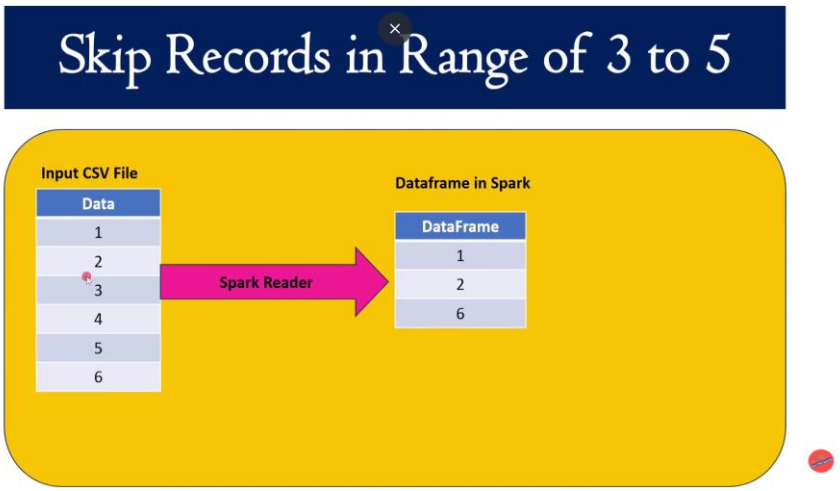
	4	2007	ZOEY	ERIE	F	9
1	5	2007	ZOE	ULSTER	F	5
2	6	2007	ZOE	WESTCHESTER	F	24
3	7	2007	ZOE	BRONX	F	13
4	8	2007	ZOE	NEW YORK	F	55
5	9	2007	ZOE	NASSAU	F	15
6	10	2007	ZOE	ERIE	F	6
7	11	2007	ZOE	SUFFOLK	F	14

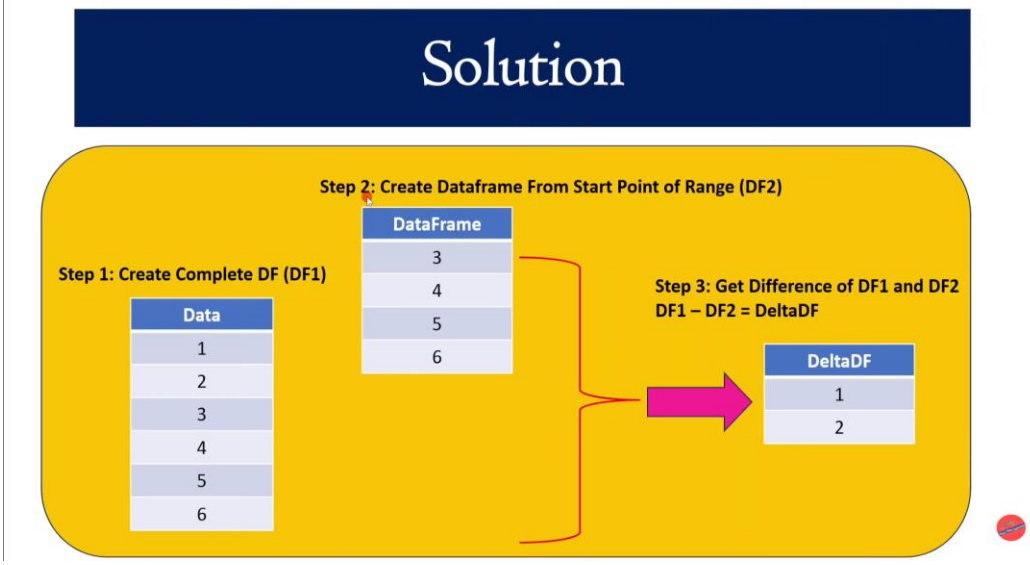
246 rows | 1.56 seconds runtime

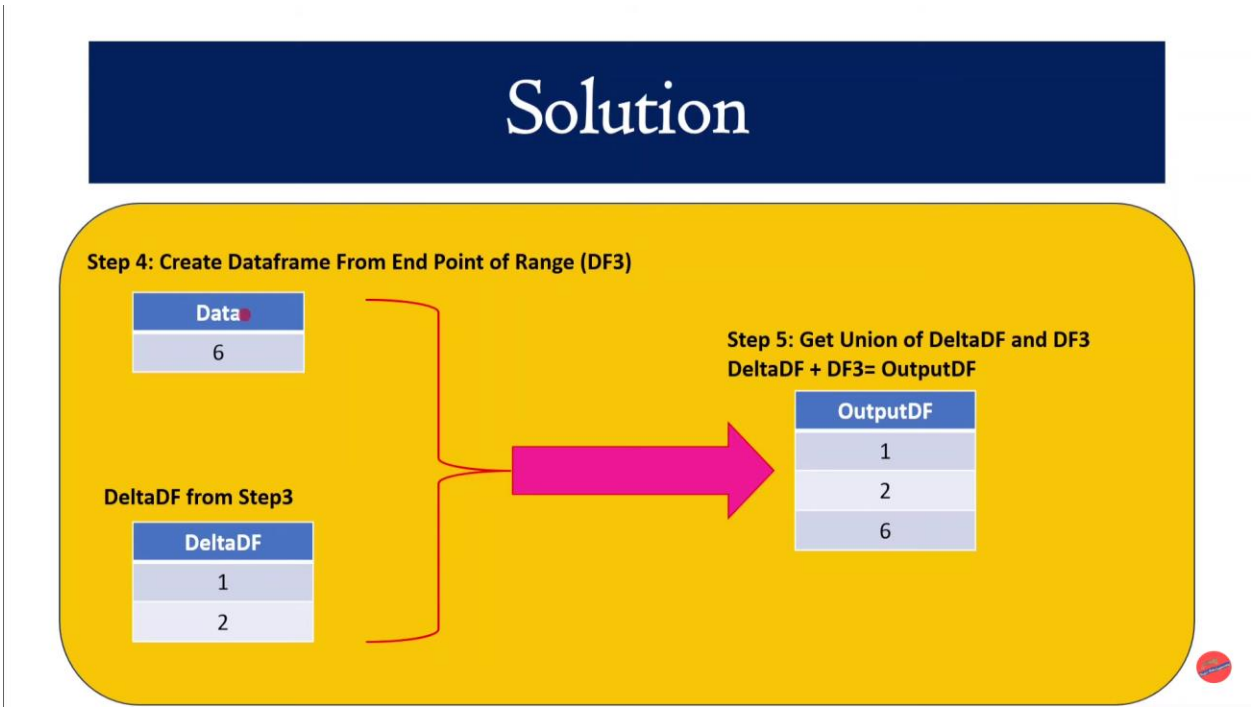
246

--

SKIP SPECIFIC RANGE OF RECORDS WHILE READING CSV FILE







WRITE A PROGRAM TO SKIP RECORDS BASED ON SPECIFIC RANGES IN THE MIDDLE OF CSV FILE , SAY FOR EXAMPLE ROWS 11 TO 20

```
#Verify Data in CSV File
%fs
head dbfs:/FileStore/tables/baby_names_id.csv
```

```
Id,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
8,2007,ZOE,NEW YORK,F,55
9,2007,ZOE,NASSAU,F,15
10,2007,ZOE,ERIE,F,6
11,2007,ZOE,SUFFOLK,F,14
12,2007,ZOE,KINGS,F,34
13,2007,ZOE,MONROE,F,9
14,2007,ZOE,QUEENS,F,26
15,2007,ZOE,ALBANY,F,5
16,2007,ZISSY,ROCKLAND,F,5
17,2007,ZISSY,KINGS,F,27
18,2007,ZION,KINGS,M,15
19,2007,ZION,BRONX,M,14
20,2007,ZEV,ROCKLAND,M,6
Command took 5.59 seconds -- by audaciousazure@gm
```

```
#Step1 – Create Data frame by Reading All Rows
fullDF = ( spark.read.format(“csv”).option(“inferSchema”,True).option(“header”,True)
.option(“sep”,”,”).load(“dbfs:/FileStore/tables/baby_names_id.csv”))
display(fullDF)
print(fullDF.count())
```

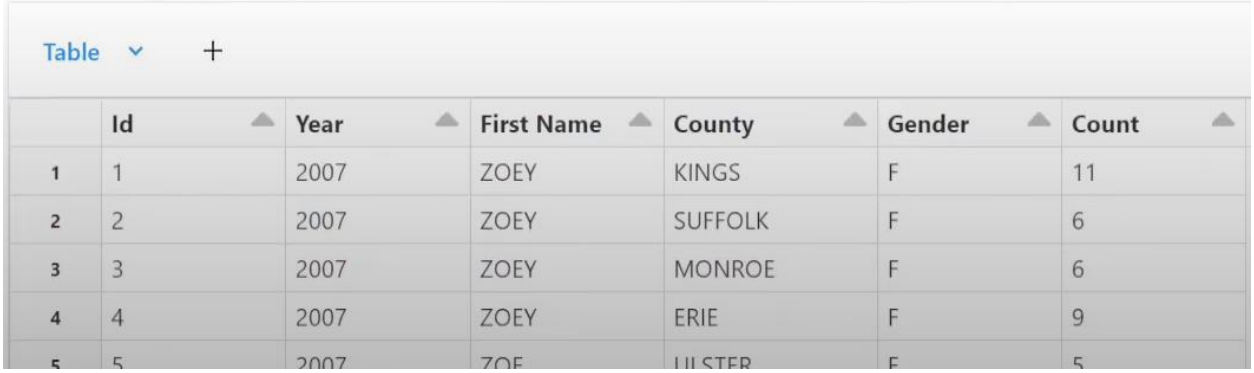


Table +

	Id	Year	First Name	County	Gender	Count
1	1	2007	ZOEY	KINGS	F	11
2	2	2007	ZOEY	SUFFOLK	F	6
3	3	2007	ZOEY	MONROE	F	6
4	4	2007	ZOEY	ERIE	F	9
5	5	2007	ZOE	ULSTER	F	5

```
250
--
Step-2 Create Dataframe by Reading Data only from Starting Row of Specific Range
skipStartDF =
(spark.read.format(“csv”).option(“inferSchema”,True).option(“header”,True)
.option(“sep”,”,”).option(“skipRows”,10)
.load(“dbfs:/FileStore/tables/baby_names_id.csv”))
display(skipStartDF)
print(skipStartDF.count())
```

Table +

	10	2007	ZOE	ERIE	F	6
1	11	2007	ZOE	SUFFOLK	F	14
2	12	2007	ZOE	KINGS	F	34
3	13	2007	ZOE	MONROE	F	9
4	14	2007	ZOE	QUEENS	F	26
5	15	2007	ZOE	ALBANY	F	5
6	16	2007	ZISSY	ROCKLAND	F	5
7	17	2007	ZISSY	KINGS	F	27

240 rows | 1.06 seconds runtime

```
240
--
Step-3: Create Data Frame by Reading Data Only from Ending Row of Specific Range
```

```
skipEndDF =
(spark.read.format("csv").option("inferSchema",True).option("header",True)
.option("sep",",").option("skipRows",20)
.load("dbfs:/FileStore/tables/baby_names_id.csv"))
display(skipEndDF)
print(skipEndDF.count())
```

Table +

	20	2007	ZEV	ROCKLAND	M	6
1	21	2007	ZEV	KINGS	M	30
2	22	2007	ZARA	QUEENS	F	10
3	23	2007	ZAIRE	KINGS	M	14
4	24	2007	ZACKARY	SUFFOLK	M	6
5	25	2007	ZACKARY	ERIE	M	5
6	26	2007	ZACHARY	NASSAU	M	41
7	27	2007	ZACHARY	NEW YORK	M	53

230 rows | 1.24 seconds runtime

```
230
```

--

```
Step4 – Find the Delta Between fullDF and skipStartDF
deltaDF = fullDF.subtract(skipStartDF)
display(deltaDF)
```

Table +

	Id	Year	First Name	County	Gender	Count
1	10	2007	ZOE	ERIE	F	6
2	2	2007	ZOEY	SUFFOLK	F	6
3	4	2007	ZOEY	ERIE	F	9
4	5	2007	ZOE	ULSTER	F	5
5	9	2007	ZOE	NASSAU	F	15
6	8	2007	ZOE	NEW YORK	F	55
7	3	2007	ZOEY	MONROE	F	6

10 rows | 1.59 seconds runtime

Step 5: Combine deltaDF and skipEndDF

```
finalDF= deltaDF.union(skipEndDF)
display(finalDF.orderBy("Id"))
```

Table +

	Id	Year	First Name	County	Gender	Count
7	7	2007	ZOE	BRONX	F	13
8	8	2007	ZOE	NEW YORK	F	55
9	9	2007	ZOE	NASSAU	F	15
10	10	2007	ZOE	ERIE	F	6
11	21	2007	ZEV	KINGS	M	30
12	22	2007	ZARA	QUEENS	F	10
13	23	2007	ZAIRE	KINGS	M	14

240 rows | 1.43 seconds runtime

UDF Combining All 5 Steps

```
1  def skipRowsRangeCSV(filePath, startPos, endPos):
2    fullDF =( spark.read.format("csv")
3                    .option("inferSchema", True)
4                    .option("header", True )
5                    .option("sep",",")
6                    .load(filePath))
7
8    skipStartDF =( spark.read.format("csv")
9                      .option("inferSchema", True)
10                     .option("header", True )
11                     .option("sep",",")
12                     .option("skipRows", startPos)
13                     .load(filePath))
14   skipEndDF =( spark.read.format("csv")
15                    .option("inferSchema", True)
16                    .option("header", True )
17                    .option("sep",",")
18                    .option("skipRows", endPos)
19                    .load(filePath))
20   deltaDF =fullDF.subtract(skipStartDF)
21   finalDF =deltaDF.union(skipEndDF)
22
23   return finalDF
```

```
outputDF = skipRowsRangeCSV(filePath="dbfs:/FileStore/tables/baby_names_id.csv", startPos=10, endPos=20)

display(outputDF.orderBy("Id"))
```

Table +

	Id	Year	First Name	County	Gender	Count
6	6	2007	ZOE	WESTCHESTER	F	24
7	7	2007	ZOE	BRONX	F	13
8	8	2007	ZOE	NEW YORK	F	55
9	9	2007	ZOE	NASSAU	F	15
10	10	2007	ZOE	ERIE	F	6
11	21	2007	ZEV	KINGS	M	30
12	22	2007	ZARA	QUEENS	F	10

240 rows | 4.22 seconds runtime

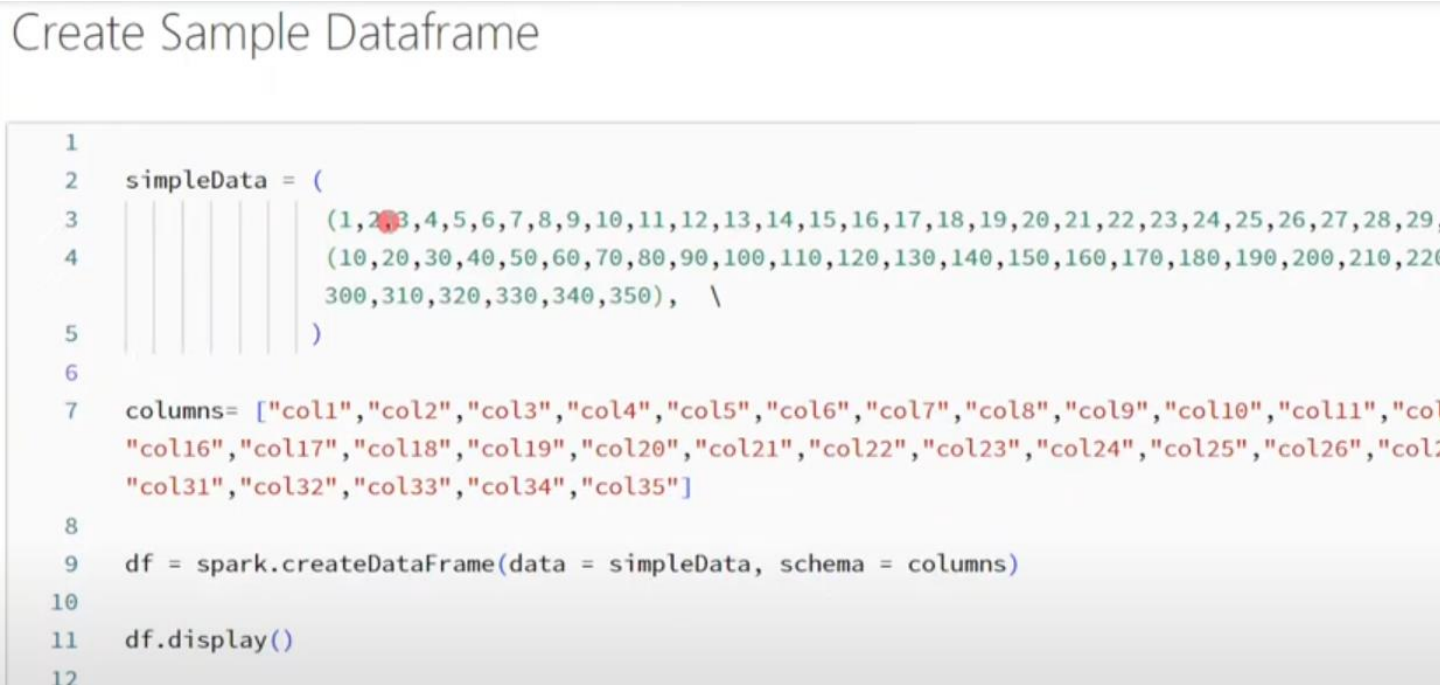
REORDERING COLUMNS IN DELTA TABLE

Reordering table columns is related to performance optimization. Reordering table columns is impacting performance of the delta engine, delta table. Delta lake is nothing but keeping data in bigdata file format parquet is called delta format. But along with data file there is one more layer which is called delta layer which is maintaining metadata and statistics of the data files. Using this metadata, we can perform ACID transactions. And also, with the help of statistics delta engine can improve the performance while performing certain data analytics because that is helping to avoid full data scanning. Delta engine does not need to scan all the data files. This process is called data skipping so that is being performed with the help of statistics. But coming to statistics in delta lake statistics is collected only for first 32 columns in a particular delta table. Let's say we are having delta table which is having 100's or 1000's of columns . Even though we are having so many number of columns, delta engine will collect statistics only for first 32 columns using which it will improve the performance through data skipping but lets say we are using one of the column more frequently in our data analytics but that is not part of first 32 columns then what happens delta engine does not have any statistics for that particular column but that is used in our data analytics predication then delta engine has to scan all the data files so that it is going to hit the performance so if we are going to use certain columns more frequently in our data analytics then we have to keep those

columns in the first 32 list. This is how reordering column is related to performance optimization.

CODE

```
#CREATE DELTA TABLE
%sql
CREATE OR REPLACE TABLE COLUMNREORDERDEMO (col1 LONG, col2 LONG, col3 LONG, col4 LONG, col5 LONG, col6 LONG, col7 LONG, col8 LONG, col9 LONG, col10 LONG, col11 LONG, col12 LONG, col13 LONG, col14 LONG, col15 LONG,col16 LONG, col17 LONG, col18 LONG, col19 LONG, col20 LONG, col21 LONG, col22 LONG, col23 LONG, col24 LONG, col25 LONG, col26 LONG, col27 LONG, col28 LONG, col29 LONG, col30 LONG, col31 LONG, col32 LONG, col33 LONG, col34 LONG, col35 LONG)
USING DELTA
LOCATION '/FileStore/tables/reordercolumn'

#CREATE SAMPLE DATA FRAME
```



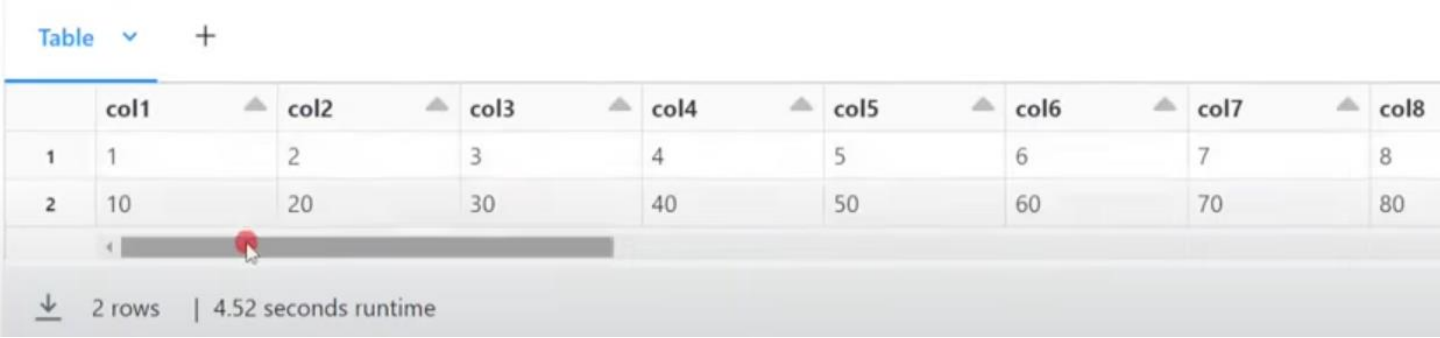


Table +

	col1	col2	col3	col4	col5	col6	col7	col8
1	1	2	3	4	5	6	7	8
2	10	20	30	40	50	60	70	80

2 rows | 4.52 seconds runtime

#WRITE SAMPLE DATAFRAME INTO DELTA TABLE

df.write.format("delta").mode("append").saveAsTable("ColumnReorderDemo")

Basically spark is working in the concept of distributed parallel processing which means here in this example we are writing 2 records so it will create 2 different data files just 2 different cores will process one record at a time so as a result it is going to create two different data files internally both will be in parquet format but in real time we are going to deal with thousands or millions of records so each and every data file contains many records, it will be thousands or millions of records. In this example, we have created 2 different data files. Each file will have only one record.

#Query table for reference

%sql

SELECT * FROM COLUMNREORDERDEMO

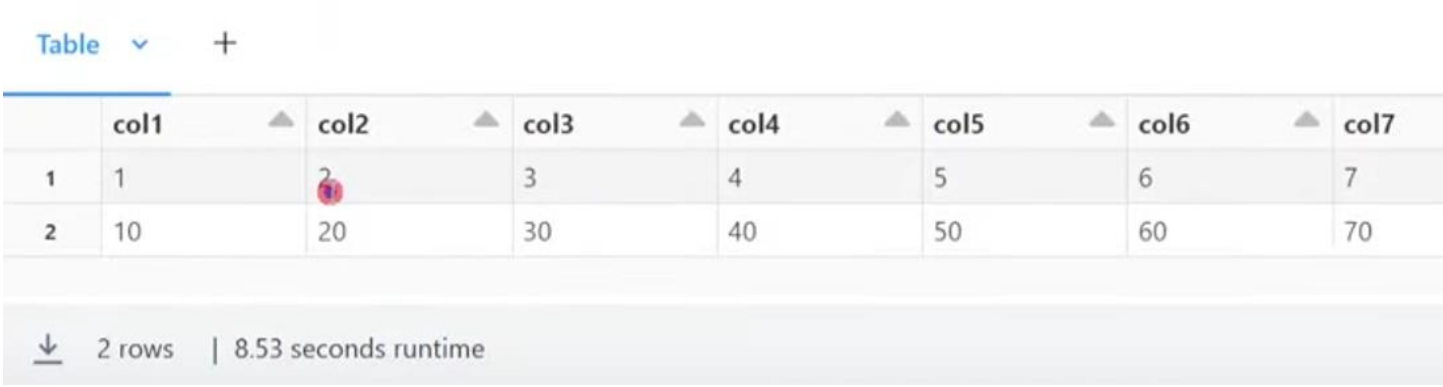


Table +

	col1	col2	col3	col4	col5	col6	col7
1	1	2	3	4	5	6	7
2	10	20	30	40	50	60	70

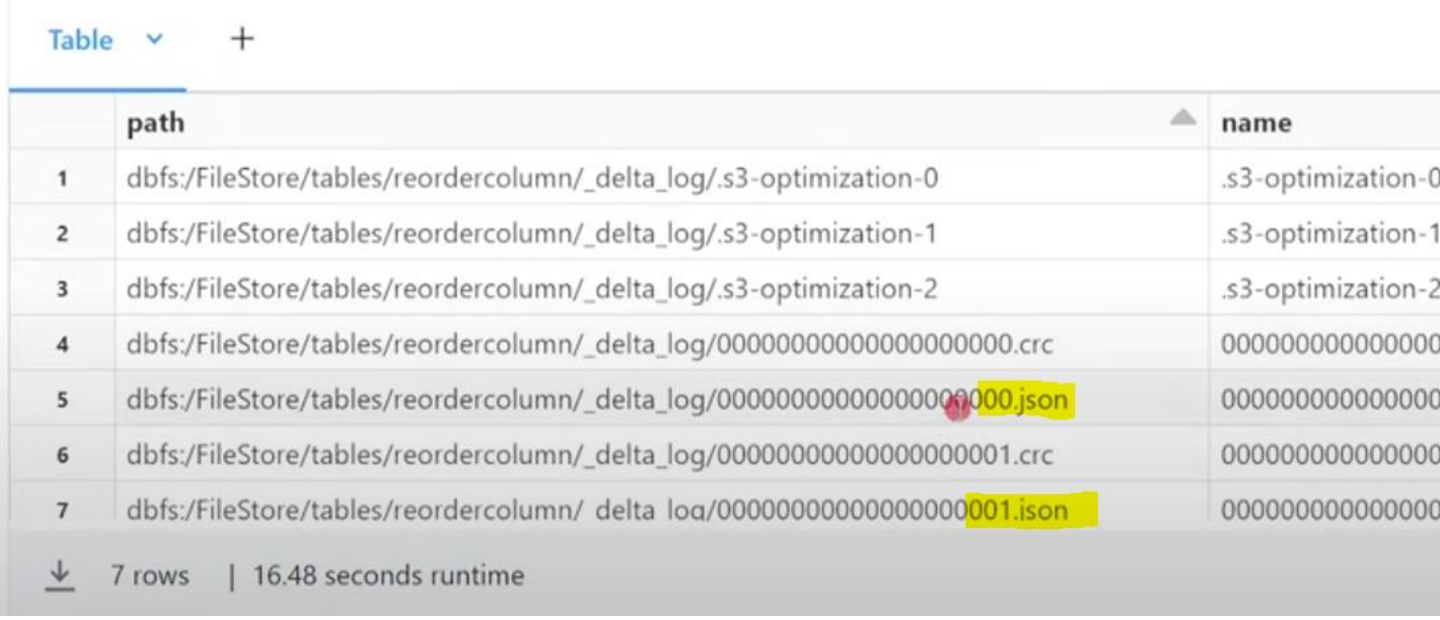
2 rows | 8.53 seconds runtime

This is the delta table. Delta Table is having 35 columns.

Whenever we are creating delta table, it is going to create delta layer for each table. **'/FileStore/tables/reordercolumn'** is the location of the table named COLUMNREORDERDEMO. Within that it is going to create _delta_log folder that is called delta layer. Within that it is going to maintain metadata and statistics. For that it will use Json log files

#VIEW DELTA LAYER

```
%fs
ls /FileStore/tables/reordercolumn/_delta_log/
```

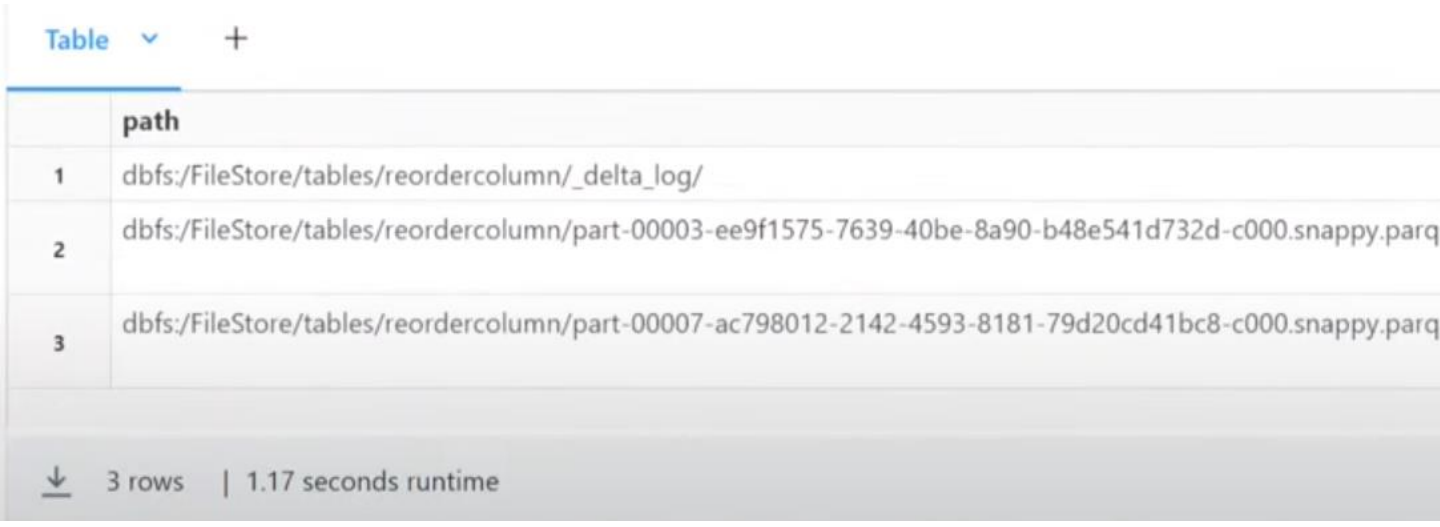


	path	name
1	dbfs:/FileStore/tables/reordercolumn/_delta_log/.s3-optimization-0	.s3-optimization-0
2	dbfs:/FileStore/tables/reordercolumn/_delta_log/.s3-optimization-1	.s3-optimization-1
3	dbfs:/FileStore/tables/reordercolumn/_delta_log/.s3-optimization-2	.s3-optimization-2
4	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000000.crc	00000000000000
5	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000000.json	00000000000000
6	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000001.crc	00000000000000
7	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000001.json	00000000000000

7 rows | 16.48 seconds runtime

In above output, we are having json log files. So here 1st version is nothing but table creation step and we are having one more json, in this step we actually inserted 2 records and if we want to see data files, we can do query like below.

```
%fs
ls /FileStore/tables/reordercolumn/
```



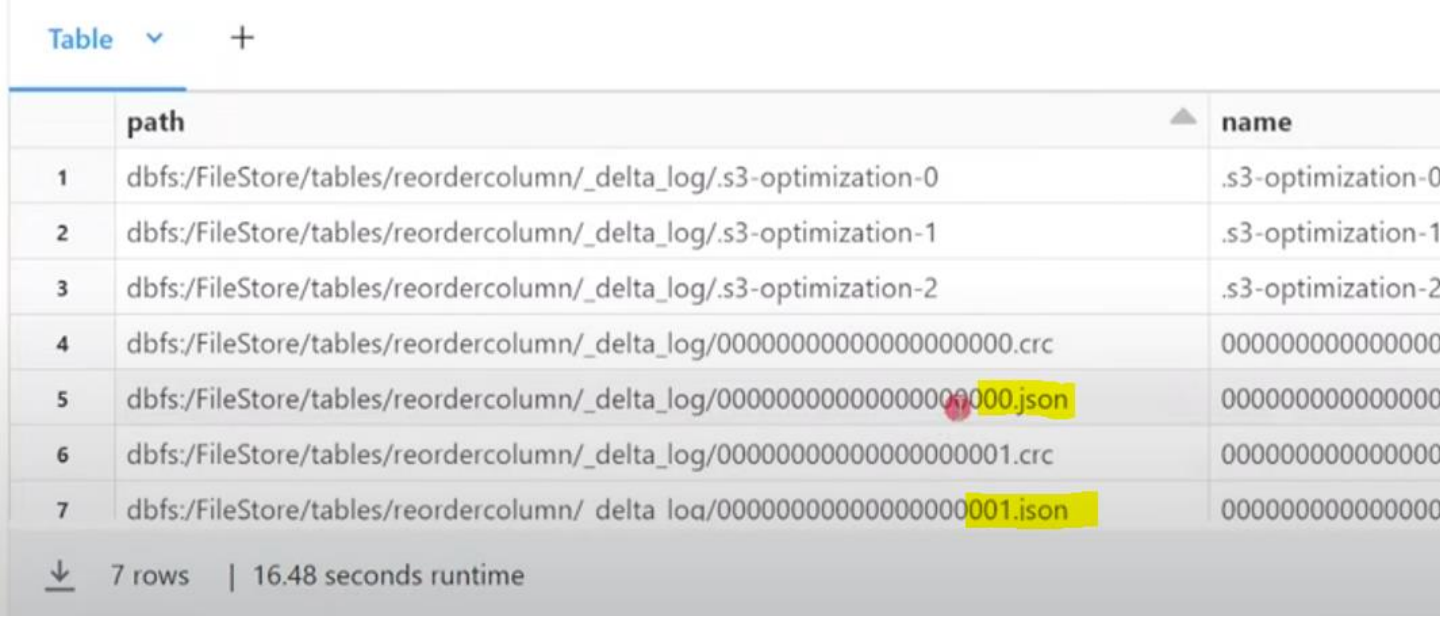
	path
1	dbfs:/FileStore/tables/reordercolumn/_delta_log/
2	dbfs:/FileStore/tables/reordercolumn/part-00003-ee9f1575-7639-40be-8a90-b48e541d732d-c000.snappy.parq
3	dbfs:/FileStore/tables/reordercolumn/part-00007-ac798012-2142-4593-8181-79d20cd41bc8-c000.snappy.parq

3 rows | 1.17 seconds runtime

We can see that it has created 2 different parquet files. So each file will contain one record.

--

```
%fs
ls /FileStore/tables/reordercolumn/_delta_log/
```



	path	name
1	dbfs:/FileStore/tables/reordercolumn/_delta_log/.s3-optimization-0	.s3-optimization-0
2	dbfs:/FileStore/tables/reordercolumn/_delta_log/.s3-optimization-1	.s3-optimization-1
3	dbfs:/FileStore/tables/reordercolumn/_delta_log/.s3-optimization-2	.s3-optimization-2
4	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000000.crc	00000000000000
5	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000000.json	00000000000000
6	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000001.crc	00000000000000
7	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000001.json	00000000000000

Table + 7 rows | 16.48 seconds runtime

And now this is the delta layer.

If we want to view data inside json file, we need to use below command

#View Json Log File

%fs

head /FileStore/tables/reordercolumn/_delta_log/00000000000000000001.json

output:


```
{"commitInfo":{"timestamp":1693843094517,"userId":"1383370256722896","userName":"audaciousazure@gmail.c
rationParameters":{"mode":"Append","partitionBy":"[]"},"notebook":{"notebookId":"4065070741378897"},"cl
zu","readVersion":0,"isolationLevel":"WriteSerializable","isBlindAppend":true,"operationMetrics":{"numF
s":"2","numOutputBytes":"18303"},"engineInfo":"Databricks-Runtime/12.2.x-scala2.12","txnId":"9c1e29f8-2
c"}}
{"add":{"path":"part-00003-ee9f1575-7639-40be-8a90-b48e541d732d-c000.snappy.parquet","partitionValues":
Time":1693843094000,"dataChange":true,"stats":"{\"numRecords\":1,\"minValues\":{\"col1\":1,\"col2\":2,\
\":5,\"col6\":6,\"col7\":7,\"col8\":8,\"col9\":9,\"col10\":10,\"col11\":11,\"col12\":12,\"col13\":13,\"
l16\":16,\"col17\":17,\"col18\":18,\"col19\":19,\"col20\":20,\"col21\":21,\"col22\":22,\"col23\":23,\"c
26\":26,\"col27\":27,\"col28\":28,\"col29\":29,\"col30\":30,\"col31\":31,\"col32\":32},\"maxValues\":{\
\":3,\"col4\":4,\"col5\":5,\"col6\":6,\"col7\":7,\"col8\":8,\"col9\":9,\"col10\":10,\"col11\":11,\"col1
\":14,\"col15\":15,\"col16\":16,\"col17\":17,\"col18\":18,\"col19\":19,\"col20\":20,\"col21\":21,\"col2
\":24,\"col25\":25,\"col26\":26,\"col27\":27,\"col28\":28,\"col29\":29,\"col30\":30,\"col31\":31,\"col3
1\":0,\"col2\":0,\"col3\":0,\"col4\":0,\"col5\":0,\"col6\":0,\"col7\":0,\"col8\":0,\"col9\":0,\"col10\"
0,\"col13\":0,\"col14\":0,\"col15\":0,\"col16\":0,\"col17\":0,\"col18\":0,\"col19\":0,\"col20\":0,\"col
\":0,\"col24\":0,\"col25\":0,\"col26\":0,\"col27\":0,\"col28\":0,\"col29\":0,\"col30\":0,\"col31\":0,\"
ION_TIME":"1693843094000000","MIN_INSERTION_TIME":"1693843094000000","MAX_INSERTION_TIME":"16938430940
E":"268435456"}}}
{"add":{"path":"part-00007-ac798012-2142-4593-8181-79d20cd41bc8-c000.snappy.parquet","partitionValues":
Time":1693843094000,"dataChange":true,"stats":"{\"numRecords\":1,\"minValues\":{\"col1\":10,\"col2\":20
ol5\":50,\"col6\":60,\"col7\":70,\"col8\":80,\"col9\":90,\"col10\":100,\"col11\":110,\"col12\":120,\"co
```

Here statistics is collected for only first 32 columns. So only 32 columns was selected in above screenshot.

Let's say in our delta table, we will do frequent data analytics based on column35 . so, if that is the case, we don't have statistics.so delta engine has to scan entire file list so it is going to hit the performance. Now my 35th col is very important column, so we have to bring it to first 32 column list so data bricks is providing 2 different approaches .

APPROACH 1:

#REORDER COLUMN 35 to FIRST

%sql

ALTER TABLE ColumnReorderDemo CHANGE COLUMN col35 FIRST

Now this col35 will be reposition to first column.

Now table got altered.

#QUERY TABLE

%sql

SELECT * FROM COLUMNREORDERDEMO

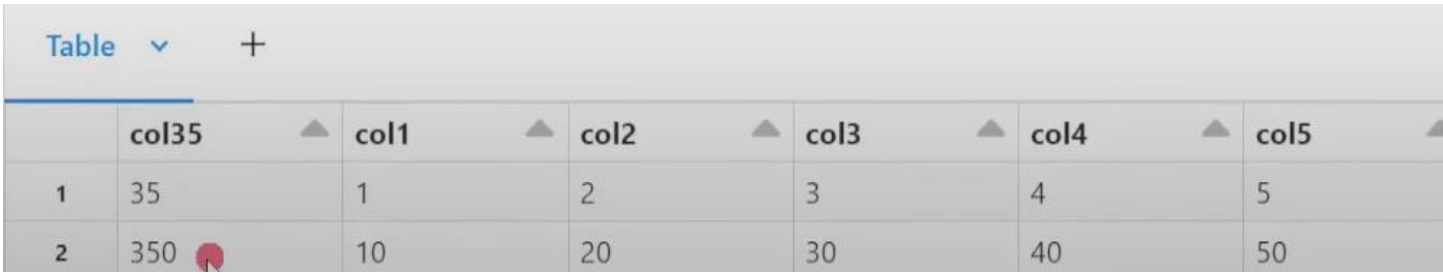


Table +

	col35	col1	col2	col3	col4	col5
1	35	1	2	3	4	5
2	350	10	20	30	40	50

--

#View Delta Layer

%fs

ls /FileStore/tables/reordercolumn/_delta_log/

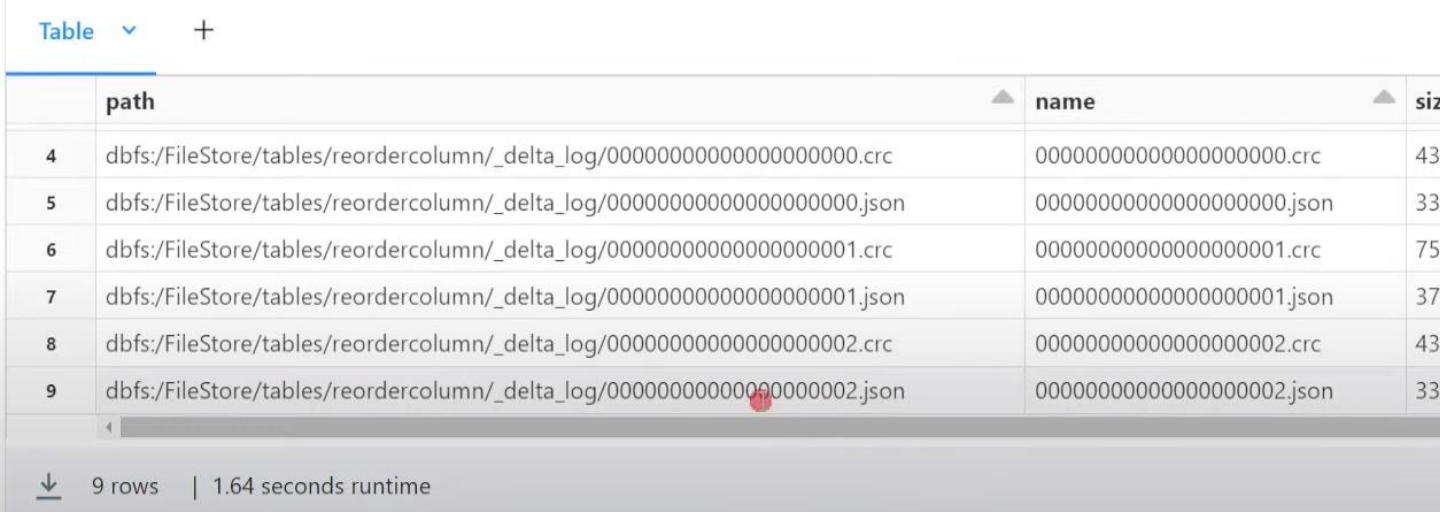


Table +

	path	name	siz
4	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000000.crc	00000000000000000000.crc	43
5	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000000.json	00000000000000000000.json	33
6	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000001.crc	00000000000000000001.crc	75
7	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000001.json	00000000000000000001.json	37
8	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000002.crc	00000000000000000002.crc	43
9	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000002.json	00000000000000000002.json	33

9 rows | 1.64 seconds runtime

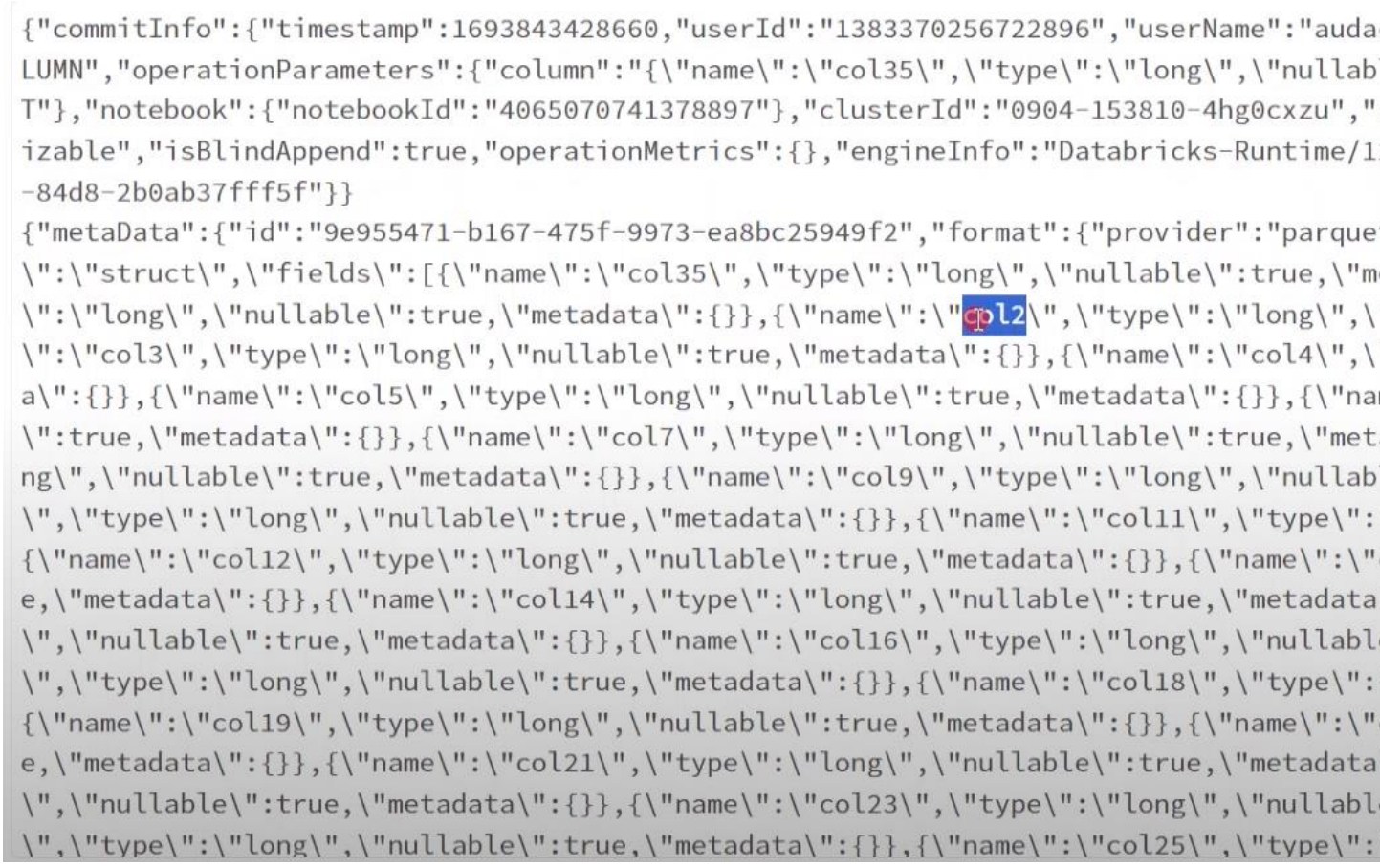
#View Json Log

%fs

head /FileStore/tables/reordercolumn/_delta_log/00000000000000000000000000002.json

Op: here its collecting metadata and statistics. Here this is table column positioning.

So in output, position: FIRST is mentioned.



{"commitInfo":{"timestamp":1693843428660,"userId":"1383370256722896","userName":"auda
LUMN","operationParameters":{"column":"{\"name\":\"col35\",\"type\":\"long\",\"nullab
T"},"notebook":{"notebookId":"4065070741378897"},"clusterId":"0904-153810-4hg0cxzu","
izable","isBlindAppend":true,"operationMetrics":{},"engineInfo":"Databricks-Runtime/1
-84d8-2b0ab37fff5f"}}
{"metaData":{"id":"9e955471-b167-475f-9973-ea8bc25949f2","format":{"provider":"parque
\":\"struct\",\"fields\":[{\"name\":\"col35\",\"type\":\"long\",\"nullable\":true,\"m
\":\"long\",\"nullable\":true,\"metadata\":{}},{\"name\":\"col2\",\"type\":\"long\",\
\":\"col3\",\"type\":\"long\",\"nullable\":true,\"metadata\":{}},{\"name\":\"col4\",\
a\":{}},{\"name\":\"col5\",\"type\":\"long\",\"nullable\":true,\"metadata\":{}},{\"na
\":true,\"metadata\":{}},{\"name\":\"col7\",\"type\":\"long\",\"nullable\":true,\"met
ng\",\"nullable\":true,\"metadata\":{}},{\"name\":\"col9\",\"type\":\"long\",\"nullab
\",\"type\":\"long\",\"nullable\":true,\"metadata\":{}},{\"name\":\"col11\",\"type\":
{\"name\":\"col12\",\"type\":\"long\",\"nullable\":true,\"metadata\":{}},{\"name\":\"
e,\"metadata\":{}},{\"name\":\"col14\",\"type\":\"long\",\"nullable\":true,\"metadata
\",\"nullable\":true,\"metadata\":{}},{\"name\":\"col16\",\"type\":\"long\",\"nullabl
\",\"type\":\"long\",\"nullable\":true,\"metadata\":{}},{\"name\":\"col18\",\"type\":
{\"name\":\"col19\",\"type\":\"long\",\"nullable\":true,\"metadata\":{}},{\"name\":\"
e,\"metadata\":{}},{\"name\":\"col21\",\"type\":\"long\",\"nullable\":true,\"metadata
\",\"nullable\":true,\"metadata\":{}},{\"name\":\"col23\",\"type\":\"long\",\"nullabl
\",\"type\":\"long\",\"nullable\":true,\"metadata\":{}},{\"name\":\"col25\",\"type\":

APPROACH 2 : REORDER COLUMN 35 – AFTER COL10

I need to place my col35 in between col10 and col11. So for that we use keyword AFTER

```
%sql
ALTER TABLE COLUMNREORDERDEMO CHANGE COLUMN COL35 AFTER COL10
```

Output : OK

```
#VIEW TABLE
%sql
SELECT * FROM COLUMNREORDERDEMO
```

Table +

	col3	col4	col5	col6	col7	col8
1	3	4	5	6	7	8
2	30	40	50	60	70	80

2 rows | 1.55 seconds runtime

Now after performing reordering of columns, we are going to insert 2 more records.

Create Another Dataframe

```
1
2  simpleData = (
3      (100,200,300,400,500,600,700,800,900,1000,1100,1200,1300,1400,1500,1600,1700,1800
       2600,2700,2800,2900,3000,3100,3200,3300,3400,3500),  \
4      (1000,2000,3000,4000,5000,6000,7000,8000,9000,10000,11000,12000,13000,14000,15000,
       22000,23000,24000,25000,26000,27000,28000,29000,30000,31000,32000,33000,34000,3500
5      )
6
7  columns= ["col1","col2","col3","col4","col5","col6","col7","col8","col9","col10","co
   "col16","col17","col18","col19","col20","col21","col22","col23","col24","col25","col
   "col31","col32","col33","col34","col35"]
8
9  df = spark.createDataFrame(data = simpleData, schema = columns)
10
11 df.display()
12
```

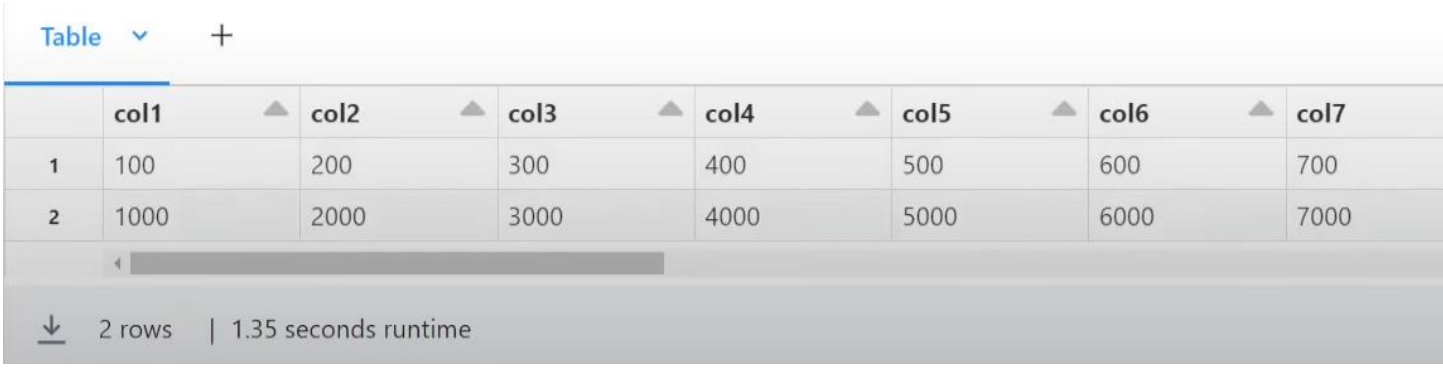


Table +

	col1	col2	col3	col4	col5	col6	col7
1	100	200	300	400	500	600	700
2	1000	2000	3000	4000	5000	6000	7000

2 rows | 1.35 seconds runtime

#LOAD DATAFRAME INTO DELTA TABLE

df.write.format("delta").mode("append").saveAsTable("ColumnReorderDemo")

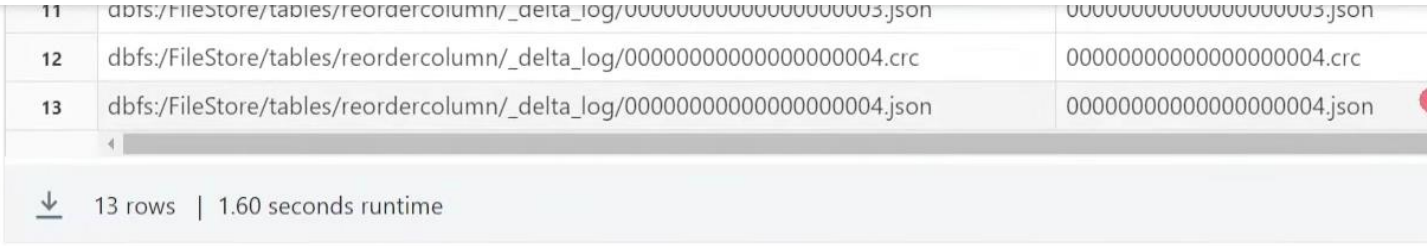
Now 2 more records will be inserted into this delta table.

#VIEW DELTA LAYER

%fs

ls /FileStore/tables/reordercolumn/_delta_log/

Now we are going to see what the log files are generated under delta layer so for that we are executing above command. we can see 2 more json file got created . one for is for 35th column repositioned after col10 and 2nd json file is for appending 2 more records.



11	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000003.json	00000000000000000003.json
12	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000004.crc	00000000000000000004.crc
13	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000004.json	00000000000000000004.json

13 rows | 1.60 seconds runtime

#VIEW JSON LOG FILE

%fs

head /FileStore/tables/reordercolumn/_delta_log/00000000000000000000000000004.json

Op: its having metadata along with statistics.

{"commitInfo":{"timestamp":1693843667365,"userId":"1383370256722896","userName":"audaciousazure@gmail.com","o
perationParameters":{"mode":"Append","partitionBy":"[]"},"notebook":{"notebookId":"4065070741378897"},"cluste
g0cxzu","readVersion":3,"isolationLevel":"WriteSerializable","isBlindAppend":true,"operationMetrics":{"numFil
ws":"2","numOutputBytes":"18305"},"engineInfo":"Databricks-Runtime/12.2.x-scala2.12","txnId":"d501c3f2-bd22-4
3"}}
{"add":{"path":"part-00003-e801e407-9fae-49f0-b426-099f632287c5-c000.snappy.parquet","partitionValues":{},"si
onTime":1693843668000,"dataChange":true,"stats":"{\"numRecords\":1,\"minValues\":{\"col1\":100,\"col2\":200,\
\":400,\"col5\":500,\"col6\":600,\"col7\":700,\"col8\":800,\"col9\":900,\"col10\":1000,\"col35\":3500,\"col11
00,\"col13\":1300,\"col14\":1400,\"col15\":1500,\"col16\":1600,\"col17\":1700,\"col18\":1800,\"col19\":1900,\
1\":2100,\"col22\":2200,\"col23\":2300,\"col24\":2400,\"col25\":2500,\"col26\":2600,\"col27\":2700,\"col28\":
0,\"col30\":3000,\"col31\":3100},\"maxValues\":{\"col1\":100,\"col2\":200,\"col3\":300,\"col4\":400,\"col5\":
ol7\":700,\"col8\":800,\"col9\":900,\"col10\":1000,\"col35\":3500,\"col11\":1100,\"col12\":1200,\"col13\":130
ol15\":1500,\"col16\":1600,\"col17\":1700,\"col18\":1800,\"col19\":1900,\"col20\":2000,\"col21\":2100,\"col22
00,\"col24\":2400,\"col25\":2500,\"col26\":2600,\"col27\":2700,\"col28\":2800,\"col29\":2900,\"col30\":3000,\
lCount\":{\"col1\":0,\"col2\":0,\"col3\":0,\"col4\":0,\"col5\":0,\"col6\":0,\"col7\":0,\"col8\":0,\"col9\":0,
\":0,\"col11\":0,\"col12\":0,\"col13\":0,\"col14\":0,\"col15\":0,\"col16\":0,\"col17\":0,\"col18\":0,\"col19\
l21\":0,\"col22\":0,\"col23\":0,\"col24\":0,\"col25\":0,\"col26\":0,\"col27\":0,\"col28\":0,\"col29\":0,\"col
0}}","tags":{"INSERTION_TIME":"1693843668000000","MIN_INSERTION_TIME":"1693843668000000","MAX_INSERTION_TIME"
0","OPTIMIZE_TARGET_SIZE":"268435456"}}}
{"add":{"path":"part-00007-20f0c8ac-1d32-4383-bda2-cb328922822c-c000.snappy.parquet","partitionValues":{},"si
onTime":1693843668000,"dataChange":true,"stats":"{\"numRecords\":1,\"minValues\":{\"col1\":1000,\"col2\":2000

In our case col35 is one of the important column which will be used in most of the data analytics predication, so that's the reason we have repositioned. Now the statistics is being collected for this column col35. So, using statistics delta engine can perform data skipping and it can improve the performance.

If we look at json log file head /FileStore/tables/reordercolumn/_delta_log/00000000000000000000000001.json.

Here in this 00000000000000000000000001.json , newly added 2 records was not updated here. it has collected statistics only for first 32 columns during that snapshot so which means it is still holding the statistics for first 32 columns not for col35. Col35 is not included. So now we have repositioned our columns so going forward whatever columns we are adding whatever records we are adding or deleting so it is going to consider statistics according to my repositioned column order but we have to consider statistics for previously added data as well. So there is one operation called recompute, we have to recompute the statistics so we will give function StatisticsCollection.recompute. so we are just recomputing the statistics for old data as well.

Recompute Old Delta Data Files

```
1  %scala
2  import com.databricks.sql.transaction.tahoe._
3  import org.apache.spark.sql.catalyst.TableIdentifier
4  import com.databricks.sql.transaction.tahoe.stats.StatisticsCollection
5
6  val tableName = "ColumnReorderDemo"
7  val deltaLog = DeltaLog.forTable(spark, TableIdentifier(tableName))
8
9  StatisticsCollection.recompute(spark, deltaLog)
```

Op:

```
import com.databricks.sql.transaction.tahoe._
import org.apache.spark.sql.catalyst.TableIdentifier
import com.databricks.sql.transaction.tahoe.stats.StatisticsCollection
tableName: String = ColumnReorderDemo
deltaLog: com.databricks.sql.transaction.tahoe.DeltaLog = com.databricks.sql.transaction.tahoe.Delta
```

#VIEW DELTA LAYER

%fs

ls /FileStore/tables/reordercolumn/_delta_log/

Its created one more json log file that is 5th version.json

Table +

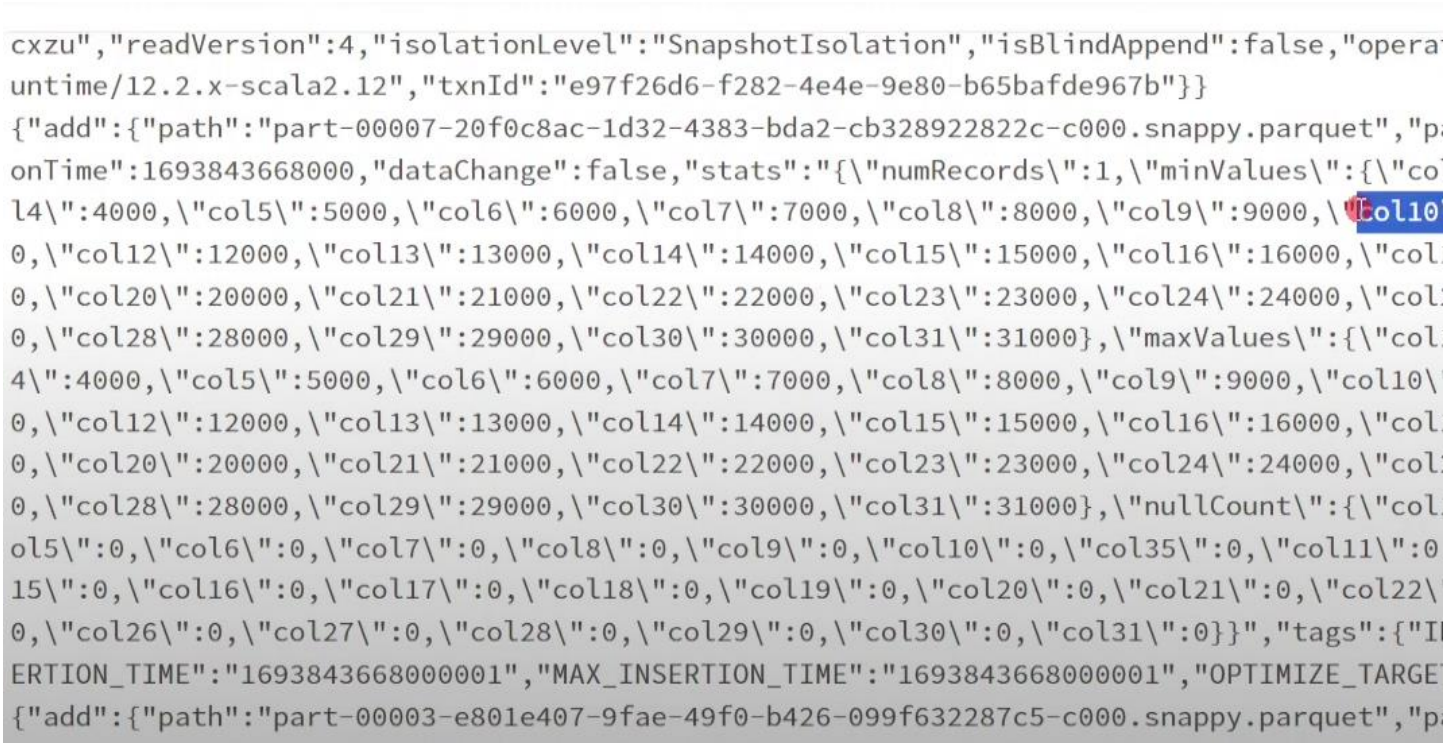
	path	name
10	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000003.crc	00000000000000
11	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000003.json	00000000000000
12	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000004.crc	00000000000000
13	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000004.json	00000000000000
14	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000005.crc	00000000000000
15	dbfs:/FileStore/tables/reordercolumn/_delta_log/00000000000000000005.json	00000000000000

#VIEW JSON LOG FILE

%fs

head /FileStore/tables/reordercolumn/_delta_log/00000000000000000005.json

In output we can see 4 output part files. We can see 4 files and statistics for each file.



Here we can see col35 is recomputed statistics for all 4 rows.

HOW TO QUERY DATAFRAME USING SPARK SQL

In spark version 3.4 onwards this new feature got introduced to query data frame using spark sql. This feature is already available in previous versions also. Let's say we have a pyspark data frame and we have to use spark sql to explore data on the data frame. Then work around is we have to convert data frame into table or tempView as a first step. And in the next cell, we have to convert that cell into SQL using magic command %sql then we have to start exploring using spark SQL. That is the process. But spark has simplified this requirement even simpler.

Create Sample Dataframe

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


# Define the schema for the DataFrame
schema = StructType([
    StructField("ProductID", IntegerType(), True),
    StructField("ProductName", StringType(), True),
    StructField("Category", StringType(), True),
    StructField("Price", DoubleType(), True),
    StructField("StockQuantity", IntegerType(), True)
])

# Create a list of rows
data = [
    (1, "Laptop", "Electronics", 999.99, 50),
    (2, "Smartphone", "Electronics", 699.99, 100),
    (3, "Headphones", "Electronics", 49.99, 200),
    (4, "Book", "Books", 19.99, 300),
    (5, "Tablet", "Electronics", 299.99, 75)
]
```

```python

 # Create a DataFrame
 productDF = spark.createDataFrame(data, schema=schema)

 # Show the DataFrame
 productDF.display()

```

▸ (3) Spark Jobs

▸ productDF: pyspark.sql.dataframe.DataFrame = [ProductID: integer, ProductName: string ... 3 m

Table +

	ProductID	ProductName	Category	Price	StockQuantity
1	1	Laptop	Electronics	999.99	50
2	2	Smartphone	Electronics	699.99	100
3	3	Headphones	Electronics	49.99	200
4	4	Book	Books	19.99	300
5	5	Tablet	Electronics	299.99	75

5 rows | 1.54 seconds runtime

#OLD APPROACH

productDF.createOrReplaceTempView("v_product") → temp view got created.

Now we want to use spark SQL to explore data on my data frame which is converted as a temp view

```
%sql
select * from v_product
```

Table +

	ProductID	ProductName	Category	Price	StockQuantity
1	1	Laptop	Electronics	999.99	50
2	2	Smartphone	Electronics	699.99	100
3	3	Headphones	Electronics	49.99	200
4	4	Book	Books	19.99	300
5	5	Tablet	Electronics	299.99	75

5 rows | 1.77 seconds runtime

--

NEW APPROACH

QUERY DATAFRAME USING SPARK SQL

```
sqlDF = spark.sql("select * from {table}", table=productDF)
```

In this select statement , we are using curly braces within that table. Whenever we are using curly braces, it means it is a parameterized spark SQL statement so table is a parameter, basically we can give any name. even we can give abc and we need to supply value for that parameter abc =productDF in select query and productDF is nothing but my dataframe so we are not converting our data frame into view or table, directly we are using data frame in spark SQL so for that we are using parameterized spark SQL concept.

```
sqlDF.display()
```

Table +

	ProductID	ProductName	Category	Price	StockQuantity
1	1	Laptop	Electronics	999.99	50
2	2	Smartphone	Electronics	699.99	100
3	3	Headphones	Electronics	49.99	200
4	4	Book	Books	19.99	300
5	5	Tablet	Electronics	299.99	75

5 rows | 1.77 seconds runtime

--

```
sqlDF = spark.sql("select {column} from {table}",
table=productDF,column=productDF["ProductName"])
```

here column also parameterized and table also parameterized. And outside we have to give parameter table = productDF that is a data frame and also column= productDF["ProductName"]

```
sqlDF.display()
```

Table +

	ProductName
1	Laptop
2	Smartphone
3	Headphones
4	Book
5	Tablet

```
#TRANSFORM DATAFRAME USING SPARK SQL
transformDF = spark.sql("select
ProductID,concat(ProductName,Category),Price*StockQuantity as Total_cost from
{table}", table = productDF)
transformDF.display()
```

Table +

	ProductID	concat(ProductName, Category)	Total_cost
1	1	LaptopElectronics	49999.5
2	2	SmartphoneElectronics	69999
3	3	HeadphonesElectronics	9998
4	4	BookBooks	5996.999999999999
5	5	TabletElectronics	22499.25

5 rows | 1.47 seconds runtime

Use of this feature is in todays bigdata world, most of the developers coming from database or data warehousing background, they are more familiar with sql than pyspark and sql is more readable when compared to pyspark and for pyspark most of the methods we have to import otherwise it will throw error so these are the couple of reasons why spark SQL is preferred for some developers over pyspark.

#TRANSFORM DATAFRAME USING PYSPARK

Transform Dataframe Using Pyspark

```
1  from pyspark.sql.functions import col,concat,expr
2  transformDFNew = productDF.select(col("ProductID"),concat(col("ProductName"), col("Category")).alias
   ("ProductNameCategory"),(expr("Price * StockQuantity")).alias("Total_cost"))
3  transformDFNew.display()
```

▸ (3) Spark Jobs

▸ transformDFNew: pyspark.sql.dataframe.DataFrame = [ProductID: integer, ProductNameCategory: string ... 1 more field]

Table +

	ProductID	ProductNameCategory	Total_cost
1	1	LaptopElectronics	49999.5
2	2	SmartphoneElectronics	69999
3	3	HeadphonesElectronics	9998
4	4	BookBooks	5996.999999999999
5	5	TabletElectronics	22499.25

5 rows | 1.02 seconds runtime

#JOIN DATAFRAMES

Create Product and Sales Dataframes

```
1
2   from pyspark.sql import Row
3   from pyspark.sql.functions import expr
4
5   # Sample data for products
6   product_data = [
7       Row(product_id=1, product_name="Laptop", unit_price=800),
8       Row(product_id=2, product_name="Smartphone", unit_price=500),
9       Row(product_id=3, product_name="Tablet", unit_price=300),
10      Row(product_id=4, product_name="Desktop", unit_price=1000),
11      Row(product_id=5, product_name="Printer", unit_price=200),
12  ]
13
```

```
# Sample data for sales
sales_data = [
    Row(sale_id=101, product_id=1, quantity=5),
    Row(sale_id=102, product_id=2, quantity=8),
    Row(sale_id=103, product_id=1, quantity=3),
    Row(sale_id=104, product_id=3, quantity=6),
    Row(sale_id=105, product_id=4, quantity=2),
    Row(sale_id=106, product_id=1, quantity=7),
]

# Create DataFrames for products and sales
product_df = spark.createDataFrame(product_data)
sales_df = spark.createDataFrame(sales_data)

product_df.display()

sales_df.display()
```

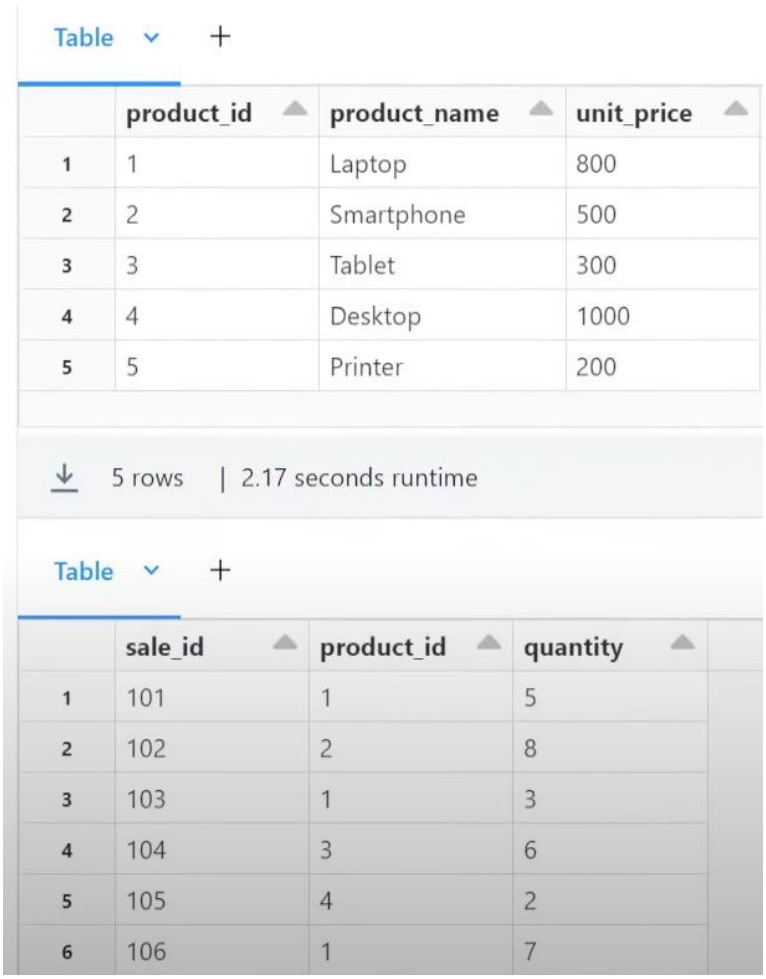


Table +

	product_id	product_name	unit_price
1	1	Laptop	800
2	2	Smartphone	500
3	3	Tablet	300
4	4	Desktop	1000
5	5	Printer	200

5 rows | 2.17 seconds runtime

Table +

	sale_id	product_id	quantity
1	101	1	5
2	102	2	8
3	103	1	3
4	104	3	6
5	105	4	2
6	106	1	7

--

JOIN DATAFRAMES USING SPARK SQL

Join Dataframes

```
1  joinDF = spark.sql("select * from {table1} a join {table2} b on a.{joiningKey} = b.{joiningKey}
   table2=sales_df, joiningKey =product_df["product_id"])
2  joinDF.display()
```

▸ (3) Spark Jobs

▸ joinDF: pyspark.sql.dataframe.DataFrame = [product_id: long, product_name: string ... 4 more fields]

Table +

	product_id	product_name	unit_price	sale_id	product_id	quantity
1	1	Laptop	800	101	1	5
2	1	Laptop	800	103	1	3
3	1	Laptop	800	106	1	7
4	2	Smartphone	500	102	2	8
5	3	Tablet	300	104	3	6
6	4	Desktop	1000	105	4	2

Using spark sql we don’t need to create temp view from dataframe, directly we can use spark sql on top of pyspark data frame.

EXCEPT FUNCTION IN SPARK SQL

This feature is available starting from data bricks run time version 9.1 .

Create Sample Dataframe

```python
from pyspark.sql.functions import rand

# Sample data for product dimension
data = [
    (1, "ProductA", "Category1", "BrandX", "Supplier1", 100, 10.99, "2023-01-01", "2
    (2, "ProductB", "Category2", "BrandY", "Supplier2", 50, 15.49, "2023-01-02", "20
    (3, "ProductC", "Category1", "BrandX", "Supplier1", 75, 8.99, "2023-01-03", "202
    (4, "ProductD", "Category3", "BrandZ", "Supplier3", 200, 25.99, "2023-01-04", "2
    (5, "ProductE", "Category2", "BrandY", "Supplier2", 60, 12.99, "2023-01-05", "20
]

# Define column names
columns = [
    "product_id", "product_name", "category", "brand", "supplier",
    "stock_quantity", "price", "start_date", "end_date", "active"
]
```

```python
# Create the DataFrame
df = spark.createDataFrame(data, columns)

# Show the resulting DataFrame
df.display()
```

▸ (3) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [product_id: long, product_name: string ... 8 more fields]

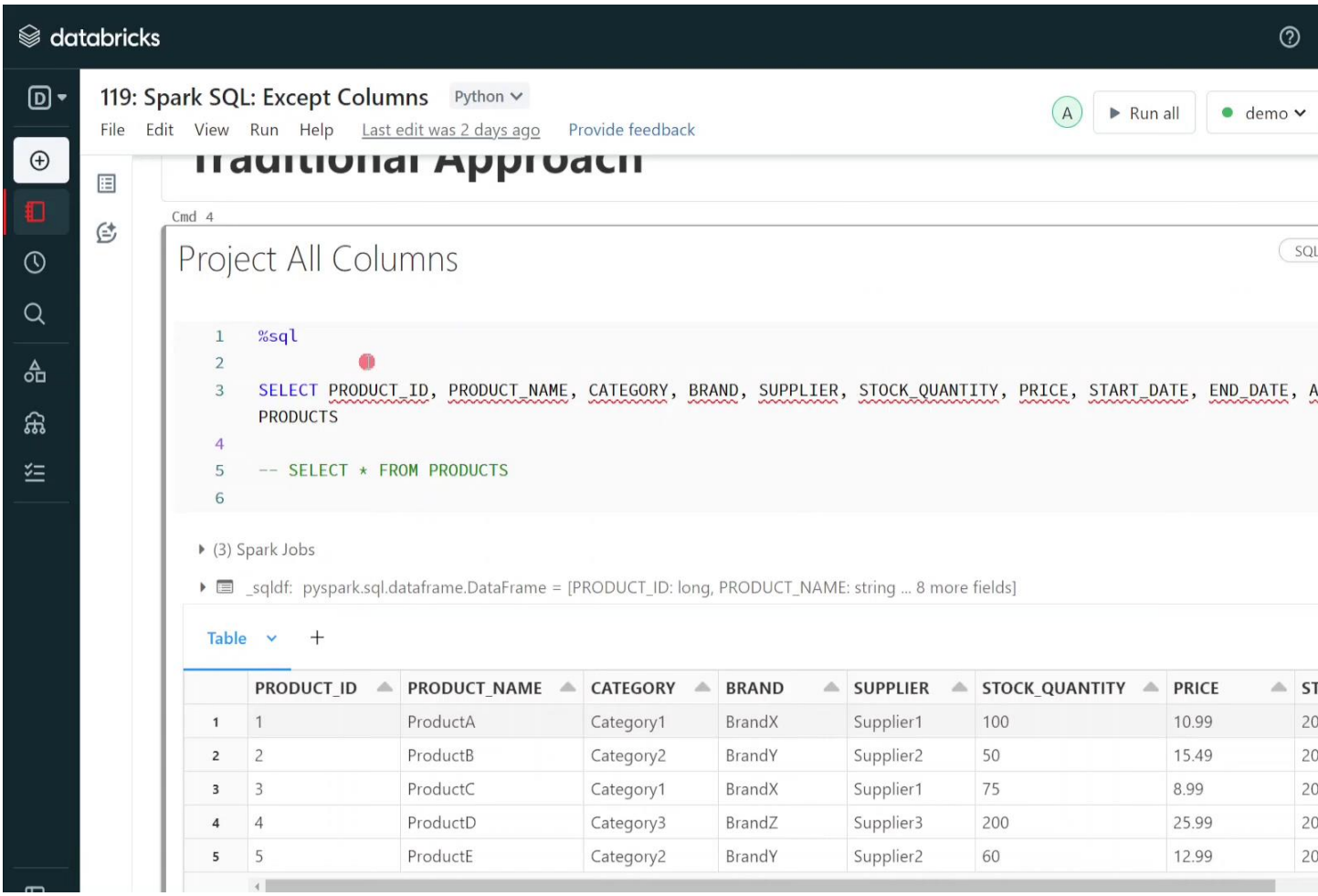
Table +

	product_id	product_name	category	brand	supplier	stock_quantity	price	sta
1	1	ProductA	Category1	BrandX	Supplier1	100	10.99	202
2	2	ProductB	Category2	BrandY	Supplier2	50	15.49	202
3	3	ProductC	Category1	BrandX	Supplier1	75	8.99	202
4	4	ProductD	Category3	BrandZ	Supplier3	200	25.99	202
5	5	ProductE	Category2	BrandY	Supplier2	60	12.99	202

```
#CONVERT DATAFRAME TO VIEW FOR SQL OPERATIONS
df.createOrReplaceTempView(“Products”)
--
```

TRADITIONAL APPROACH

#PROJECT ALL COLUMNS



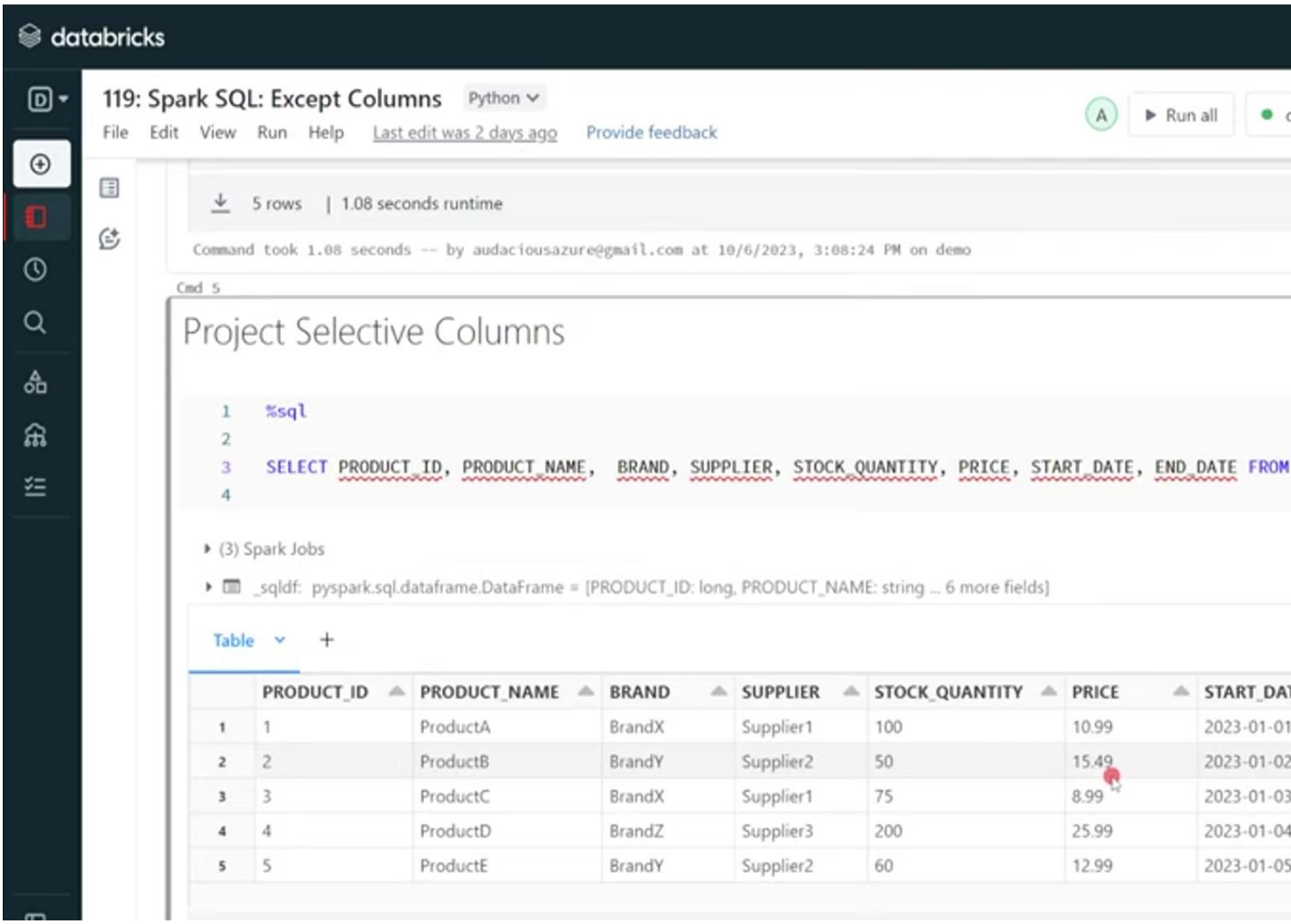
Most of the databases are supporting maximum of 254 columns per table but in later versions in today's world we will have 1000's of columns . If we have 5000's of columns for our spark table, its not good idea to write all the columns explicitly so for that there is one more option provided by traditional databases or even spark SQL that is nothing but *.

SELECT * FROM PRODUCTS

Whenever we are giving *, which means spark engine or sql engine will internally check the metadata for this table then it will retrieve all the columns and * will be replaced by all the columns. So, whenever we are giving certain query using * , internally it is going to convert that query to above select format as shown in above screenshot. So, it's always better to explicitly specify the column in terms of performance because we can avoid overhead so that SQL engine does not need to look at the metadata and retrieving all the columns and replacing the query that is not needed. But whenever we

are doing certain data analytics , its not possible to write all the columns manually so we will simply give select * .

But let's say imagine in this data analytics, we have used all 10 columns for our exploration but in real time we are not going to select all the columns . In most of the scenarios, we will select only few columns in real time. Let us assume we want to perform data analytics but we are not interested in the columns category and active, so in traditional way, we will select only the columns which we required in.



Here we are getting output except 2 columns . This is fine if we have less no of columns in my table. But whenever we are having huge no of columns in a table, let's say 5000 columns in a table, out of that I have to select 4500 columns , its not possible to write all the columns manually but still in the traditional databases we don't have any other approach we have to write all column names explicitly but coming to spark SQL interesting feature provided so that we can simplify our work faster. So, for that we are using except function.

Now if we don’t want to select CATEGORY, ACTIVE from products table and remaining all we need to select so for that we use above command.

Meaning of this query is select all columns except category and active .

```
%sql

SELECT * EXCEPT(CATEGORY, ACTIVE) FROM PRODUCTS

```

(3) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [product_id: long, product_name: string ... 6 more fields]

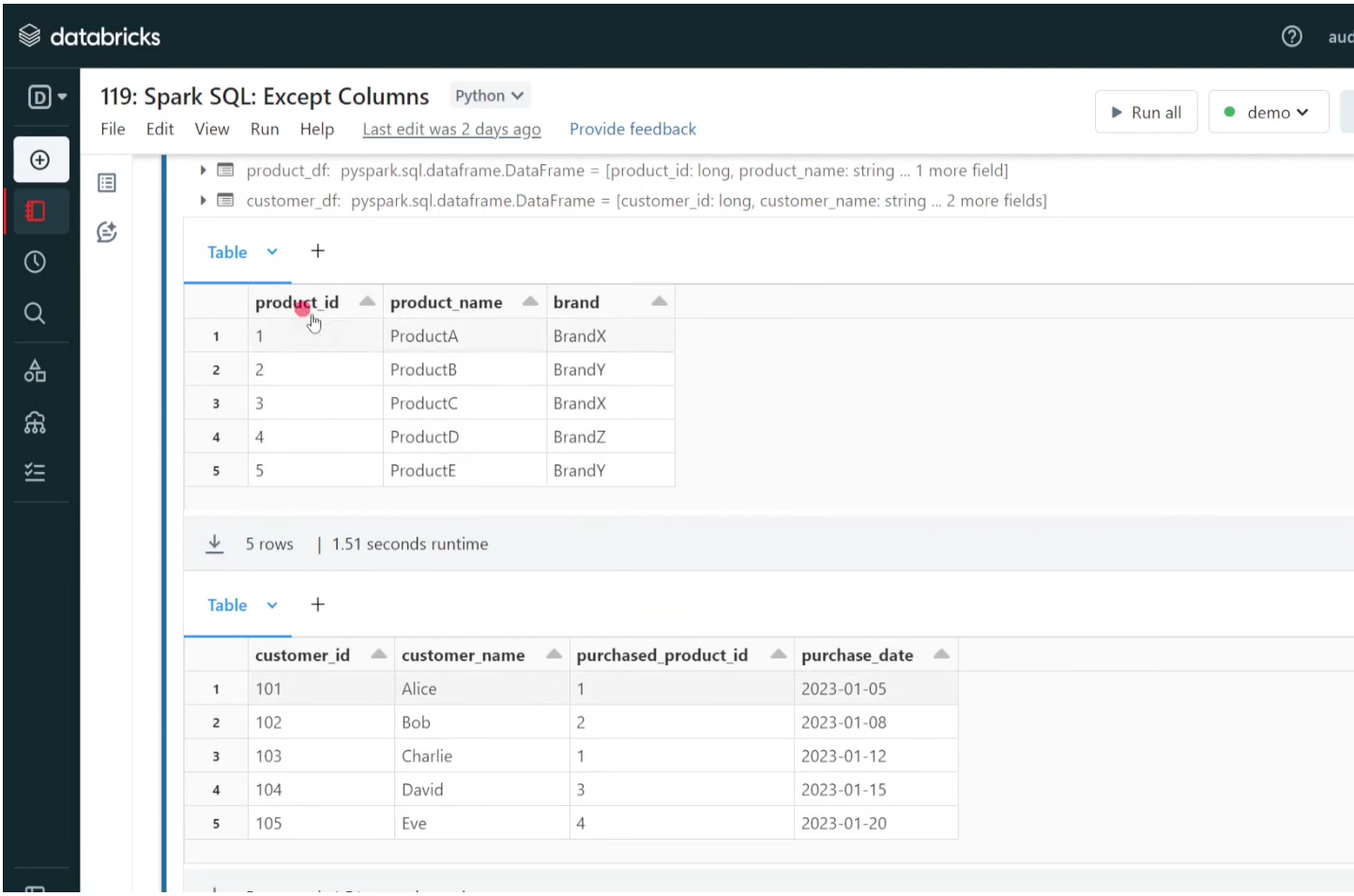
Table +

	product_id	product_name	brand	supplier	stock_quantity	price	start_date
1	1	ProductA	BrandX	Supplier1	100	10.99	2023-01-01
2	2	ProductB	BrandY	Supplier2	50	15.49	2023-01-02
3	3	ProductC	BrandX	Supplier1	75	8.99	2023-01-03
4	4	ProductD	BrandZ	Supplier3	200	25.99	2023-01-04
5	5	ProductE	BrandY	Supplier2	60	12.99	2023-01-05

This except function can be used along with join too.

```
# Sample data for the product dimension table
product_data = [
    (1, "ProductA", "BrandX"),
    (2, "ProductB", "BrandY"),
    (3, "ProductC", "BrandX"),
    (4, "ProductD", "BrandZ"),
    (5, "ProductE", "BrandY")
]

# Sample data for the customer table
customer_data = [
    (101, "Alice", 1, "2023-01-05"),
    (102, "Bob", 2, "2023-01-08"),
    (103, "Charlie", 1, "2023-01-12"),
    (104, "David", 3, "2023-01-15"),
    (105, "Eve", 4, "2023-01-20")
]
```

```python
# Define column names for both tables
product_columns = ["product_id", "product_name", "brand"]
customer_columns = ["customer_id", "customer_name", "purchased_

# Create DataFrames for both tables
product_df = spark.createDataFrame(product_data, product_columr
customer_df = spark.createDataFrame(customer_data, customer_col

product_df.display()
customer_df.display()

product_df.createOrReplaceTempView("Product")
customer_df.createOrReplaceTempView("Customer")
```



	product_id	product_name	brand
1	1	ProductA	BrandX
2	2	ProductB	BrandY
3	3	ProductC	BrandX
4	4	ProductD	BrandZ
5	5	ProductE	BrandY

	customer_id	customer_name	purchased_product_id	purchase_date
1	101	Alice	1	2023-01-05
2	102	Bob	2	2023-01-08
3	103	Charlie	1	2023-01-12
4	104	David	3	2023-01-15
5	105	Eve	4	2023-01-20

```
1  %sql
2
3  SELECT P.* EXCEPT(BRAND),C.* EXCEPT(CUSTOMER_ID) FROM PRODUCT P JOIN CUSTOMER C ON C.PURCHASED_PRODUCT
```

▸ (3) Spark Jobs

▸ _sqldf: pyspark.sql.dataframe.DataFrame = [product_id: long, product_name: string ... 3 more fields]

Table +

	product_id	product_name	customer_name	purchased_product_id	purchase_date
1	1	ProductA	Alice	1	2023-01-05
2	1	ProductA	Charlie	1	2023-01-12
3	2	ProductB	Bob	2	2023-01-08
4	3	ProductC	David	3	2023-01-15
5	4	ProductD	Eve	4	2023-01-20

In Pyspark

Cmd 12

```
1  display(df.select([col for col in df.columns if col not in {'category', 'active'}]))
```

▸ (3) Spark Jobs

Table +

	product_id	product_name	brand	supplier	stock_quantity	price	start_
1	1	ProductA	BrandX	Supplier1	100	10.99	2023-
2	2	ProductB	BrandY	Supplier2	50	15.49	2023-
3	3	ProductC	BrandX	Supplier1	75	8.99	2023-
4	4	ProductD	BrandZ	Supplier3	200	25.99	2023-
5	5	ProductE	BrandY	Supplier2	60	12.99	2023-

AUTOLOADER IN DATABRICKS

Autoloader is one of the advanced feature provided by databricks . Autoloader is specific to databricks, not spark. So EMR cluster can't support auto loader. We can also use autoloader for delta tables in databricks. AutoLoader is mainly used for incremental data loading.

INCREMENTAL DATA LOAD:

Incremental data loading is most commonly used data loading pattern in data engineering ETL pipelines. Incremental loading refers to the process of loading only new or changed data since the last load, instead of reloading the entire dataset which means whenever we are ingesting data from multiple source systems , we have to process only modified or newly added data, we should not process entire data . so, this incremental data loading pattern is one of the complex and most challenging in most of the data engineering projects. This is crucial in data processing for several reasons. First, it significantly reduces the volume of data that needs to be processed and transferred, which can save on costs and improve performance. Second, it allows for more frequent updates, which means data can be more current and valuable for decision-making.

There are multiple approaches in order to handle this incremental data load but each and every pattern is having certain problem. So, autoloader will resolve those problems.

CHARACTERISTICS OF INCREMENTAL DATA LOAD:

- Incremental data load will process only newly added or modified data files. We should not process entire data or non-modified data.
- We have to process data as soon as it has been detected. When we receive new file, immediately it should be processed.
- We should not miss any new files. We should not process previously processed file.
- It should provide good performance. Even though we are having multiple different patterns to handle incremental data load. We have to design incremental data load pipeline with good performance. Even though we are going to have huge number of files under large number of directories still there should not be any performance bottleneck.
- It should be repeatable pattern which means we should be able to schedule . then it should be able to process same thing in a repeated manner.

--

Even though we are having millions or billions of pipelines running across the world to handle incremental data load. But each and every pipeline will be suffering from some of the issue. They are getting deviated from any of above characteristics. So, in order to avoid that, we will use autoloader.

COMMON PATTERNS OF INCREMENTAL DATA LOADING:

Before autoloader, few patterns are there to handle incremental data loading.

Pattern 1: Water Mark Method:

This is traditional and most popular method which is water mark method. As per this method, when we ingest data from source system , we are going to capture maximum timestamp or one of the incremental ID column i.e.; maximum ID. We are going to capture that value and we will store it in one of the control table which is called water mark table. Then as per our ETL pipeline logic when pipeline is getting triggered next time, it will first go through control table and it will get the value of maximum time stamp or maximum ID then it will go back to the source system then it will compare whatever the new records either modified or added greater than this particular watermark timestamp or ID, it is going to pull only those value. This we call as watermark method and it is traditional approach and it is implemented in most of the traditional relational databases and data ware housing. But we have certain problem with this method. Once we have created this solution, we will create triggers based on particular interval which is schedule based trigger or we can go with event-based trigger. But coming to downside of this approach, even though there is no changes to the source system , there is no modified or new data but still at the scheduled interval, program will start executing , it will go back to the source system and it will just compare with the watermark time stamp and it will check if there is any new or modified record. When it comes to know that there is no change in the source system then it won't execute anything then the process will be terminated. So the conclusion is even though there is no changes but still our program is getting executed that is unnecessary because each execution is associated with particular cost because of computing platform , infrastructure . Because of these reasons, it is costing something. So even though there is no necessity to execute but still it is getting executed at the regular interval so this is one of the downside of this particular pattern.

Pattern 2: Checkpointing

This is most commonly used in spark structured streaming. So as per spark structured streaming it is not going to check source system when there is no change. It is going to collect offset value for the processed data and it is going to store those offset value under the checkpoint location. With the help of this checkpoint location, spark structured streaming can compare the source and target easily and it can process only the newly modified or added data but the problem with this pattern is whenever we are having huge number of files millions or billions of files across thousands or millions of folders then it is going to recognize performance bottle neck. Its not going to be efficient in order to handle huge amount of data files so that is the problem with this pattern.

Apart from that we can go with our own approaches like we can use azure logic apps, azure functions, we can go with our own approaches also. Different patterns we can

design but each and every pattern is coming up with some short coming. There is some problem with each and every pattern .

--

There are multiple patterns to handle incremental data load but each and every pattern is coming up with its own disadvantages. Best solution is we can go with autoloader. Autoloader is giving the feature of processing incremental data continuously,efficiently and automatically as soon as new files arrive. So, this is the concept of autoloader.

Autoloader is feature provided by databricks in order to handle incremental data load continuously and efficiently. There are certain other limitations like this is **currently supporting only cloud storage systems like ADLS in azure, Amazon S3 bucket, Google Cloud Storage(GCS),DBFS. These are different cloud storage system we are having in the market. Currently it is supporting only cloud storage system.**

Autoloader is supporting only certain list of file formats such as JSON, CSV,PARQUET,AVRO,ORC,TEXT and BINARYFILE formats currently supported by autoloader.

COMPONENTS OF AUTOLOADER

Autoloader is having 3 major internal components. 1st one is internal database so whenever autoloader is processing ingesting data from different source systems it is storing metadata of processed files within internal database which is RocksDB.

2nd component is in order to process incremental data it is using spark structured streaming concept. So coming to spark structured streaming it is processing incremental data but it is having certain problem with file listing. Whenever it has to handle huge amount of data then it is having certain performance bottle neck . That is actually eliminated with the help of this first component and 3rd component. But coming to processing engine, its quite similar to spark structured streaming. And 3rd component is, autoloader is using Advanced and optimized cloud native components in order to identify new files as soon as they arrive. So with the help of this processed data and also newly available data with the help of these components, autoloader is performing better than spark structured streaming.

TWO PARTS OF AUTOLOADER

When autoloader is processing the data, this is actually having 2 parts.

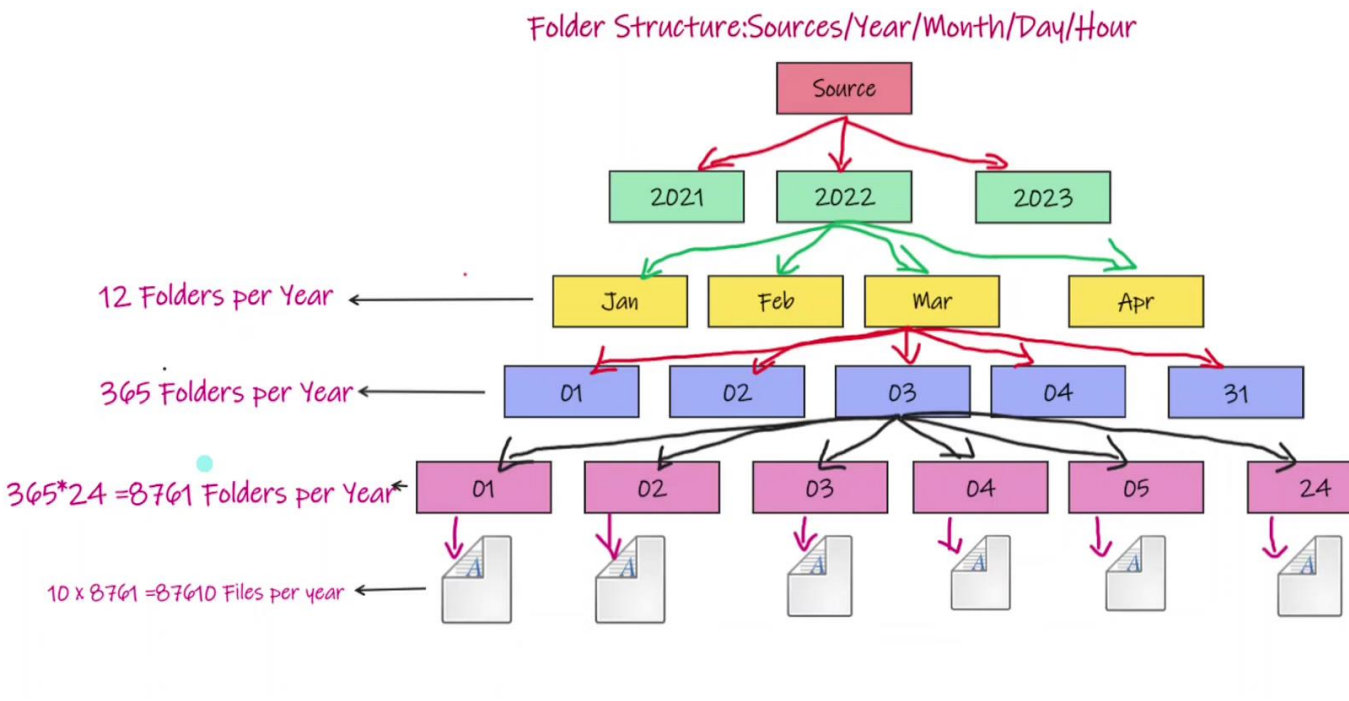
One is Cloud Files Data Reader – This is nothing but identifying the new files and how it is reading from the source system. And the 2nd one is Cloud Notification Services. This is optional. Coming to Cloud Notification Services, there are internally 2 options. One option is directory listing that is the default one. If we don't specify any option, then it is going to act as a directory listing. 2nd option is File Notification Method even we can choose that explicitly for that we have to grant certain access.

FILE DETECTION MODE

We are having 2 different file detection mode. One is directory listing mode. 2nd one is File Notification Mode. Coming to Directory Listing Mode that is a default one. We don't need to specify anything explicitly but coming to File Notification Mode we have to specify explicitly. Directory Listing Mode is more suitable in order to process the data in batch mode which means we are getting list of files for every 1hr/6 hrs/every day/every week . If we are getting real time data in a huge number of files per second or per minute then its better to go with file notification mode.

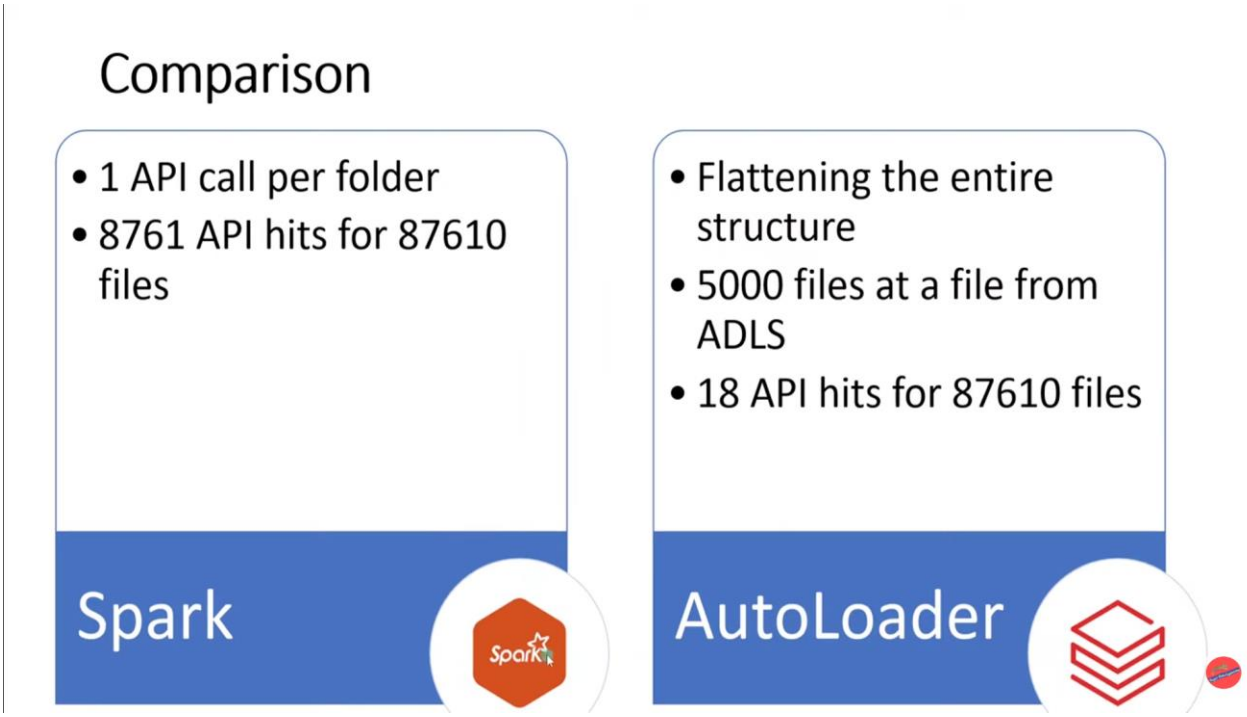
DIRECTORY LISTING MODE: This is the default mode. We don't need to specify explicitly in our programming. There is no specific permission needed in order to use directory listing mode except access to storage account in the form of mount point or we can directly access with certain key. Even for creating mount point, we are having different approaches like service principal, access key or AAD. With the help of those approaches, we need to grant access to the storage account. That is the only thing. Apart from that, we don't need to configure any specific permission. This Directory listing is more optimized and efficient directory listing program compared to spark option. Even spark structured streaming is also comparing list of files under directory and it is processing. But this is completely different from spark structured streaming.

Directory Listing



Let's say we are processing data from multiple source systems and our ADLS is divided as per above hierarchy. On top level we are having source systems and within that we are having year. Within that we are having month. Within that we are having day. Within that we are having hour. So, in real time we might have multiple source systems but for understanding we took only one source system. In one source system, we are having multiple years 2021, 2022 and 2023. Let's assume only one year we are having currently 2022. 2022 it is having multiple monthly folders internally so there will be 12 folders per year folder and each month it is divided into multiple subfolders for each. For example, for the month of jan, we will have 31 subfolders. For the month of feb , we will have 28/29 subfolders. For the month of march, we will have 31 subfolders. And, coming to each day,again it is sub-divided into 24 subfolders one for each hour. So we will get 365*24=8761 folders per year. Now each hour we will assume we will have 10 files under each folder.so we will be having 87610 files per year. This is just simple example but in real time we will have complex hierarchial structure.

In order to process these files lets assume we are going to use spark structured streaming in one case and autoloader in another use case.



So as per spark structured streaming, its having the concept of 1 API call per folder. In order to process newly added or modified files, it will send one API call per folder. In this case we are having 8761 folders which means in order to process 8761 folders, it will send 8761 times of API calls and in order to process 87610 files it needs 8761 API calls. But coming to autoloader, it is following different powerful algorithm . As per autoloader it is going to consider entire folder in flatten structure so it is basically flattening entire structure and as per ADLS feature it is supporting 5000 files for single API call which means in order to process 87610 files it will just send only 18 API calls. And for each and every storage no of files will different. For example, for Amazon S3 bucket limit will be something different. For GCS it will be different. But for ADLS 5000 is the limit. In order to process 87610 files, it will just take only 18API hits. But for same process, spark structured streaming will take 8761 API hits. Autoloader will quickly finish the data ingestion compared to spark structured streaming. This is the power of autoloader in directory listing mode.

FILE NOTIFICATION MODE

This is mostly suitable whenever we are processing millions of files under huge number of directories and also data is coming in real time for every minute or every second. We are receiving many files so for those scenarios file notification mode is more powerful than directory listing mode. For file notification mode, we have to grant certain access like for ADLS account we have to grant contributor access. For storage queue also contributor access. And also, for event subscription we have to grant

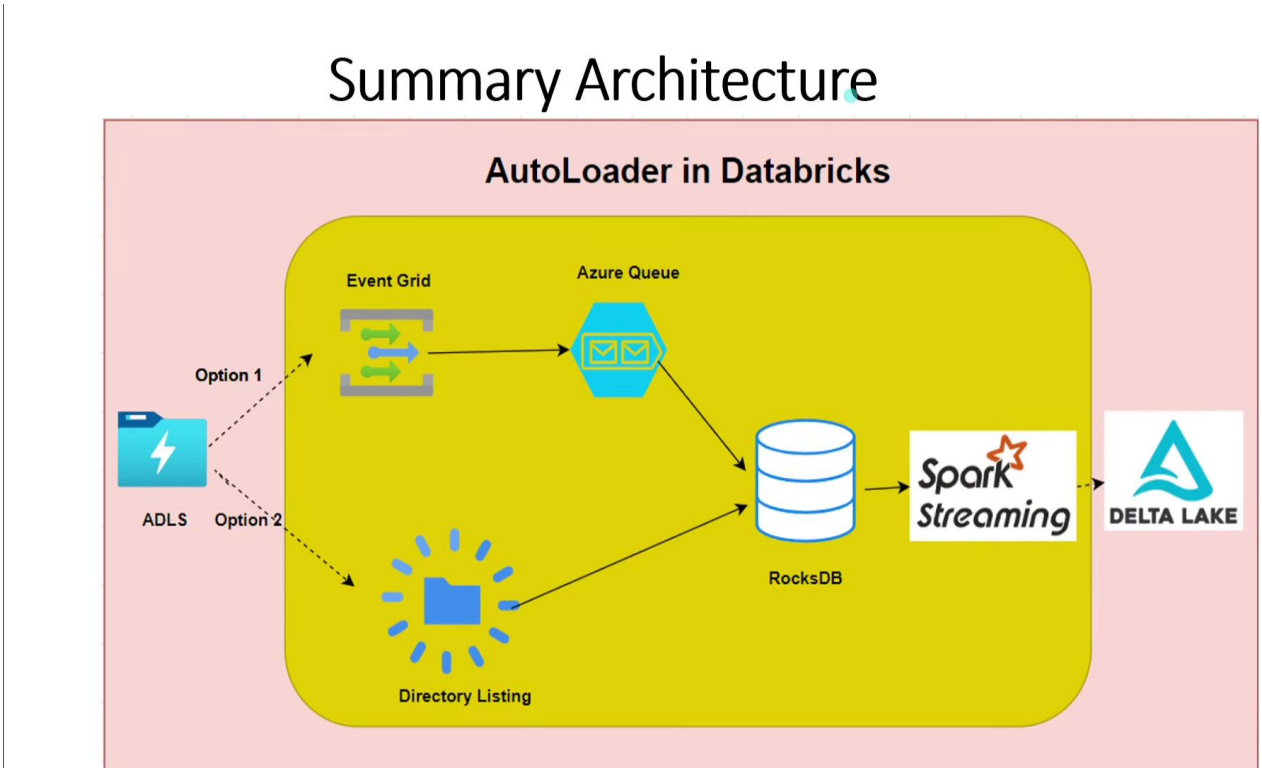
contributor access for these services. Once it is done then autoloader will automatically set up the notification and queue services for us then it will start using that , we don't need to manually set up anything but in the program in order to grant access to Q-storage , we have to create a service principal and we have to give certain configuration details through our code.

DIFFERENT PARAMETERS TO PASS FOR FILE DETECTION MODE

Subscription ID, Resource Group , Tenant ID, Client ID, Client Secret.

Client ID and Client Secret is nothing but service principal credentials.

ARCHITECTURE OF AUTOLOADER IN DATABRICKS



We are having ADLS , we have to process the data into one of the medallion architecture in our delta lake so for that we can go with spark structured streaming but that is not powerful so that's the reason we want to go with autoloader. So, coming to autoloader we have 2 different options. We can use event grid and Azure Queue(This is nothing but File Notification Mode) so for that these services will be created by autoloader itself automatically but we have to configure certain things and we need to grant certain access. So for this method, autoloader will consume metadata from ADLS and it will just store processed metadata within RocksDB. With the help of RocksDB and source system , spark structured streaming will be able to process the data efficiently into our medallion architecture. This is one approach.

Second approach is This is default one option 2. In this option, we don't need to explicitly mention the type. By default, it will be using directory listing . So in the directory listing, basically it is going to list down all the files under the different folders and it is going to maintain processed data within the RocksDB and with the help of this RocksDB and directory listing, spark structured streaming will be able to pull the unprocessed or modified newly added data efficiently from the source system and that can be processed and pushed into medallion architecture within the delta lake. This is the summary architecture of autoloader.

CODE:

First step is we have to grant access to the storage account to autoloader so for that we can use spark configuration settings.

```
1  # ADLS
2  adlsAccount = dbutils.secrets.get(scope="<key vault secr
3  adlsKey = dbutils.secrets.get(scope="<key vault secrets>
4
5  spark.conf.set("fs.azure.account.key." + adlsAccount + "
6  spark.conf.set("fs.azure.account.key." + adlsAccount + "
7
```

After giving spark config settings, autoloader will be able to ingest data from the source system.

Once this is done, basic configuration .

Whenever we have to use autoloader then we have to use format cloudFiles . Whenever we are using cloudFiles which means it is going to use autoloader internally. One of the mandatory parameter is option, we have to specify which file format we are going to process . In ADLS, we might have CSV file, JSON file or parquet file. So, what kind of file format we have to process in this ingestion so for that we will use option. Within option, we have to tell cloudFiles.format as Json. So many options available in autoloader feature.

```
df = (spark.readStream \
                .format("cloudFiles")
                .option("cloudFiles.format", "json")
                .load(location)
```

--

While creating data frame, we used to specify the option to define the schema explicitly with the help of struct type and struct field and we used to mention that and that is recommended for good performance. Incase if we go with inferSchema then unnecessarily data bricks engine will scan the datafiles once in order to derive the schema but that is very costly operation in regular spark processing but coming to autoloader even that is not very costly because autoloader efficiently samples the data to infer the schema so even we can go with inferSchema option. For that we have to go with option(cloudFiles.inferColumnTypes as true) By default it will be false but if we want to go with inferSchema option we will give cloudFiles.inferColumnTypes as true . Along with this option we should give schemaLocation. Once we have created inferSchema where we have to store that information for that we have to give schema Location of our choice, it could be ADLS or DBFS.

Schema Option

```
df = (spark.readStream \
                .format("cloudFiles")
                .option("cloudFiles.format", "json")
                .schema(user_schema)  # user Defined Schema
                .load(location))

# Usually Infer Schema is very expensive and not recommended, but not with Auto Loader
infer the schema and stores it under cloudFiles.schemaLocation in our storage.
# Another Option cloudFiles.inferColumnTypes determines the proper data type from our

df = (spark.readStream
                .format("cloudFiles")
                .option("cloudFiles.format", "json")
                .option("cloudFiles.schemaLocation", '/mnt/data/inferred_schema')
                .option("cloudFiles.inferColumnTypes", "true")
                .load(location))
```

These are the different options we can explicitly define the schema and use or we can ask autoloader engine to infer the schema .

Apart from that we can specify schema hints as well. Whenever we are telling autoloader to infer the column type. Let’s say we are having one column emp_id which is having all the numerical values but databricks autoloader might infer that as integer . Let’s

assume that column should be bigint instead of regular integer then what we can do is only for those particular columns we can give hints as well.

Schema Hints

```
# We can enforce a part of our schema. This can easily be done with Schema
# In this example, I can make sure that emp_id is read as bigint and not i

df = (spark.readStream
        .format("cloudFiles")
        .option("cloudFiles.format", "json")
        .option("cloudFiles.schemaLocation", '/mnt/data/inferred_s
        .option("cloudFiles.inferColumnTypes", "true")
        .option("cloudFiles.schemaHints", "emp_id bigint")
        .load(location))
```

HOW TO USE NOTIFICATION MODE?

For file notification mode we have to give below syntax option(“cloudFiles.useNotifications” as ‘true’). default it will be false. it will use directory listing mode. So, whenever we have to use file notification mode, then we have to use option(“cloudFiles.useNotifications” as ‘true’).

Use Notifications

```
df = (spark.readStream
        .format("cloudFiles")
        .option("cloudFiles.format", "json")
        .option("cloudFiles.useNotifications", 'true')
        .schema(userDefinedSchema)
        .load(location))
```

Whenever we have to use file notification mode, we have to grant access in azure portal for ADLS account, Azure Queue and Event Grid. Once that is done and also created service principal then even we can pass these below parameters along with autoloader.

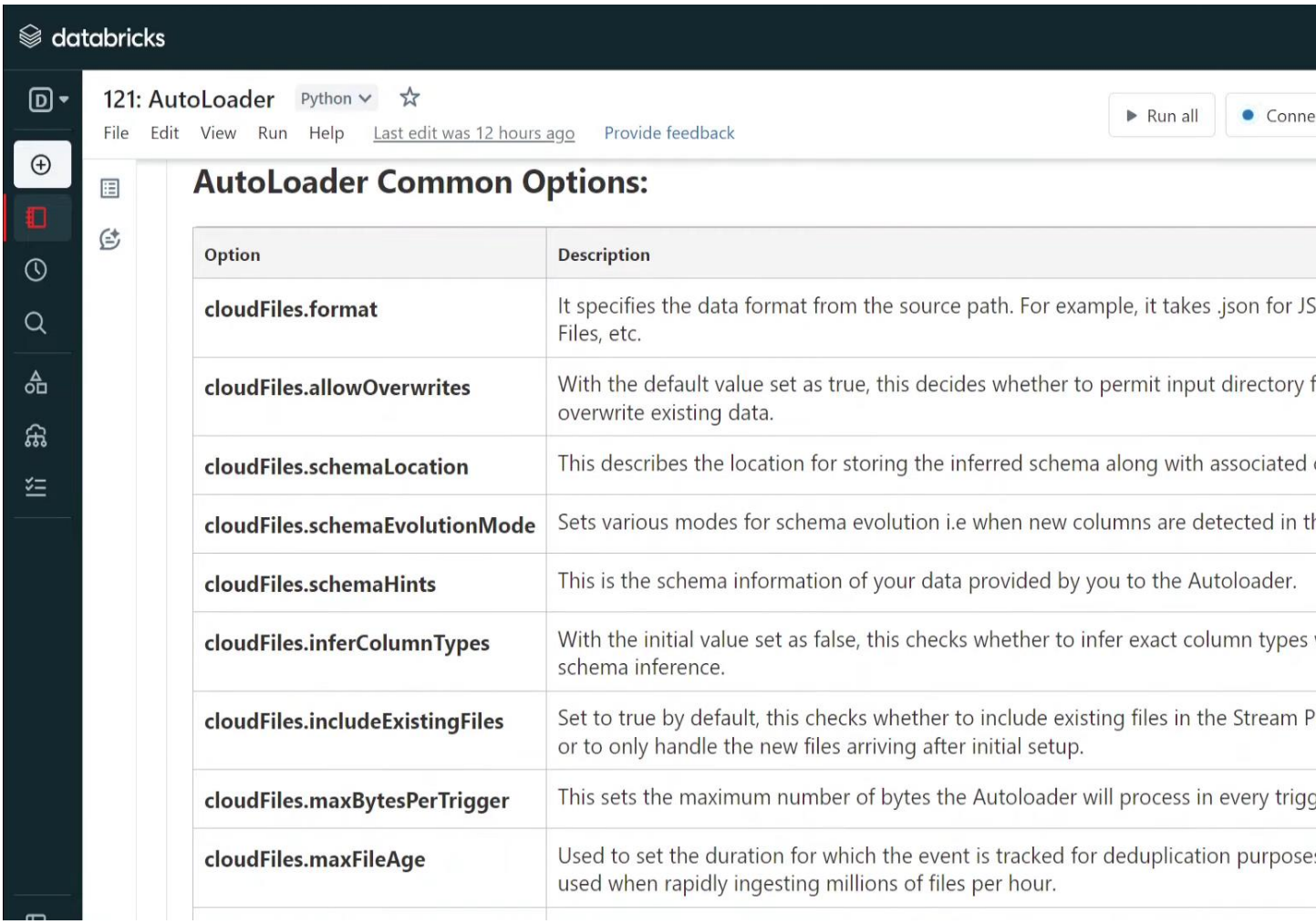
We created variable cloudFilesConf . within that we are defining all the parameters along with its value with the help of key value pair. It's a kind of python dictionary. One side we are having all the parameters and other side we are keeping all the values.

Define Queue Storage Options

```
cloudFilesConf = {
  "cloudFiles.subscriptionId": dbutils.secrets.get(scope="key-vault-secrets", key="sp-subscripti
  "cloudFiles.connectionString": queue_sas,
  "cloudFiles.format": "avro",
  "cloudFiles.tenantId": dbutils.secrets.get(scope="key-vault-secrets", key="sp-tenantId"),
  "cloudFiles.clientId": dbutils.secrets.get(scope="key-vault-secrets", key="sp-clientId"),
  "cloudFiles.clientSecret": dbutils.secrets.get(scope="key-vault-secrets", key="sp-clientKey"),
  "cloudFiles.resourceGroup": dbutils.secrets.get(scope="key-vault-secrets", key="sp-rgName"),
  "cloudFiles.useNotifications": "true",
  "cloudFiles.includeExistingFiles": "true",
  "cloudFiles.validateOptions": "true",
}
```

Then while reading the data, we can pass all above values in one go as shown below. Here we have defined options and we have passed cloudFilesConf variable and this variable is key value pair so that's the reason we need to give ** . in case if it is array, we need to give * . we need to give options because cloudFilesConf variable has multiple options.

```

autoloader_df = (spark.readStream.format("cloudFiles")
                 .options(**cloudFilesConf)
                 .load(f"wasbs://landing@{adls_account}.blob.core.windows.net/cur
                 12345/event_type={table}/")
                 )
```



cloudFiles.format → it describes which type of file we have to use for our ingestion that we can specify.

cloudFiles.allowOverwrites → Incase there is some changes to the existing record . Let's say in our existing value for one of the employee, salary is 5000. But in the source its 6000. There is some change. Then we have to overwrite the existing record . so, this is a default property. In case if we don't want , we can change as well.

cloudFiles.schemaLocation → Incase if we are going with inferSchema where we have to store that schema information so for that we can define schema location.

cloudFiles.schemaEvolutionMode → Autoloader is efficiently handling schema evolution. With this property we can play around this schema evolution. Basically, it is giving different schema evolution modes to handle schema evolution.

Schema Evolution Mode

SchemaEvolutionMode:

Option	Description
addNewColumns (default)	Stream fails. New columns are added to the schema. Existing co
rescue	Schema is never evolved and stream does not fail due to schem rescued data column.
failOnNewColumns	Stream fails. Stream does not restart unless the provided schem removed.
none	Does not evolve the schema, new columns are ignored, and dat option is set. Stream does not fail due to schema changes.

By default, it is going to use addNewColumns. Lets say in our target table we are having 5 columns. In the source we are getting 6 columns. So, one column we are getting then what happens is our streaming pipeline will fail then the new column will be added after that streaming process will be restarted. This is the default mode. Coming to rescue mode it is not going to add new column to the target but instead of that it is going to create one column which is called underscore rescue data . within that it is going to document what are the new columns we have received and at the same time it is not going to fail the streaming process. Coming to failOnNewColumns mode, let's say we are processing some sensitive data , we are not expecting any new column to arrive but in case something is coming that is unexpected then it should fail. So, for that scenarios we can go with failOnNewColumns and streaming pipeline will fail and it will not restart automatically . we have to fix the schema issue then only it will restart. Coming to the last one none, just totally ignore. Whenever we are getting some new schema which means it is turning off schema evaluation feature. These are different options available for schema evolution mode.

cloudFiles.schemaHints → In case for a particular column we want to specify particular datatype then we can go with that.

cloudFiles.inferColumnTypes → incase if we don't want to go with user defined schema then we can go with this one.

cloudFiles.includeExistingFiles → whenever we are setting up this entire autoloader process for first time and we are pointing to some data folder but there are already files available so the previously available file we have to consider or not, that property is handled with the help of this option.

cloudFiles.maxBytesPerTrigger → whenever we are processing incremental data, basically it is triggered may be could be once or micro batch or it could be continuous. But for each trigger, how much data it can fetch at a time, that property is defined with the help of this option.

cloudFiles.maxFileAge → This is one of the important parameter in order to handle the duplicate files. Let's say we are going to set 1 hour is the maximum age for all the files.so within one hour, we are going to get 2 different data files with same name then it will considered as duplicate. But after 1 hour let's say last one hour I got one file and after completing next hour again I am getting the same file then it will be treated as a new file. so, in order to define the duplicate file, this property is very important.

cloudFiles.resourceTags → This is helping to identify right resources. This is not very commonly used.

cloudFiles.validateOptions → This checks if the autoloader options discussed above are having valid value or not. For example, in our code for one of the option we might have given wrong value. If we have given wrong value that will be ignored and it will be proceeded. So whatever options we have specified in our autoloader ingestion everything will be validated. Incase certain value is incorrect then it is going to report. This is one of the important parameter.

cloudFiles.backfillInterval → For what duration we have to consider the backfilling. That is defined with the help of this parameter.

cloudFiles.partitionColumns → This is actually using hive style partition.so whenever we are ingesting data from the source system, incase the source system is partitioned based on the folder. Usually this is following this style country = UK, or city = certain city. Incase source system is following hive style partition then that can be used for autoloader ingestion.

--

Wildcard Options:

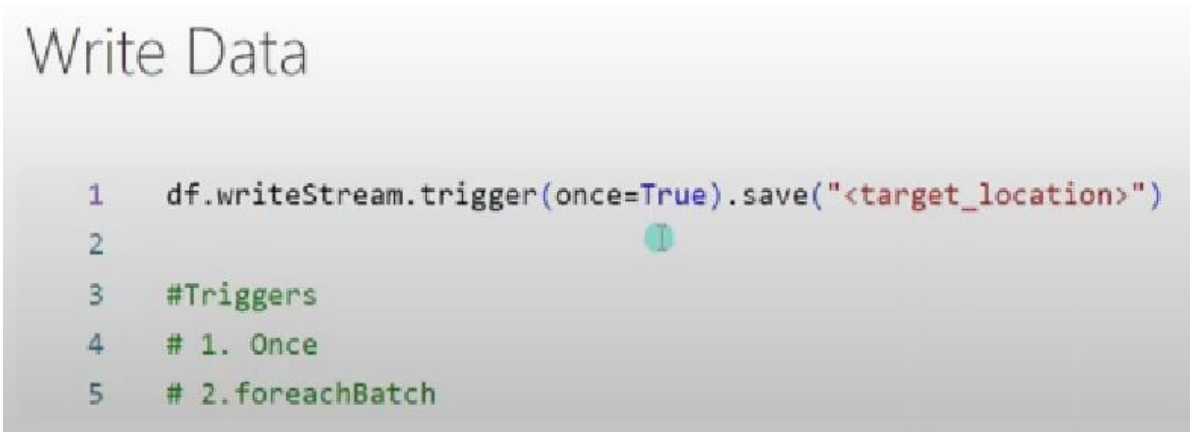
Option	Description
?	Matches any single character
*	Matches zero or more characters
[abc]	Matches a single character from character set {a,b,c}.
[a-z]	Matches a single character from the character range {a...z}.
[^a]	Matches a single character that is not from character set or range {a}. Note that the ^ charac right of the opening bracket.
{ab,cd}	Matches a string from the string set {ab, cd}.
{ab,c{de, fh}}	Matches a string from the string set {ab, cde, cfh}.

Apart from that while processing data through autoloader, we can consider wild card options also.

Read Data Using WildCard

```
1  df = spark.readStream.format("cloudFiles") \
2    .option("cloudFiles.format", <format>) \
3    .schema(schema) \
4    .load("root_folder/*/files")
```

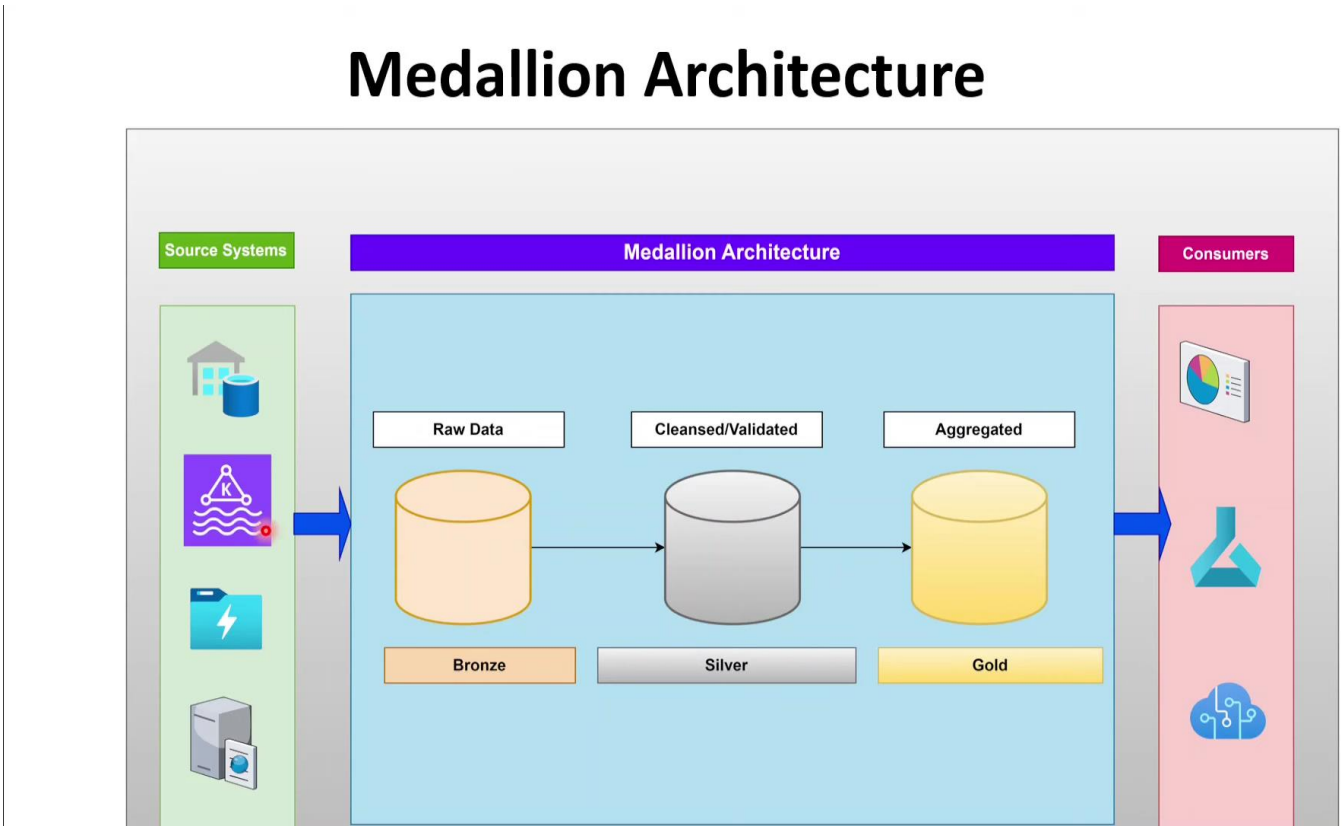
Under root folder, if we want to consider all subfolders , so * . * means any. So, we can use any wild card option while reading the data from the source system.



Once we have ingested data from the source system, we have to write the data into our target system.in the medallion architecture, we have to write the data within bronze, silver or gold layer so for that we can use writeStream command. So for trigger either we can use once or we can go for foreachBatch.

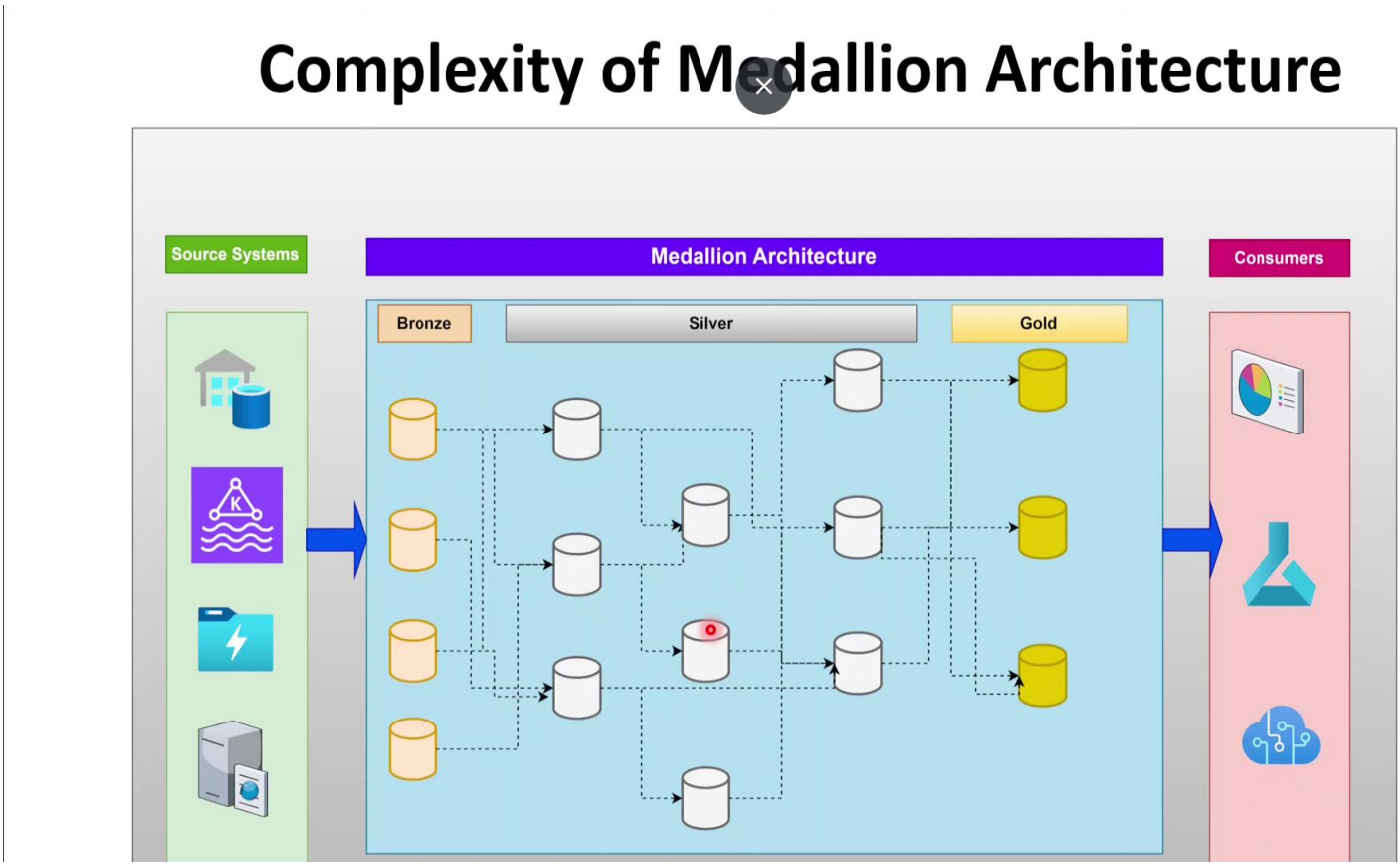
DELTA LIVE TABLES

With the help of delta live tables feature, we can build data pipelines efficiently. Medallion architecture is common or standard architecture within lake house. Lake house is nothing but its one of the data platform which is combining best features from delta lake plus data warehouse. Medallion architecture is having 3 layers Bronze Silver and Gold. ETL pipeline will connect to different data sources and it will ingest the data and putting raw data into bronze layer in the raw format itself. This is mainly used to consolidate raw data from desperate data sources. Once that is done, in next layer data cleansing/data validation process is applied. We are having some null value either we have to remove the null value or we have to put some default value to handle the null value . even if we have duplicate value even that can be removed. Or lets say we have date format from different source systems , we are receiving date format in different format actually . In one of the source system, we are having year at the beginning . in some of the source systems we are having day number at the beginning.



But coming to our system, we have to follow some standard pattern so those kind of data confirmation/data validation checks are performed within the silver layer. Now in the gold layer core logic according to the business use case is applied, it could be various transformations or business aggregations . Once we have completed the data transformations and aggregations then the final data can be supplied to different set of customers and consumers can consume the data from gold layer. This is the standard medallion architecture. Here we have only one component for each layer . But in the real time, whenever we are scanning through lake house architecture, it's not going to be that simple.

In real time medallion architecture is more complex or any ETL pipeline that we are building that is not going to be straight forward. It is not going to just receive data from data sources and directly it is moving from one table to another table and then from that to another table. That's not going to be the case. In real time our data pipeline is going to connect to various source systems and within each layer we are going to have multiple objects/multiple schemas . even one particular layer can be logically divided into multiple layers inside. So, we will have multiple different data objects , it could be tables, views, managed table, external table, different types of objects we will have and difference between these objects would be complex.

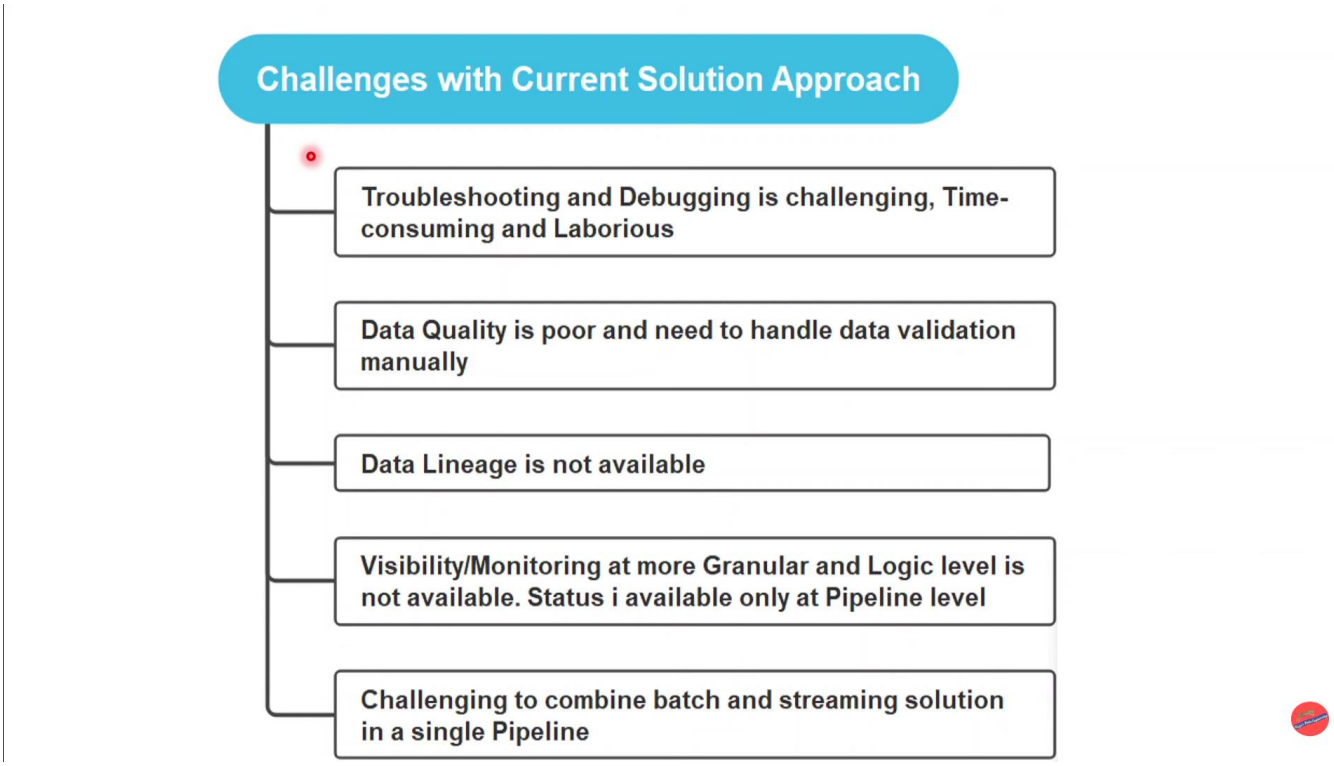


For example, if we see above screenshot, one silver table depending on another silver table, and this silver table depending on bronze table and in certain cases it will be more complex. In real time projects, even we will have much more complex architecture. So, when we have such complex architecture what are the problems we will have ?

So first of all troubleshooting and debugging is going to be very challenging because whenever we are building ETL pipelines we are going to monitor the status at pipeline level . which means we are not going to monitor logic only for this particular component. The pipeline which is responsible to ingest data from different sources and applying certain validations and transformations so we don't have proper monitoring mechanism for each logic so that's the reason it's going to be difficult in terms of troubleshooting and debugging because of that it is going to take more time and more manual work.

2nd thing is coming to data quality it's going to be poor and each and every stage manually we have to write lot of code in order to handle data quality and we don't have any mechanism to monitor those data quality checks. Let's say we are having some corrupted records or certain records which is deviating from standard then we don't have any metrics how many records violated so we don't have any mechanism manually we can build create some table error log table or we can create some location in the ADLS

in order to track those fail records and it's going to be time consuming in order to create that manual logic in each and every place. So data quality is going to be poor and we need to handle that manually.



Next important problem is data lineage is not available, whenever we are building ETL pipelines at high level we don't have data lineage , what is the dependency we are having between different components that we don't have. We don't have any tool to monitor those lineage . for that we have to manually document , we have to understand the code and manually we have to document. Let's say there is problem with silver table so in order to fix the issue first of all we have to understand what is the dependency of this table and what is the downstream and upstream objects. For example, particular silver table depending on 2 tables, and for that table , gold table is the consumer so we don't have any such mechanism to monitor the lineage visually. Instead of that we have to go through the code and we have to understand. For example, we have certain logic for the table and at the end of the table, it is going to load data into another table . something we have to understand manually through the code then we have to understand the lineage then we have to fix so we don't have any proper lineage.

Also, we have challenges with monitoring and visibility at the granular and logic level. For example, we don't have any metrics or statistics available for each logic. For example, we are having some logic in order to populate the table data or we are having certain different logic to populate the table . so, at the end of the

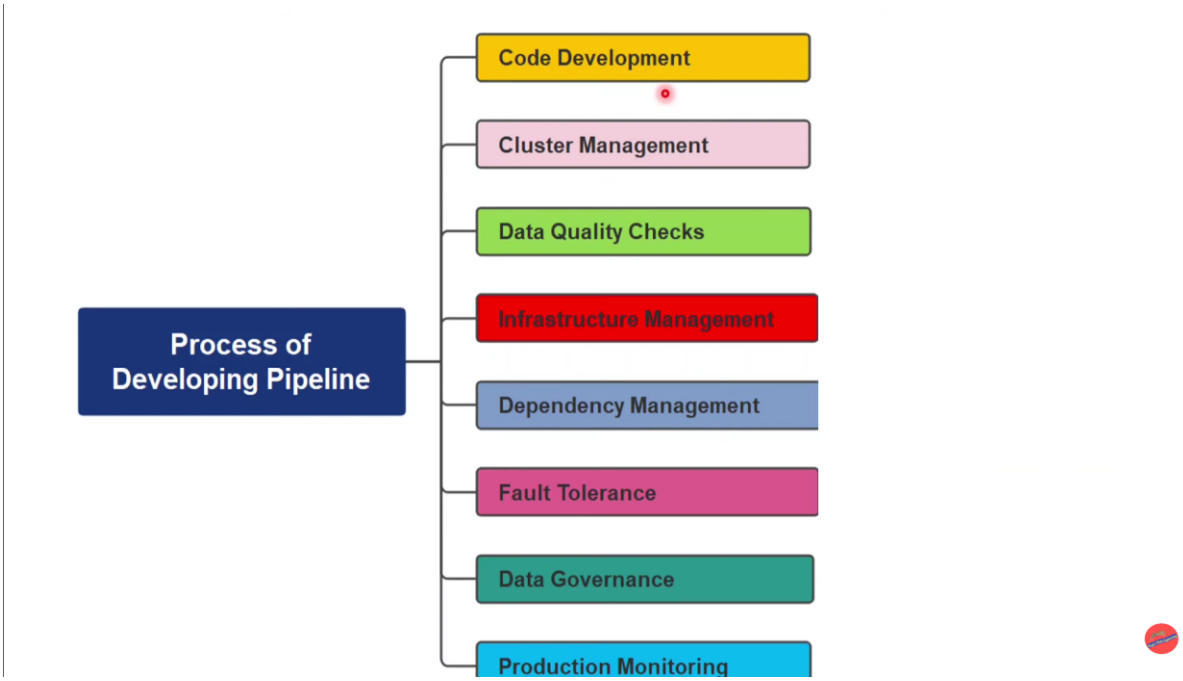
completion we are going to have complete status at the pipeline level. We are not going to have any statistics related to particular logic , how many records got processed for this particular logic and how much time it has taken. So, we don't have any information so for that again we have to go through the pipeline and we have to understand or we have to go through the spark user interface then we have to understand how much time it has taken so we don't have any centralized tool or location to track and monitor progress of the pipeline.

Last challenge will be combining the batch and streaming process. For most of the fact or real time data we are using streaming process. But for most of the dimension tables still we are going with static data. So, batch processing is applied for dimension table and streaming data that is applicable for fact tables and in certain time, we have to join fact and dimension tables so for that we have to mix batch and streaming process even though this option is available in databricks even before delta live table but that is quite complex compared to delta live table approach. So, this delta live table can simplify that process as well . so, these are the challenges with current ETL pipeline building approach.

STEPS TO FOLLOW TO BUILD ETL PIPELINE USING DATABRICKS

There are several steps in order to complete ETL pipeline starting from design until deployment . we have to take care of different components different processes

- First of all, we have to perform code development
- Secondly, developer should concentrate on cluster management like what kind of cluster we need to choose and what should be compute power for each cluster and how many number of worker nodes we need and what should be memory size , no of cores for driver and worker node and autoscaling is needed or not , termination is needed or not. For development we can go with all purpose cluster and for production we can go with job cluster. So, for that clusters we have to define the template.
- Apart from that, developer has to focus on data quality checks . whenever we are receiving data from different source systems , it will come with lot of data defect , there could be some problem with the data quality like there could be some duplicate value, some null value. Data quality check is another important part in ETL pipeline . for that databricks developers has to write lot of logic in order to handle data quality manually. This is another aspect while developing the solution.

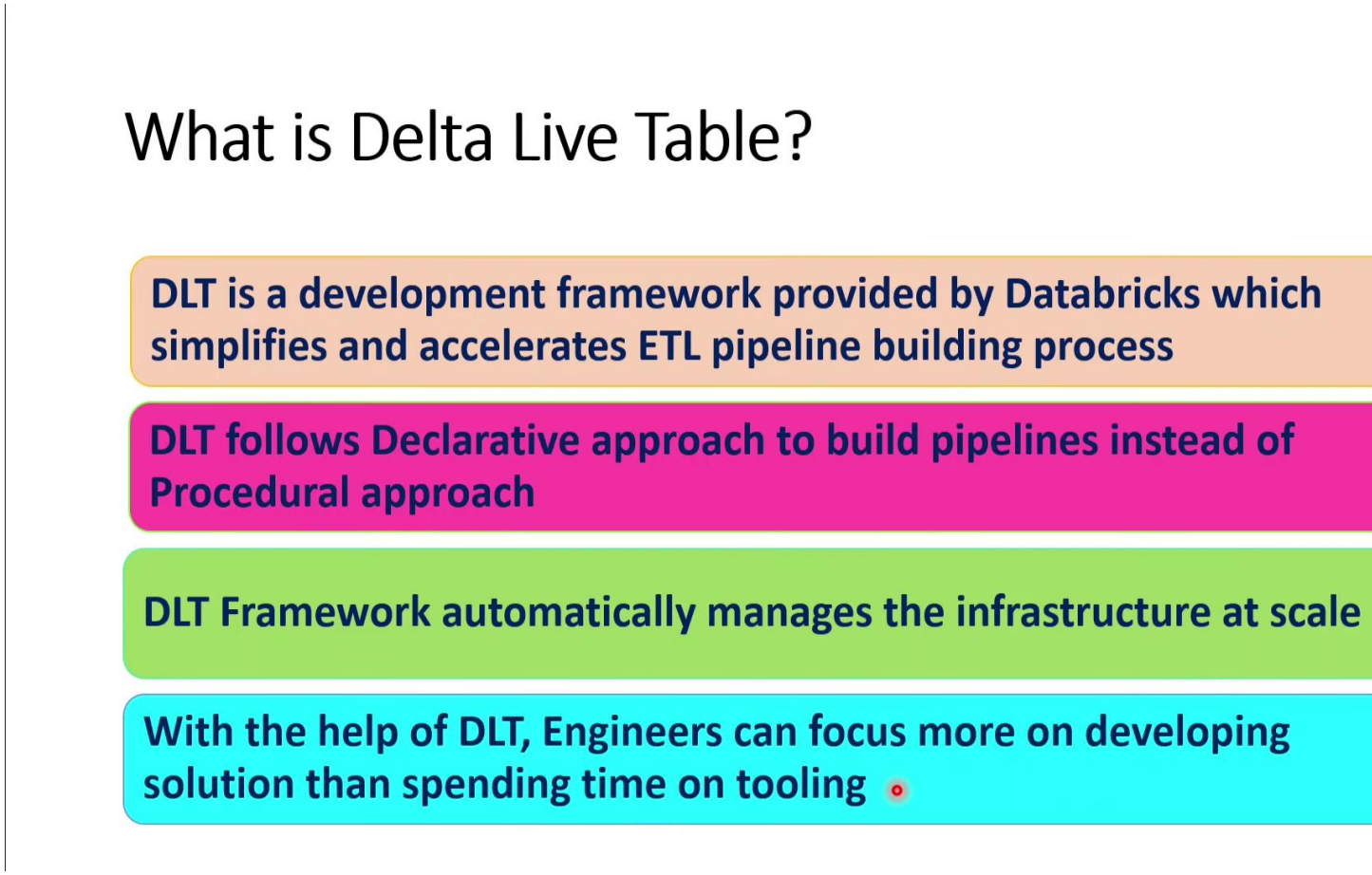


- Next thing is infrastructure management. Cluster management is also one of the infrastructure management . Apart from that we have to look into spark user interface in order to understand with the help of executors , worker nodes and driver nodes. We have to manage the infrastructure, we have to monitor and we have to make some corrective action if something goes wrong. So, infrastructure management is another important aspect of development.
- Dependency Management is nothing but understanding data lineage how one particular logic or one particular table is related to another tables. that is very important because with the help of that, debugging or troubleshooting will become easy so for that we have to go with documentation manually we have to document or each and every time developer has to go through the code in order to understand the lineage and make necessary changes after understanding the lineage.
- Fault tolerance is nothing but whenever data pipeline is getting failed then how we can handle that failure scenario so for that we can go with different approaches like retry or manually we can debug and troubleshoot after that we can rerun the pipeline manually. This is also manual process.
- Data Governance is nothing but what kind of data we are receiving , what is the life cycle of that and who can access that data . Any developer has to focus on data governance as well .
- Final thing is production monitoring. Once pipeline is created and scheduled in production, then it should be monitored time to time. If there is any issue with

the pipeline execution then that should be fixed. So currently we don't have any centralized location where we can go for monitoring. So, these are the different components or processes or aspects each developer has to focus while developing the pipeline.

Any developer has to focus more on core development and all other steps it is not necessary for developer. There should be some framework or some mechanism to handle all these process automatically then we can speed up our development. So, developer can focus only on core development instead of focusing on tool and processes. So, these are the ideal steps needed to develop ETL pipeline but currently developer has to focus on the development along with has to focus on tools and processes. So, if we get a framework or mechanism to eliminate all components except code development. If we are having developers then we need to focus only on code development then that is going to speed up the development process now solution for that is Delta live tables.

Databricks has come up with some framework , developers can simply use this framework and with the help of this delta live tables , developers can focus only on the code development and remaining other process or other components will be taken care by delta live tables.



Delta Live Tables is using declarative approach. In our regular ETL pipelines within the databricks , we are following procedural approach.

Delta Live Tables will perform data transformation internally. It will manage all the transformations within delta live engine. So we have to just declare what kind of transformations we need or what kind of end output we need then delta engine will think internally get best approach and it will start applying all the transformations automatically. So in the procedural approach we have to define transformation steps one by one manually in proper order. Databricks engine that will execute all the steps one by one but coming to delta live table, we have to tell what we want. Then it will start thinking what kind of transformations it has to apply and how it can arrive to that final output .

Coming to cluster management explicitly we don't need to create or manage any cluster , we have to just create delta live table workflow internally it will manage the cluster management.

Coming to data quality validation, in regular databricks notebook, we have to use different logics to handle null or duplicate handling , different data validation, data quality checks or data transformation activities we have to perform manually through coding but coming to declarative approach we can just tell what kind of data quality we are expecting . for example , one of the particular column that should not accept null value or one of the particular column it should not accept any value greater than certain limit so we can just tell during the declaration then automatically it is going to manage all those data quality validations. With the help of this things entire ETL pipeline will be accelerated.

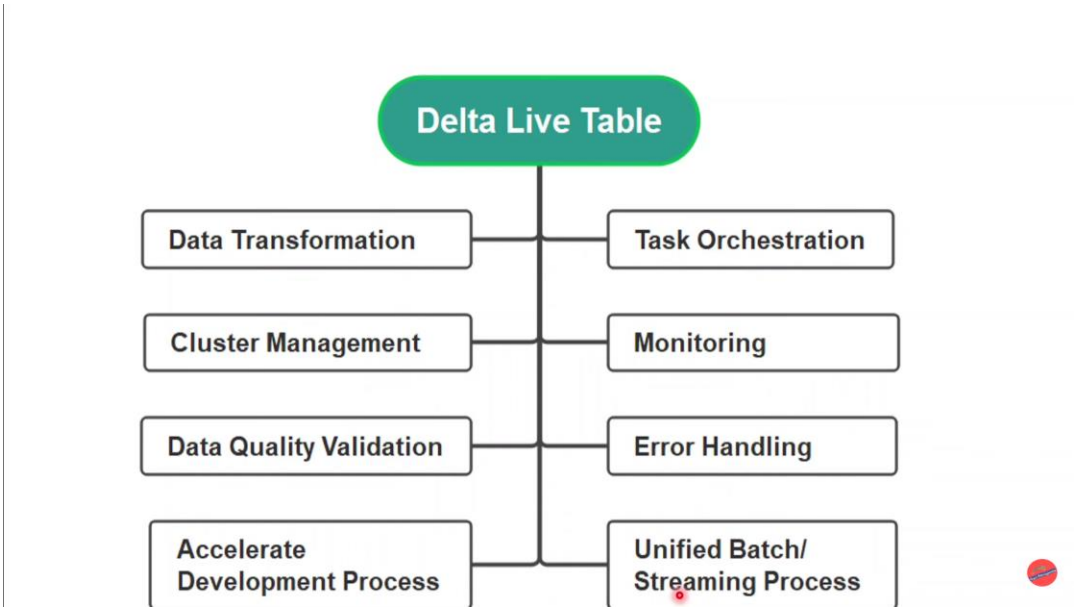
Coming to Task Orchestration, we can just tell delta engine what we want. It will just think internally how it can be done. So basically, it is going to orchestrate all the internal task one by one then it will execute them manually , we don't need to manage that explicitly .

Coming to monitoring, this is one of the important component of delta live table. With the help of UI, we can monitor. For each and every component in entire pipeline we can understand the metrics. For one particular table loading how much time it has taken, how many records it got processed, how many records got removed due to this data validation, still we can get many more metrics more information from the user interface. So, it's one of the efficient centralized platform to monitor delta live tables.

Coming to error handling, there are some options provided by delta live tables. with the help of that, error handling can be handled efficiently , retry option can be used so delta live table will automatically retry and also if one of the particular logic failed internally then we can start the re-execution starting from that particular

logic which is not possible with the procedural approach. In current procedural approach, incase one of the particular pipeline failed then we have to rerun the entire logic starting from beginning but delta live table is giving efficient options to handle the error.

Delta live table is giving option of handling batch and streaming process together.



Summary : In order to create delta live table, we have to create live tables using notebook . we have to create different live tables by defining set of logics so for each and every live table we have to tell what we want. We have to define certain logic through declarative approach. In normal notebook, we follow procedural approach. One of the major difference when compared to procedural approach is in procedural approach, we can execute each and every cell during the development in order to test the logic but coming to delta live table, we cannot execute that in regular notebook. So, once we have defined all the logic, we have created delta live table pipeline then we have to create a workflow . for that we have to get into workflow in databricks then we have to get into delta live tables tab then there we have to create workflow. There we have to choose notebook which we have developed in the first step and also, we have to configure different parameters which are needed for this particular workflow . Once we have created delta live tables workflow then finally, we can run that immediately or we can schedule that for a particular interval then once we have executed the pipeline then we can monitor using user interface provided by delta live table.

DECLARATIVE VS PROCEDURAL APPROACH

PROCEDURAL APPROACH Procedural approach is used in data bricks ETL development when we are not going with delta live table. So, whenever we are building ETL pipelines using regular notebook , regular approach that is called procedural approach.

Coming to procedural approach, we have to explicitly outline all the ETL steps . coming to ETL steps we have to specify logic or we have to write the code for extraction and applying various transformation then loading the data into target system. So, for each and every step, we have to explicitly create the logic and we have to write the code. So that is procedural approach. In order to achieve certain final output, there would be list of steps that we need to perform then only we can achieve final output so we have to follow all the steps in proper order. If we are going to change the order then its going to produce wrong output. In procedural approach, we need what kind of output we need but along with that , we have to define we have to tell how we can achieve that output. So, for each and every step, we are going to create logic saying that how we can achieve final output. Development is going to be time consuming and is error prone approach. If there is some problem then it is very difficult to understand in which step, we go wrong then debugging is going to be more complex and troubleshooting is going to be more time consuming. Stored procedures in any traditional SQL language or regular spark application or ADF these are examples of procedural approach. We will be using all iterative or conditional logic like if else in order to create different branches or switch statement, while loop , for loop, we will follow all procedural approaches.

DECLARATIVE APPROACH Declarative approach is quite opposite to procedural approach. In declarative approach, we will just declare what kind of final outcome we are looking for. So, all the internal steps will be abstracted. We will not guide the engine step by step. There will be some intelligent engine that will start thinking on our behalf and it will come up with best efficient plan so all the ETL steps are abstracted and there is no specific order to be followed just we can declare what are the items we need. We don't need to produce huge amount of code and debugging and troubleshooting is also very easy and it can be quick.

Eg for declarative approach – DBT(data built tool) and DLT – delta live table.

DIFFERENT TYPES OF DATASETS IN DELTA LIVE TABLE

We are having 3 different types of dataset for delta live tables.

first one is streaming Table.

2nd one is materialized view or live table. So, both are interchangeable either materialized view or live table both are same concepts.

3rd one is view.

STREAMING TABLE

Streaming table is more suitable to consume incremental data from the streaming data sources. Streaming data sources work in the concept of append-only mode which means in any of streaming source system the data will not be changed for the old data. Examples like Kafka, Azure Event Hub, ADLS , Kinesis. These are examples of streaming data sources. When we have to consume data from these kind of source systems we can go with streaming table. So, these kind of source systems will keep on adding new data files or new data records time to time but it is not going to update any old records. As a result, we don't need to reprocess any of old data which means any data file or data record can be processed only once in streaming table. Coming to processing method streaming table is mostly using autoloader. Coming to streaming table, autoloader is not the only option even we can go with Kafka or other streaming solutions as well but autoloader is one of the most commonly used file format or ingestion tool which is used for streaming table. Coming to autoloader, that is used to automatically detect and incrementally process new data files as soon as they arrive. Internally autoloader is using cloud storage and message queue in order to complete the incremental ingestion.

DIFFERENT USE CASES WE CAN GO WITH STREAMING TABLE

Whenever we have to process huge amount of data which is growing with high velocity and where low latency is expected. For these kind of scenarios, we can think of streaming table and also certain use cases where we don't need to recompute old past data . we don't need to touch old data. Whenever we are getting new data we have to focus only on the new data then for those kind of use cases also we can think about streaming table. And streaming table is more suitable for extraction stages. In ETL pipeline we have 3 stages Extraction , Transformation and Loading. Streaming table more suitable for extraction stage which means bronze layer. So basically, this extraction stage is dealing with different source systems. It is integrating with source systems so whenever we have to consume data from streaming sources like Kafka or azure event hub then we can think of building streaming table.

MATERIALIZED VIEW

This is quite similar to traditional database. Materialized view means basically we are defining one query which is consuming data from one or more than one table by joining and it is applying various complex transformations as well then, that final query would be executed and output will be stored in one of the table. So that is called materialized view. In regular view also we are defining a query but execution will happen dynamically each time while we are calling that view but it is not storing that data anywhere but coming to materialized view, we are defining a query , the query

will be executed then output will be stored as a live table. Materialized view is also called as live table. So, we are defining a query that query will be executed . output of that query will be stored as a live table.

USE CASE OF MATERIALIZED VIEW

Materialized View is more suitable for CDC(Change Data Capture)related use cases which means incase some changes needed for the past data then we can go with that any DML operation it could be insert, update or delete anything can be handled for those kind of scenarios.

Materialized view is nothing but result of a query is stored as a live table and manually we cannot insert or update or delete within materialized view. Incase we have to make some changes to the data output then we have to redefine our query. We can adjust only the defined query. Directly we cannot apply any changes on the materialized view and materialized view is refreshed periodically . we can create some update schedule. Based on the schedule it can be refreshed time to time.

Materialized View is mostly suitable for transformation and aggregation stages which means either silver or gold so basically in silver or gold layer what we are doing is we are consuming data from multiple tables by joining then we are applying various transformations and aggregation so materialized view is more suitable for silver and gold layers.

VIEWS

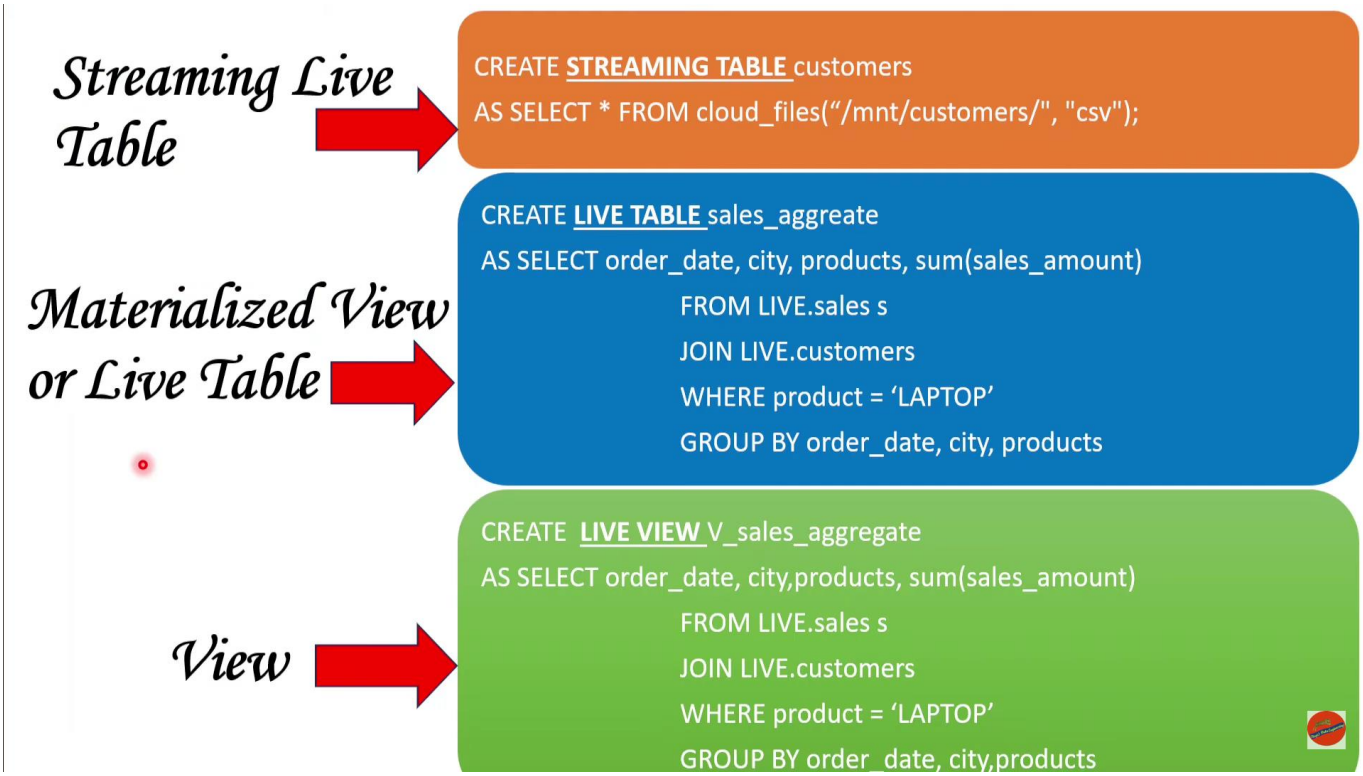
Views is quite similar to materialized view but only difference is materialized view that is storing result set as a live table which can be available within the data bricks catalog but coming to view , here also we are defining a query which will be executed whenever we are calling the view but query output that is not going to be persisted as a live table so it is not available into the catalog which means data is not publicly available for analytics and also this view can be referenced only within the pipeline where we have defined because this is not going to persist the data which means this particular view cannot be referenced in any of the pipeline, outside this particular ETL pipeline so in which pipeline we are defining this view , only within that pipeline we can reference.

USE CASE OF VIEWS

This is more suitable where end users / downstream application are not needing the data because we cannot see the data . Once we have executed then this is not stored in the catalog so we cannot see the data so where the data is not needed for end users or downstream applications then we can go with view. This is mostly suitable for data quality checks and validations. And in certain cases, we are having very complex logic for one of the dataset , it can be materialized view or streaming table so we can break

that complex logic into multiple substeps so for that we can use view. So initial steps can be converted to view then final step that can consume the data from this view (intermediate result set) then it can materialize the data as either streaming table or materialized view. These are the different use cases for view.

SQL SYNTAX:



Delta Live table is supporting only 2 programming languages at the moment. One is pyspark and 2nd is spark SQL .

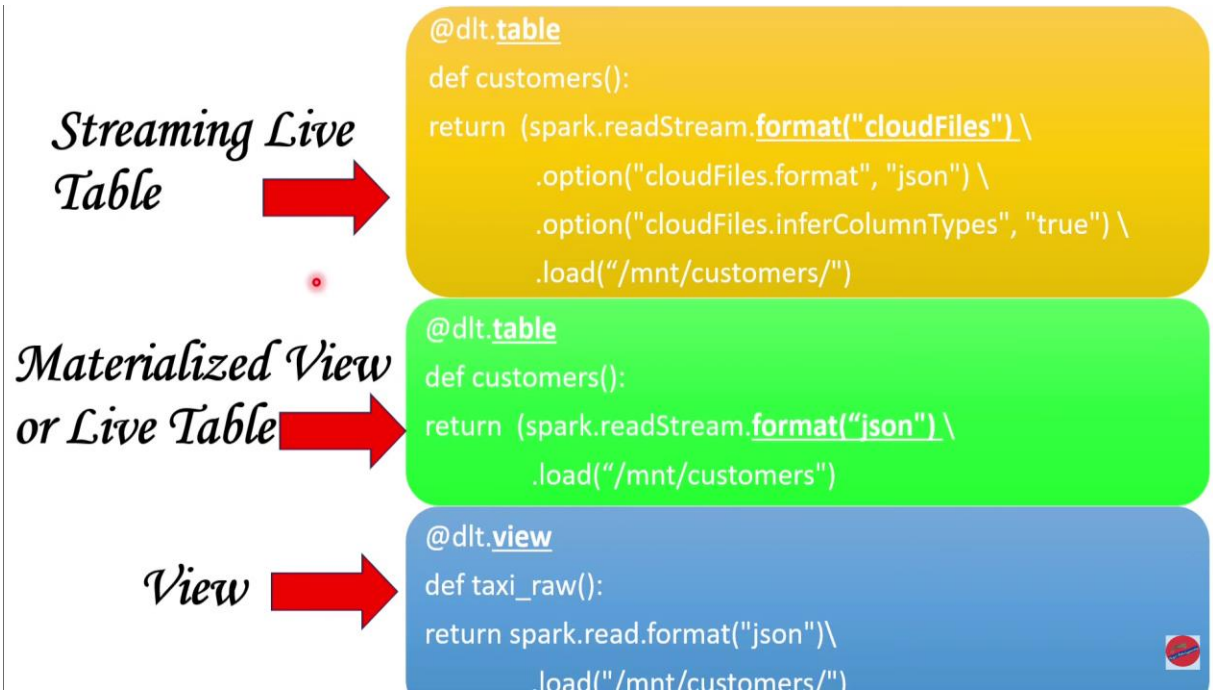
In Streaming Live Table , in syntax we need to use keyword STREAMING TABLE then we will give name for streaming table here it was given as customers then we need to give ingestion query (we have created mount point with databricks . that file is in the form of csv files. We are using cloud_files which means this is autoloader, this is one of the streaming source then we are consuming data from this streaming source . based on that , data will be persisted as a streaming table in the name of customers. So, this is how we can create a streaming table in delta live table using spark SQL table.

While defining streaming live table, we can apply many other parameters or properties .

For materialized view, we can use LIVE TABLE keyword in order to create it. Here we have a query joining 2 tables one is sales and another is customers both are live tables(can be streaming table or materialized view). In previous stage we have defined some datasets and in particular step we are consuming data from those 2 different datasets(sales and customers) and applying various transformations and then finally output will be persisted in materialized view (sales_aggregate).

View will be created by keyword LIVE_VIEW

PYSPARK SYNTAX:



Coming to pyspark, we need to use python decorators which means we need to import the library delta live table library. Once that is done, we need to use @dlt.table. @dlt.table → This is basically creating the table. @dlt.table is common for both streaming live table and materialized view in pyspark. So how come delta live engine will understand which one is streaming live table and which one is materialized view. So, delta live engine will basically check what kind of data source we are consuming data from . so, for example in streaming live table logic, data is being consumed from the cloudFiles which means autoloader which means it is one of the streaming data source. So, if we are going to consume data from any of the streaming data source then it will create streaming live table. And incase if we are going to read data from any static system then it is going to create materialized view. This is the difference but in terms of syntax both are same.

If we are going to consume data continuously then it is going to be treated as a streaming live table. And if we are going to consume data from one of the static source

then it will create materialized view. Coming to view syntax is @dlt.view and inside we can define the data source from where we have to consume the data .

TYPE	Great Fit for
Streaming Live Table	✓ While ingesting data from streaming data sources like Kafka, Event Hub etc., ✓ When no need of re-computing old data ✓ processing incrementally growing data of high volume with low latency
Materialized View	✓ While many downstream queries or processes are depending this dataset ✓ When this dataset is needed for other pipelines ✓ View and analyse the dataset during development stage
View	✓ Large query or complex logic for a dataset which can be broken into easier-to-manage queries ✓ Validating the datasets using expectations ✓ Reduce the storage by not publishing the datasets for end users and systems

Coming to materialized view, in case we are applying complex logic and that data is needed for multiple downstream queries or processes then we can go with materialized view because materialized view that is executing one query then it is persisting that output as a live table so all the subsequent process can consume the data . when we have to use this particular materialized view in another pipelines then we can go with materialized view because in regular view that can be referenced only within that particular pipeline but incase we have complex query that output should be used in multiple other pipelines then we can go with materialized view. Incase if we have to see the data and analyze the data during the development stage then we can go with materialized view because whenever we are executing it is going to materialize the data inside the catalog so we can see the data what kind of output it has produced then based on that we can test our data so for that scenarios we can go with materialized view.

Coming to view, whenever we are having very huge or complex logic for one of the dataset like streaming table or materialized view then what we can do is we can break that into multiple smaller queries for that scenario we can go with view. This is also more suitable for data quality checks and data validation using expectations. In certain cases, we have to reduce the storage because view is not storing the data

physically which means it is not occupying any storage space. If we have to reduce the storage for our use case then we can go with view.

We can use one of the delta live table dataset.

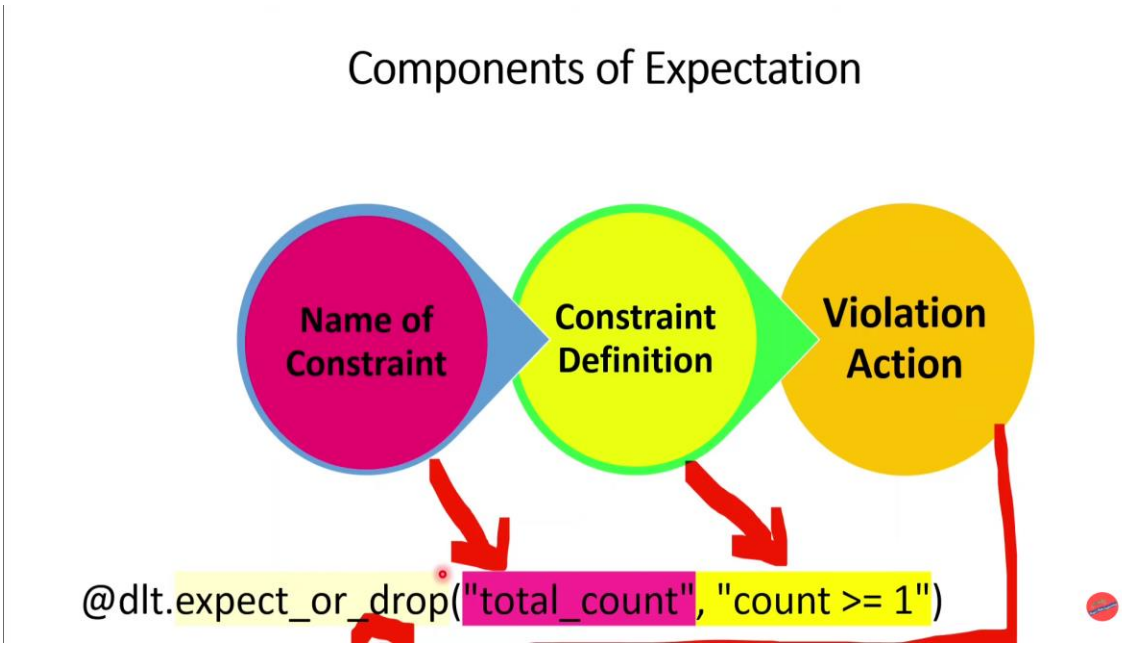
DATA QUALITY EXPECTATIONS

With the help of expectations, we can perform data quality and data validation operations in delta live table.

Expectation is one of the feature provided by delta live table in databricks to perform data quality and data validation checks. In any data engineering project, validating the data is very important step. We have to identify corrupted or bad records and we have to put proper mechanism to handle that. With the help of expectation, we can define different constraints on column of a dataset. Dataset could be tables or views which is containing list of columns and we can put certain constraints data validation checks on the column with the help of this expectation. Expectations are optional, even though it is optional, it is very commonly used in all the delta live table developments. We can define expectation with the help of python decorators or SQL Constraint class. Python decorators is one of the concept in python using which we can extend the functionality of existing method so we can add extra functionalities on top of the existing user defined function that is called python decorators. With the help of this expectation, basically we can identify good and bad records and also apply actions on violated records.

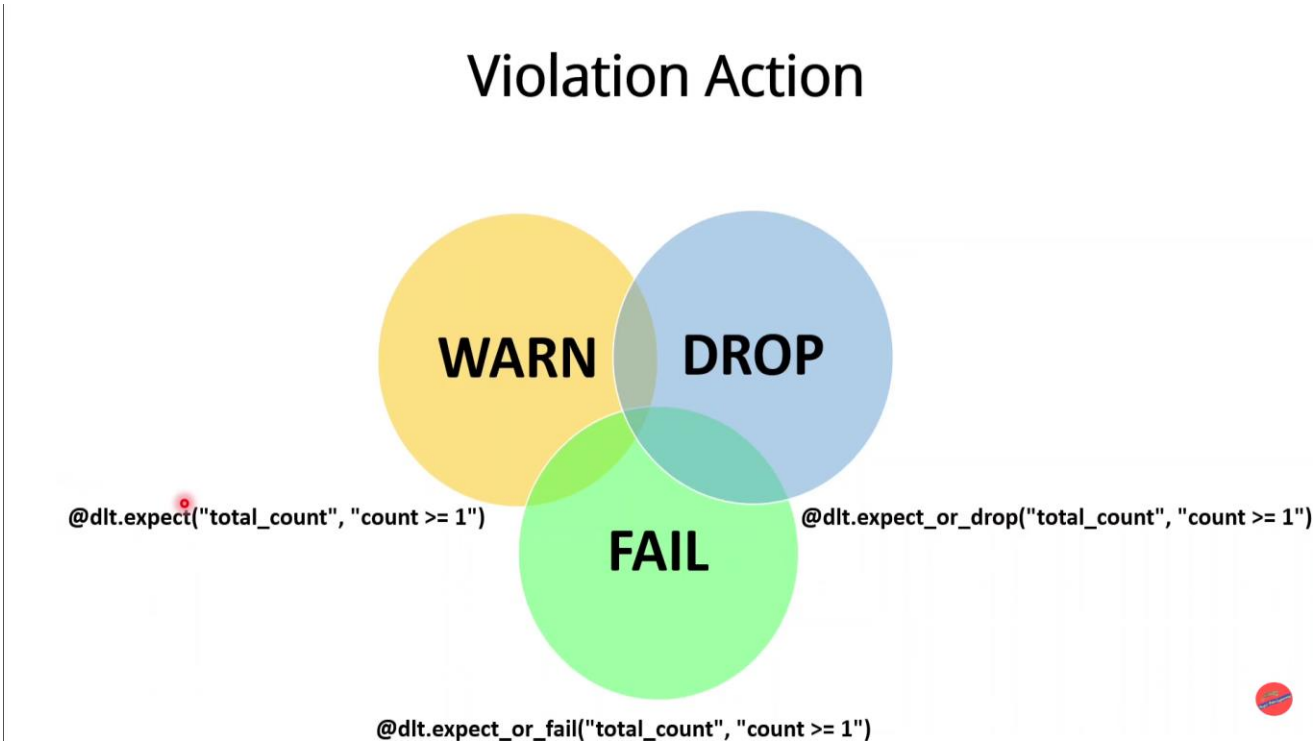
Expectation is efficient framework provided by delta live table using which we can define data quality checks instead of writing verbose logic on our own. Incase this expectation feature is not available, then with the help of pyspark or spark SQL manually we have to write lot of code to perform the validations but with the help of this expectations the entire data quality check is simplified. Simply we can configure expectations then according to that we can perform the data quality checks. Expectation is having 3 major components. 1) expect_or_drop 2) total_count 3) count>=1

total_count is name of the constraint. This is user defined name , we can give anything.



Count>=1 is the constraint definition. This is the place where we are performing the actual data quality check so here in our dataset, we are having column called count so that count column cannot accept any value which is lesser than 1 . here we are defining using the condition logic count>=1 , this is the definition of the constraint .

Coming to 3rd component, violation action . incase there are some records which is violating defined constraint then what kind of action should be performed by the delta live table engine so there are 3 different types of actions. Warn, Drop and Fail.



The above screenshot shows syntax for warn, drop, and fail.

@dlt.expect(constraint name, logic) → warn syntax.

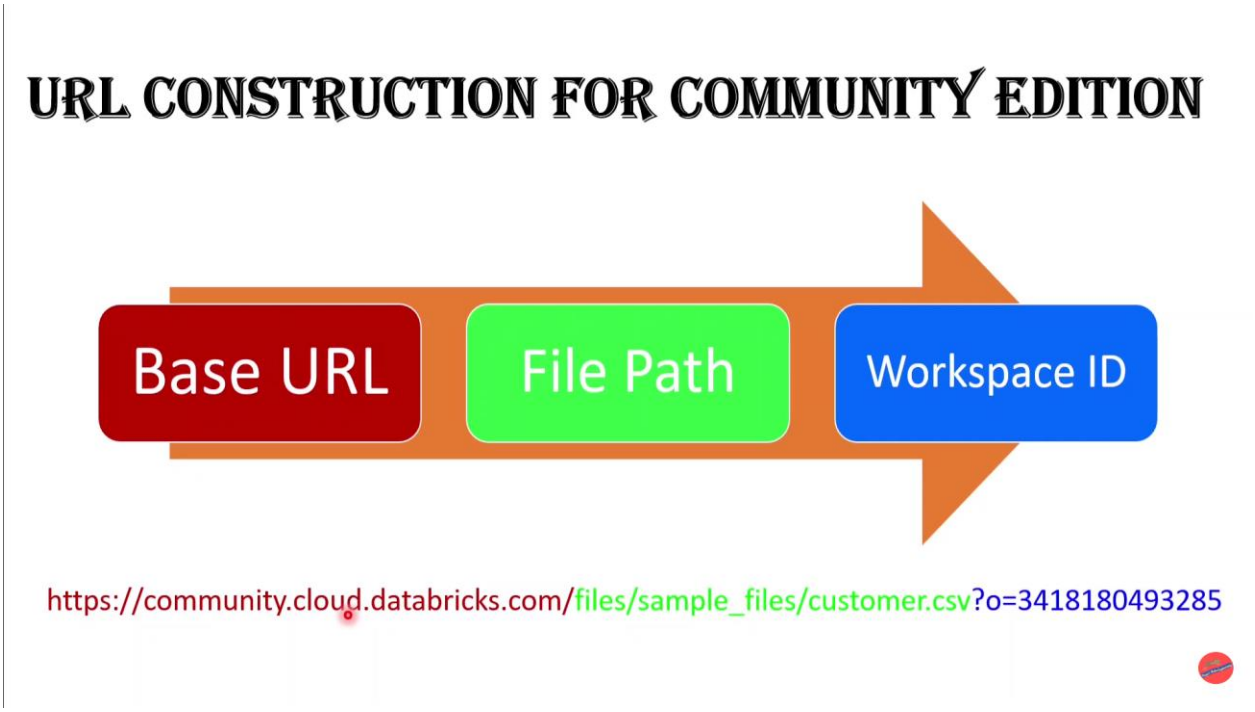
So even though there are some bad records or violated records still the process will continue and even corrupted or bad records will be written to the target system but at the same time only in the execution metrics , delta live engine will display number of records which got violated so basically, we can see the warn records no of records but still even the violated records will be written to the target table.

2nd action is drop for that we will use keyword @dlt.expect_or_drop. Incase if there are certain records which is violating this constraint, then those records will be removed from the processing will be dropped. Only the records which is complaint to that definition of constraint only those records will be written to the target table.

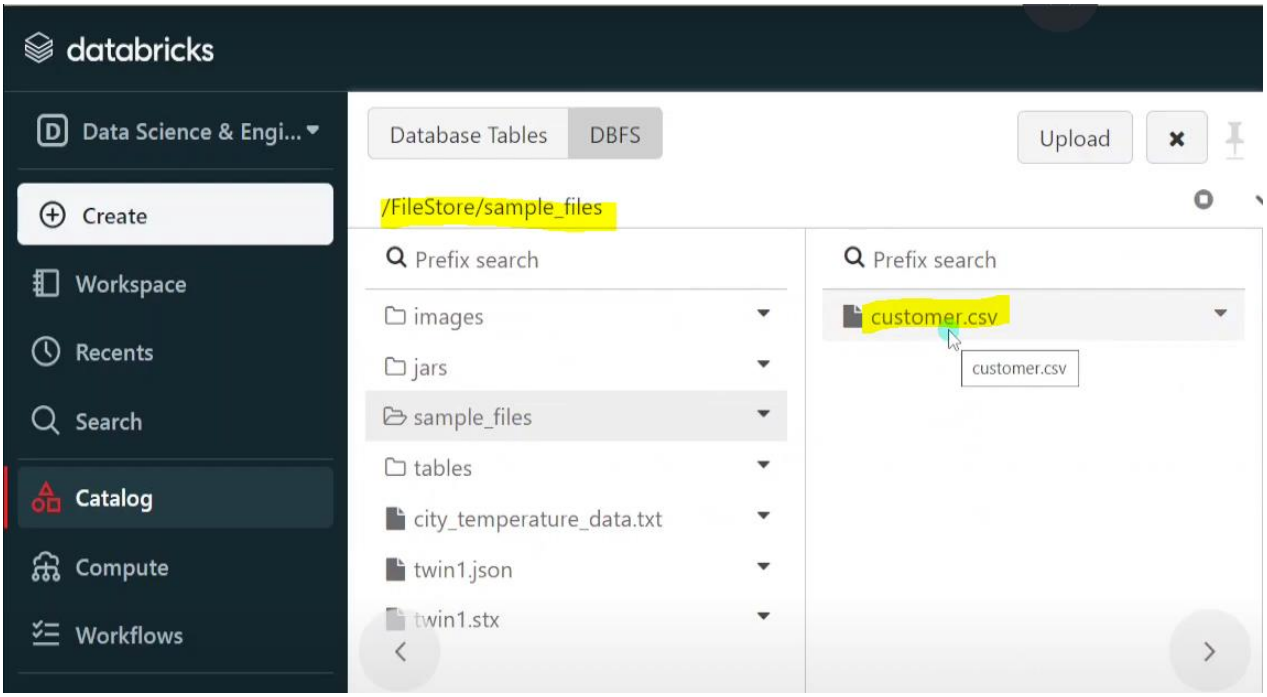
3rd action is fail. For that we will use @dlt.expect_or_fail. Inside we will define meaningful name and constraint. Incase there is some violation at least for one record then entire process will be failed then manual intervention will be needed so developer has to troubleshoot and they have to understand what the reason for this failure is then they have to correct that record then we can rerun.

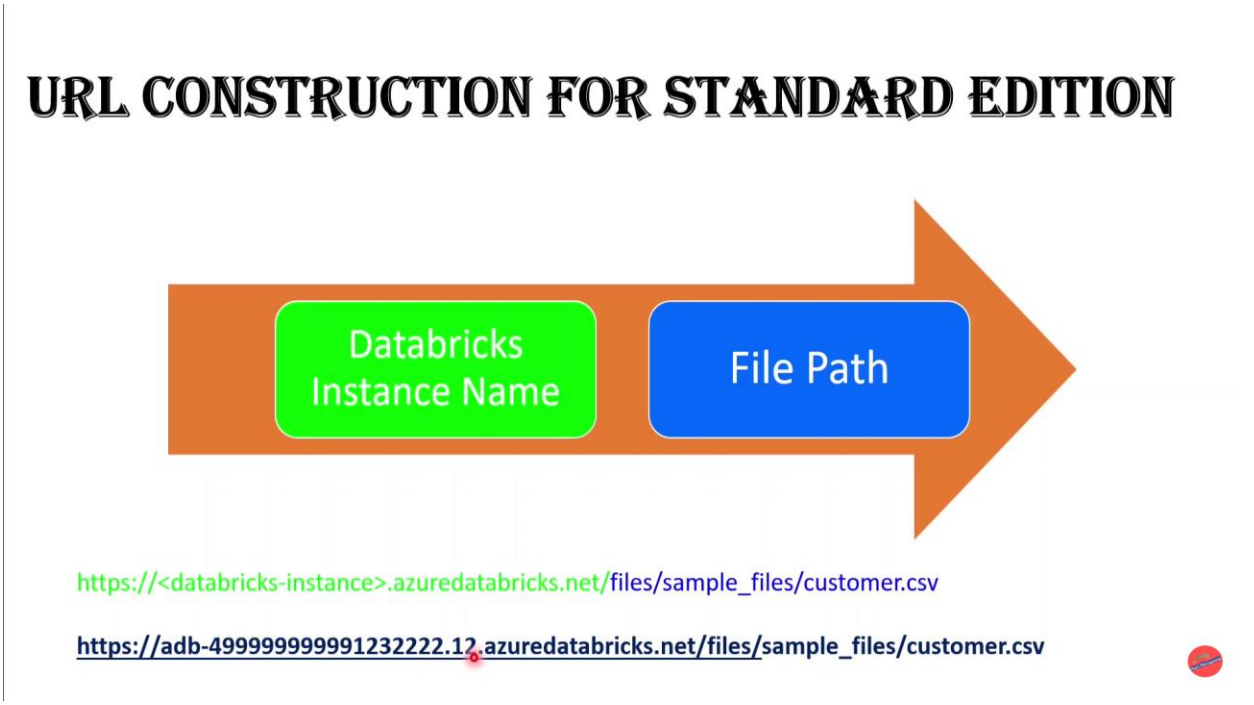
So, in all the cases we can see the metrics in the execution logs .

HOW TO DOWNLOAD FILES FROM DBFS LOCATION



We need to copy path location only from sample_files/customer.csv and paste it in base URL https://community.cloud.databricks.com/files/ sample_files/customer.csv path location we will get from below screenshot. We will copy path only from sample_files/customer.csv and workspaceid we will get it from databricks community edition link url.





BUILT-IN FUNCTION TRANSFORM FUNCTION

With the help of this transform, we can apply custom transformations to single/list of columns or all the columns within a data frame . we need to chain many such transformations , then during those scenarios , transform function is very efficient and handy. It is actually allowing us to apply element wise operations using lambda functions or applying logic at entire column level so making our code more concise , readable, and efficient. Usually, UDF are not recommended in pyspark development because it has contact switching between java platform to python platform so it will hit the performance but coming to transform function, it is actually optimized internally so it is not having any performance bottleneck . This will process transformation more efficiently .

TRANSFORM FUNCTION ADVANTAGES:

1) MODULARITY : within transform function, we actually break down complex transformations into smaller reusable components so this modular approach makes it easier to manage and maintain our code so instead of writing monolithic functions to handle complex and multiple tasks , we can create small single purpose functions that can be combined in various ways to achieve the desired output.
2) READABILITY : code readability is crucial when working in collaborative environments . transform function along with lambda expressions makes our transformations very clear and easy to understand. This readability means anyone looking at our code can quickly grasp what each part of the transformation is doing.
3) REUSABILITY : The term reusability allowing us to apply same transformation logic, same modular function across different dataframes within a particular usecase or within different usecases within a project/within multiple projects or within organization. We can split the logic into separate user defined function and we can put all the commonly used such functions into one of the utility notebook and that notebook can be called. In this way, we can reuse existing logic.

USE CASES OF TRANSFORM

1) Data cleansing : Transform function is more suitable to remove unwanted characters from a string or handling white spaces trimming white spaces , handling null data , populating missing values, handling duplicate data or making the data more standard in terms of data types. We can create separate

user defined functions to handle missing values , duplicate values across different usecases.

2) Data Transformations - incase our notebook is having lot of complex transformation logic , instead of creating monolithic logic we can segregate the logic into separate modules. Then it is more readable and also for debugging and troubleshooting. When we have complex transformations, it is better to go with transform function.
3) Feature Engineering – This is more suitable for machine learning related use cases. In machine learning models, feature engineering is one of the important concept in terms of training . so, in the context of machine learning, transform function can help to create new features from the existing data such as extracting elements from the text, creating interaction terms or normalizing values. So, this is actually enhancing our dataset and improving performance of our models.

CODE

```
#Created Sample DataFrame
from pyspark.sql.functions import col
data = [
(“Product1”, 100 , “Category1”),
(“Product2”, 200 , “Category2”),
(“Product3”,150,”Category1”),
]
columns = [“ProductName”,”Price”,”Category”]
df = spark.createDataFrame(data,columns)
df.display()
```

Table +

	ProductName	Price	Cate...
1	Product1	100	Category1
2	Product2	200	Category2
3	Product3	150	Category1

Now we want to perform calculated column DiscountedPrice based on the Price column . Transform is accepting input parameter of another function.

Waiting

2: TRANSFORM Function

```
# Define transformation function
def transform_function(df):
    return df.withColumn("DiscountedPrice", col("Price") * 0.9)

# Apply transform function using transform method
transformed_df = df.transform(transform_function)

transformed_df.display()
```

Here in above screenshot we have directly given 90% in calculation

Table

	ProductName	Price	Category	DiscountedPrice
1	Product1	100	Category1	
2	Product2	200	Category2	
3	Product3	150	Category1	

3: TRANSFORM With Parameters

```
from pyspark.sql.functions import col

# Sample DataFrame
data = [
    ("Product1", 100, "Category1"),
    ("Product2", 200, "Category2"),
    ("Product3", 150, "Category1"),
]
columns = ["ProductName", "Price", "Category"]

df = spark.createDataFrame(data, columns)
df.display()

# Define transformation function
def transform_function(df, discount_percentage):
    return df.withColumn("DiscountedPrice", col("Price") * (1 - discount_percentage

# Parameterized discount percentage
percentage = 20
```

Changing percentage to 10 from 20.

Here in 1st approach, we are calling transform by passing parameter of user defined function transform_function and parameter which is percentage.

```
# Parameterized discount percentage
percentage = 10

# Apply transform function using transform method
transformed_df = df.transform( transform_function,percentage)
#transformed_df = df.transform( transform_function,discount_p
# transformed_df = df.transform(lambda df: transform_function

transformed_df.display()
```

In below screenshot, we can see original dataframe and transformed one.

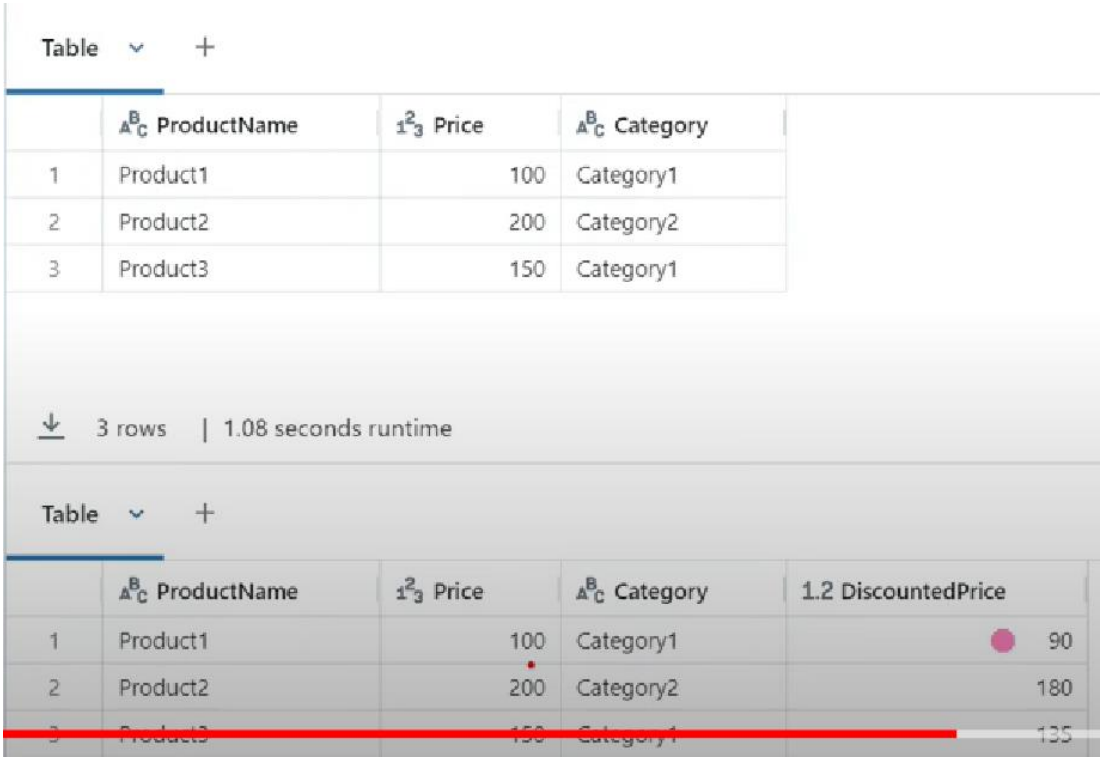


Table +

	ProductName	Price	Category
1	Product1	100	Category1
2	Product2	200	Category2
3	Product3	150	Category1

3 rows | 1.08 seconds runtime

Table +

	ProductName	Price	Category	1.2 DiscountedPrice
1	Product1	100	Category1	90
2	Product2	200	Category2	180
3	Product3	150	Category1	135

--

In 2nd approach, we are going to give variable name as well that is discount_percentage input parameter for our module and that is equal to certain value percentage value and it will provide same output as above.

```python
# Define transformation function
def transform_function(df, discount_percentage):
    return df.withColumn("DiscountedPrice", col("Price") * (1 -

# Parameterized discount percentage
percentage = 10

# Apply transform function using transform method
# transformed_df = df.transform( transform_function,percentage)
transformed_df = df.transform( transform_function,discount_perc
# transformed_df = df.transform(lambda df: transform_function(d

transformed_df.display()
```

Table +

	ProductName	Price	Category
1	Product1	100	Category1
2	Product2	200	Category2
3	Product3	150	Category1

3 rows | 1.16 seconds runtime

Table +

	ProductName	Price	Category	1.2 Discounted...
1	Product1	100	Category1	90
2	Product2	200	Category2	180
3	Product3	150	Category1	135

In this below 3rd approach if we want to apply logic at element level in which means in particular column if we want to apply for each and every value then we can go with lambda .

```
# Parameterized discount percentage
percentage = 10

# Apply transform function using transform method
# transformed_df = df.transform( transform_function,percentage)
# transformed_df = df.transform( transform_function,discount_percentage=percenta
transformed_df = df.transform(lambda df: transform_function(df, percentage))

transformed_df.display()
```

Table +

	ProductName	Price	Category
1	Product1	100	Category1
2	Product2	200	Category2
3	Product3	150	Category1

3 rows | 1.06 seconds runtime

Table +

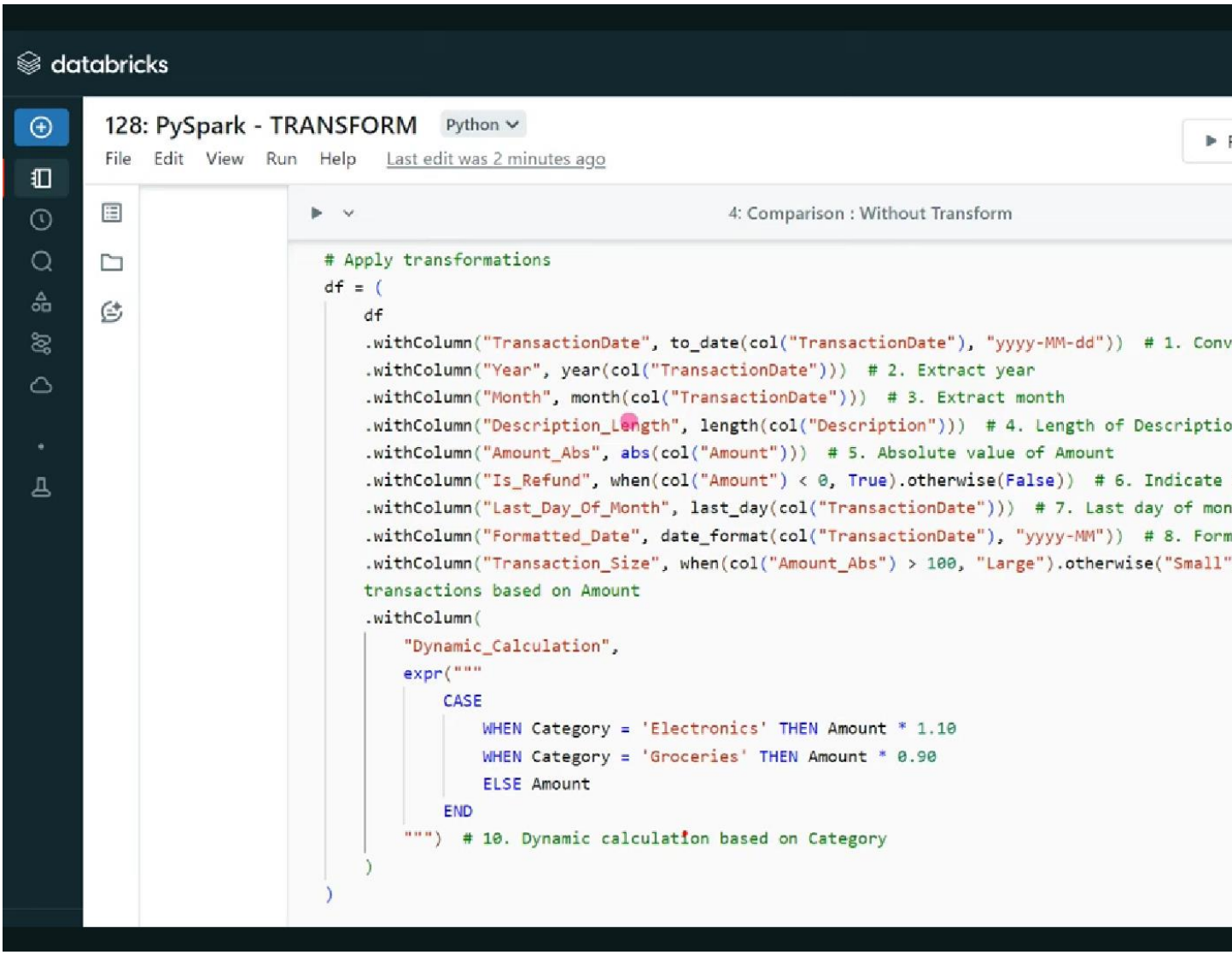
	ProductName	Price	Category	1.2 DiscountedPrice
1	Product1	100	Category1	90
2	Product2	200	Category2	180
3	Product3	150	Category1	135

4: Comparison : Without Transform

```
from pyspark.sql.functions import col, to_date, year, month, length, abs, when, last_day, date_format, e

# Sample data
data = [
    ('TXN001', '2023-02-18', 250.75, 'Electronics', 'Bought a new phone'),
    ('TXN002', '2024-02-25', -50.50, 'Groceries', 'Refunded groceries'),
    ('TXN003', '2019-03-01', 125.00, 'Clothing', 'Purchased new jacket'),
    ('TXN004', '2024-11-28', -10.00, 'Books', 'Refunded book purchase')
]

# Schema definition
schema = 'TransactionID string, TransactionDate string, Amount float, Category string, Description strin

# Create DataFrame
df = spark.createDataFrame(data, schema)
```

Coming to below approach, this is not more readable and it is not good for debugging and troubleshooting.

So, if we look at without transform code, this is not more readable even though its having lesser no of lines but still it is not readable and not recommended. Here we have combined all transformations in one place.



--

So, coming to transform approach,

5: Comparison : With Transform

```
from pyspark.sql.functions import col, to_date, year, month, length, abs, when, last_day, 

# Sample data
data = [
    ('TXN001', '2023-02-18', 250.75, 'Electronics', 'Bought a new phone'),
    ('TXN002', '2024-02-25', -50.50, 'Groceries', 'Refunded groceries'),
    ('TXN003', '2019-03-01', 125.00, 'Clothing', 'Purchased new jacket'),
    ('TXN004', '2024-11-28', -10.00, 'Books', 'Refunded book purchase')
]

# Schema definition
schema = 'TransactionID string, TransactionDate string, Amount float, Category string, Des

# Create DataFrame
df = spark.createDataFrame(data, schema)
```

Here we have splitted logic into separate modules as shown below and it is more readable.

```
# Transformation functions
def convert_to_date(df):
    return df.withColumn("TransactionDate", to_date(col("TransactionDate"), "yyyy-MM-dd"))

def extract_year(df):
    return df.withColumn("Year", year(col("TransactionDate")))

def extract_month(df):
    return df.withColumn("Month", month(col("TransactionDate")))

def description_length(df):
    return df.withColumn("Description_Length", length(col("Description")))

def absolute_amount(df):
    return df.withColumn("Amount_Abs", abs(col("Amount")))

def indicate_refund(df):
    return df.withColumn("Is_Refund", when(col("Amount") < 0, True).otherwise(False))

def last_day_of_month(df):
    return df.withColumn("Last_Day_Of_Month", last_day(col("TransactionDate")))
```

```python
def formatted_date(df):
    return df.withColumn("Formatted_Date", date_format(col("TransactionDate"), "yyyy-MM

def categorize_transaction(df):
    return df.withColumn("Transaction_Size", when(col("Amount_Abs") > 100, "Large").oth

def dynamic_calculation(df):
    return df.withColumn(
        "Dynamic_Calculation",
        expr("""
            CASE
                WHEN Category = 'Electronics' THEN Amount * 1.10
                WHEN Category = 'Groceries' THEN Amount * 0.90
                ELSE Amount
            END
        """)
    )
```

In case if we want to make changes to any particular logic, we don't need to touch main data frame. Directly we can change only in that particular module. But in the previous approach in case if we want to make changes in between logic of (without transform code), its not good approach.

```
# Apply transformations using the transform function
df = (
    df.transform(convert_to_date)
      .transform(extract_year)
      .transform(extract_month)
      .transform(description_length)
      .transform(absolute_amount)
      .transform(indicate_refund)
      .transform(last_day_of_month)
      .transform(formatted_date)
      .transform(categorize_transaction)
      .transform(dynamic_calculation)
)

# Show the results
df.display()
```

So using transform function, this is how we are chaining together all our custom transformations.

So, if we look at without transform code, this is not more readable even though its having lesser no of lines but still it is not readable and not recommended. Here we have combined all transformations in one place.

In real time , we can see actual benefit of using transform,

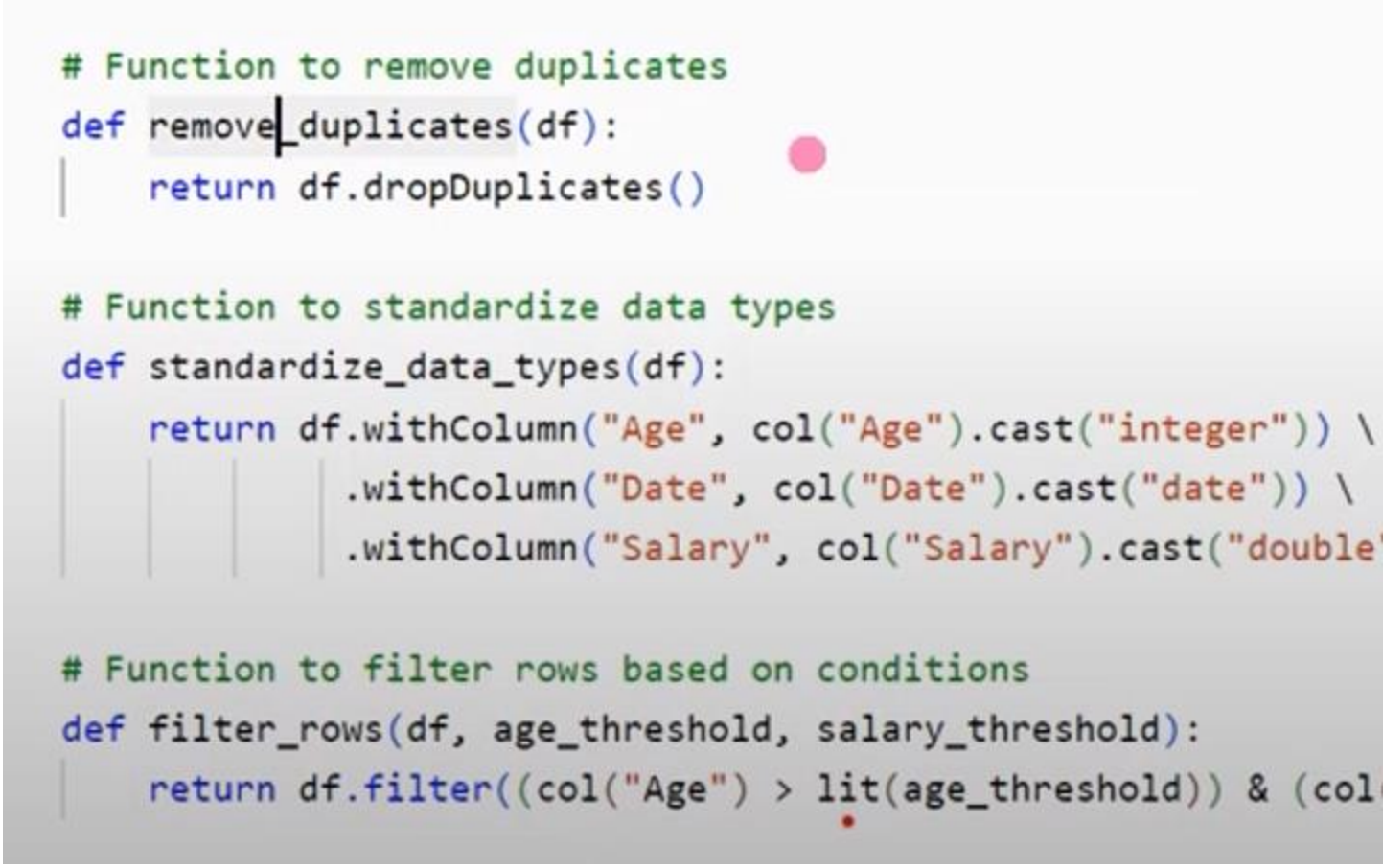
7: Data Cleansing Use Case

```
from pyspark.sql.functions import col, lit, when, avg, sum as _sum

# Sample DataFrame with complex data
data = [
    ("Alice", 34, "2023-06-01", 3000.0),
    ("Bob", 45, "2023-06-02", None),
    ("Cathy", None, "2023-06-03", 2500.0),
    ("Alice", 34, "2023-06-01", 3000.0),
    (None, 45, None, 4000.0)
]
df = spark.createDataFrame(data, ["Name", "Age", "Date", "Salary"])
```

Here if name is missing, we need to populate with unknown. Here we are filling all the values with default values as shown below. Later populating those missing values with the help of handle_nulls function.

```
# Define transformation functions

# Fill values for null handling
name_fill = "Unknown"
age_fill = 0
date_fill = "1900-01-01"
salary_fill = 0.0
age_threshold = 30
salary_threshold = 2000
bonus_percentage = 0.10

# Function to handle null values
def handle_nulls(df, name_fill, age_fill, date_fill, salary_fill):
    return df.fillna({
        'Name': name_fill,
        'Age': age_fill,
        'Date': date_fill,
        'Salary': salary_fill
    })
```

In one function we are standardizing data types, in another function we are filtering rows, in another function removing duplicates.



```python
# Function to remove duplicates
def remove_duplicates(df):
    return df.dropDuplicates()

# Function to standardize data types
def standardize_data_types(df):
    return df.withColumn("Age", col("Age").cast("integer")) \
             .withColumn("Date", col("Date").cast("date")) \
             .withColumn("Salary", col("Salary").cast("double"

# Function to filter rows based on conditions
def filter_rows(df, age_threshold, salary_threshold):
    return df.filter((col("Age") > lit(age_threshold)) & (col
```

In another logic we are performing bonus, and finally we are calling them in our original transformation so wherever needed applied lambda function . in certain cases applying at data frame level. Removing duplicates is not on particular column level or value level so we don’t use lambda

```python
# Function to add a new calculated column 'Bonus'
def add_bonus_column(df, bonus_percentage):
    return df.withColumn("Bonus", col("Salary") * lit(bonus_percentage))

# Function to perform group-by and aggregate results
def group_and_aggregate(df):
    return df.groupBy("Name").agg(
        avg("Age").alias("Average_Age"),
        _sum("Salary").alias("Total_Salary"),
        _sum("Bonus").alias("Total_Bonus")
    )

# Apply transformations using transform function with parameters


df_transformed = (df.transform(lambda df: handle_nulls(df, name_fill, age_fill, date_fill, salary_f
Handle null values
                  .transform(remove_duplicates)  # Remove duplicates
                  .transform(standardize_data_types)  # Standardize data types
                  .transform(lambda df: filter_rows(df, age_threshold, salary_threshold))  # Filte
                  .transform(lambda df: add_bonus_column(df, bonus_percentage))  # Add bonus colum
                  .transform(group_and_aggregate))  # Group and aggregate
```

```python
# Show final DataFrame
df_transformed.display()
```

▸ (3) Spark Jobs

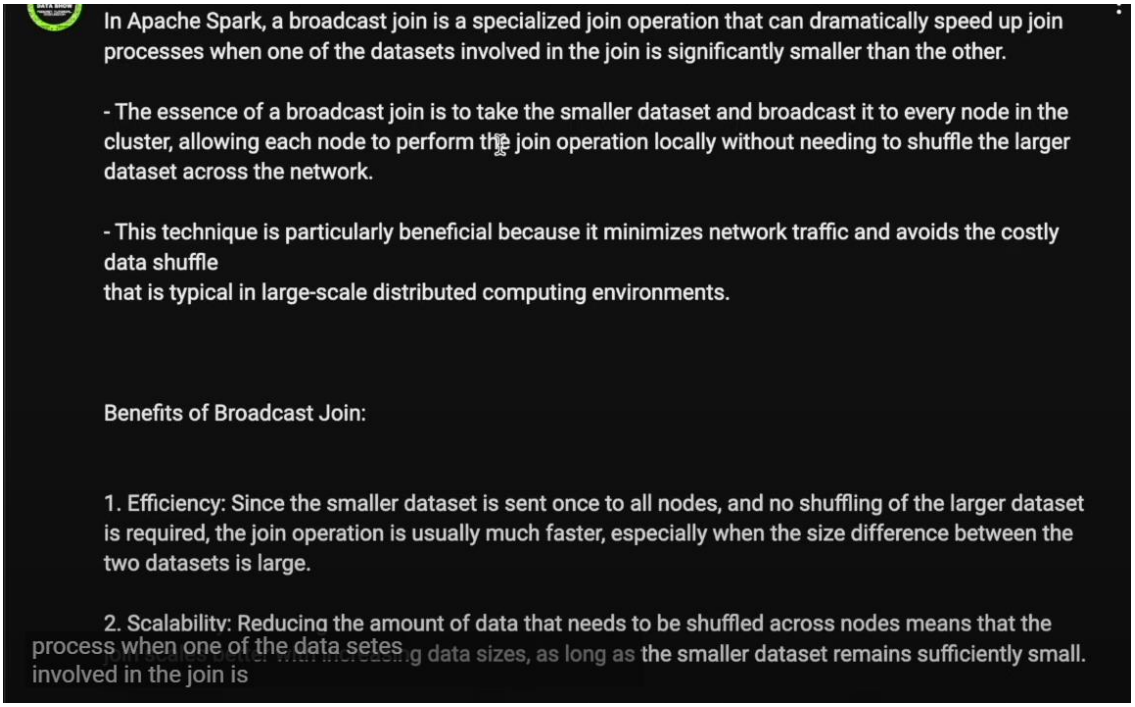
▸ df: pyspark.sql.dataframe.DataFrame = [Name: string, Age: long ... 2 more fields]

▸ df_transformed: pyspark.sql.dataframe.DataFrame = [Name: string, Average_Age: double ... 2 mo

Table ⌄ +

	Name	1.2 Average_Age	1.2 Total_Salary	1.2 Total_Bonus
1	Unknown	45	4000	400
2	Alice	34	3000	300

BROAD CAST JOIN : Whenever join happened between 2 data frames , where one data frame is of bigger data frame and other is of smaller data frame in join, shuffle will happen. Whenever data shuffle happens, it creates a new stages . whenever we are calling an action, spark job is created. From that job multiple stages will be created based on wider transformation mostly because that will involve shuffle and it will shuffle data from multiple executor. Let's suppose we are having 5 executors . so, on 5 executors our data frame data like partitions of data will be existing on 5 different executor. For applying join , every data has to move from their partition , not every partition like most of the data has to shuffle then only they will able to apply join . In broadcast join, we can take smaller data frame and instead of creating partition and keeping the data at different executor , we can take the whole data frame and then copy the data on every executor . suppose we are having 5 executor 1, 2,3,4,5 . so, take the small data frame , copy it on every executor and larger data frame partitions will be already there in every executor then this will actually reduce the shuffle and instead of wider transformation mostly it will convert into narrow transformation because during join we don't need to shuffle our data.

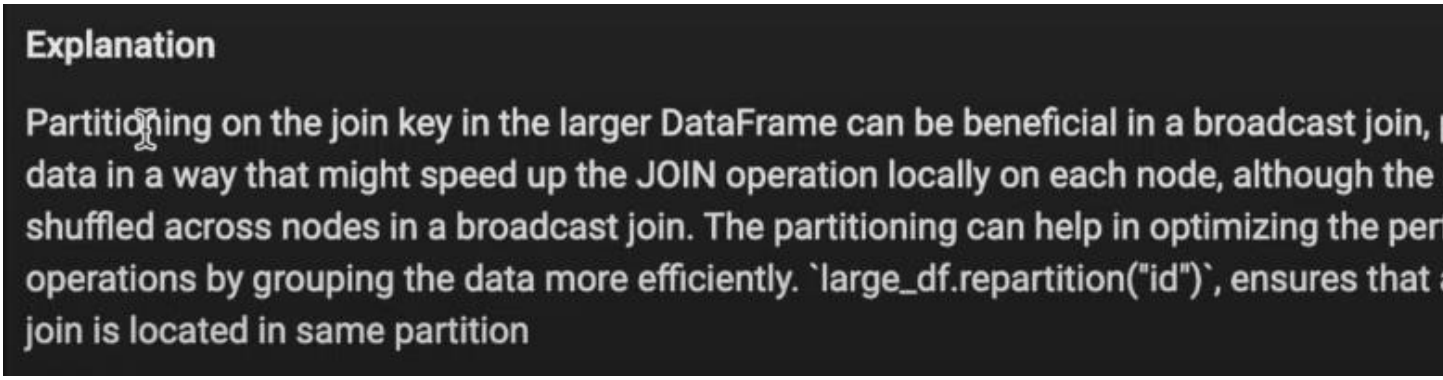


Spark will decide to use broadcast join automatically sometimes. Spark is very very smart. There is one variable called autoBroadcastJoinThreshold . Using that it will detect if the one data frame is smaller or not with compared to another data frame. If

it is smaller like below that limit , it will simply broadcast the smaller data frame . Broadcast join is a type of Narrow transformation in apache spark.

Before applying broadcast join, we are applying repartition on large data frame. This is very good strategy . In production real time scenario, before applying broadcast join, we will repartition the larger data frame so that it makes a shuffle and most of the similar type of partition of data exist on the single executors and this will actually reduce more shuffle .

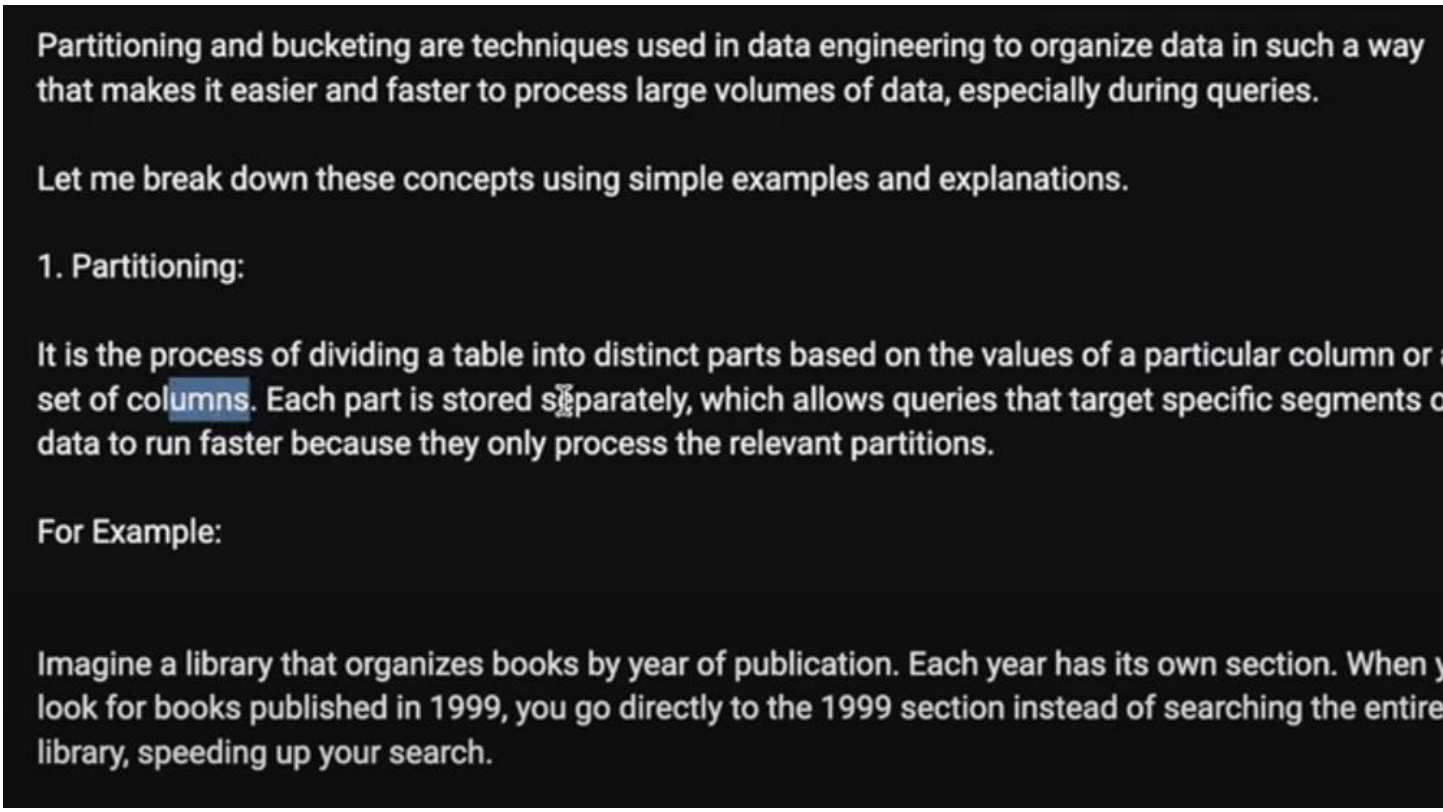
What is the primary benefit of repartitioning the larger data frame on the join key before a broadcast join ? All necessary data for the join is located in the same partition, improving JOIN.

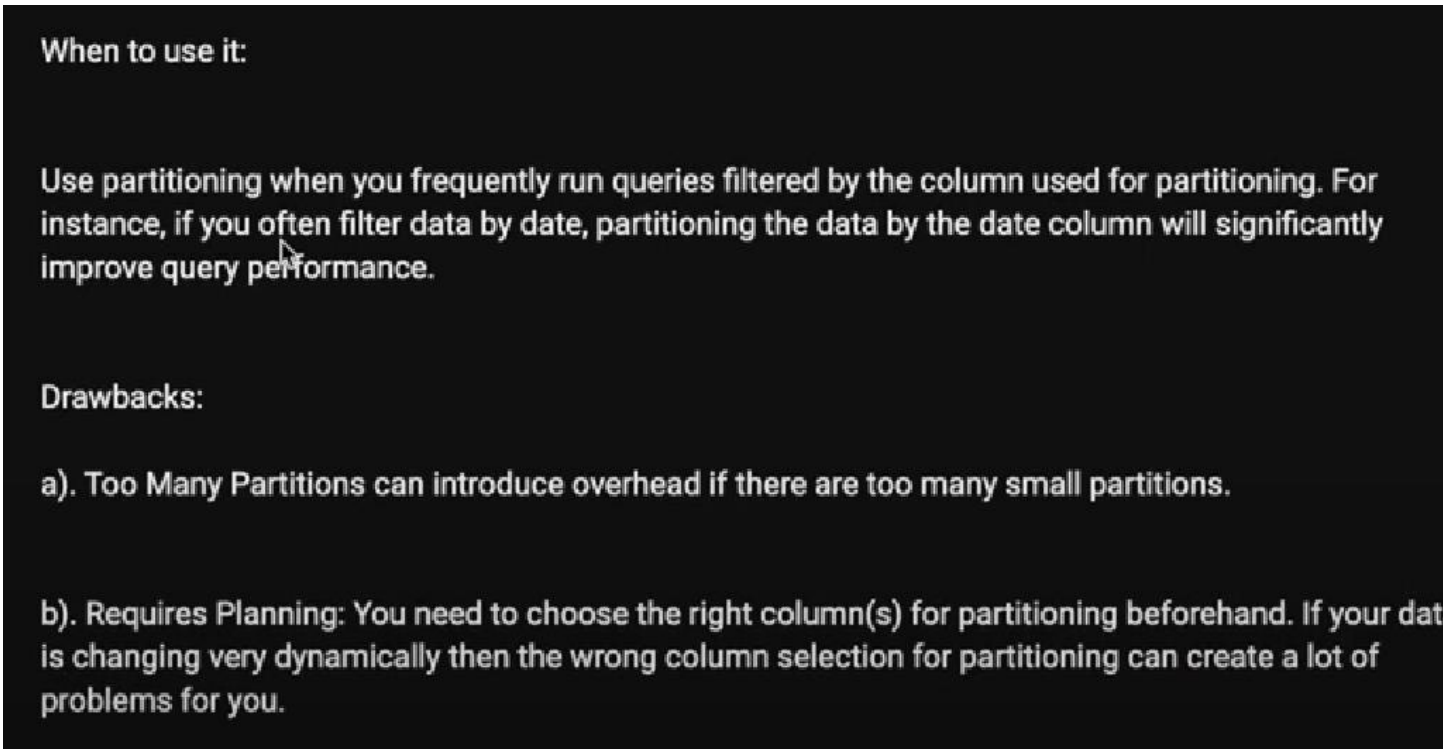


Have you done some optimization in code ? Instead of directly applying broadcast join, sometimes we applied repartitions on larger data frame and then had done broadcast join and this helped us a lot to optimize my code because it reduces the shuffle. It converted from wide transformation to narrow transformation. There is no need of creating additional stage. Whenever action is called, job is created and whenever any shuffling is required then wider transformation is done but broadcast join will reduce it.

PARTITIONING AND BUCKETING

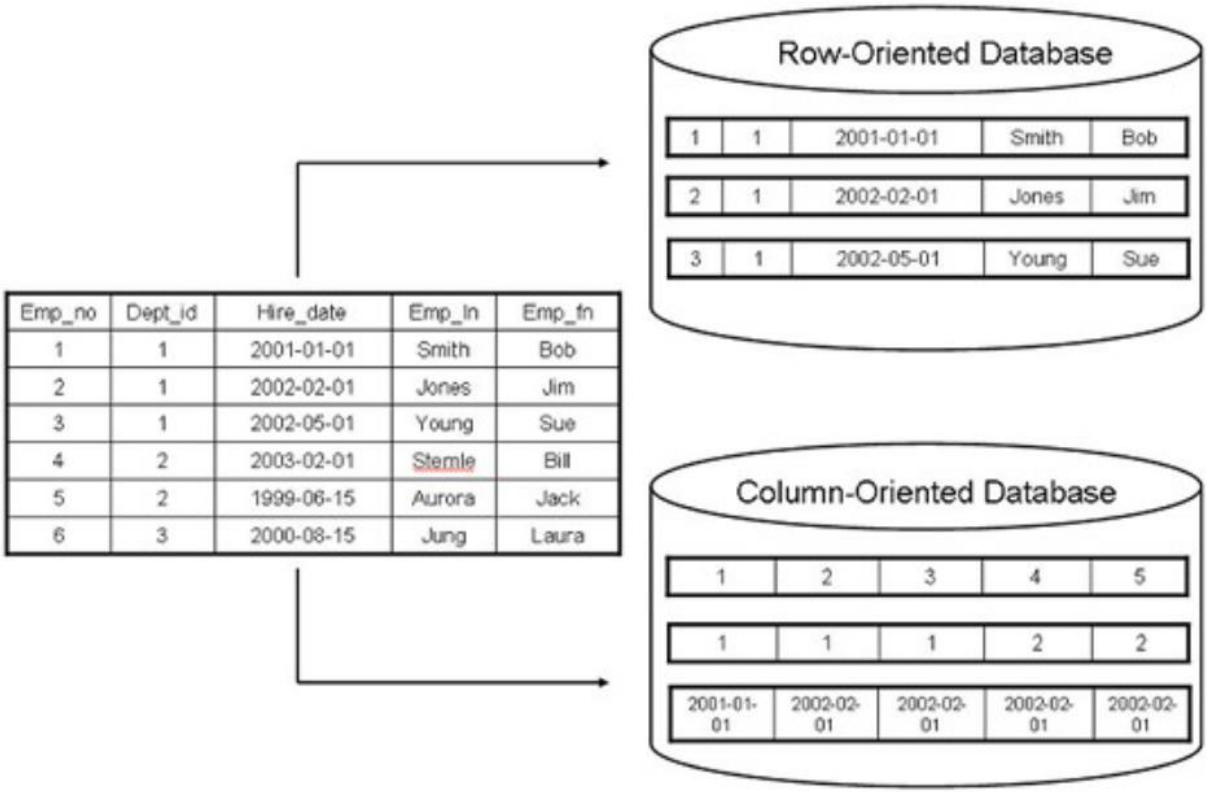
Suppose our data has columns like cust_id, custname and location. If we keep every file in single folder it will be really hard to scan. Suppose whenever we are firing the query like select * from table where location= ” “ . If we want to create multiple sub folders , we will create Partition By on location folder and instead of keeping all the files in a single folder , it will create multiple sub folder with the name location = A, location = B , location = C and then it will keep the corresponding files like parquet, csv or orc files in that particular location.





What is the potential downside of over partitioning data in the data warehouse or even while writing to Data Lake Storage like AWS S3 or GCP ?

Over partitioning can result in large number of small partitions which equates to a large number of small files(small file problem). This can degrade the performance of the file system and the overall query performance due to increased overhead in managing multiple small files.



https://www.linkedin.com/pulse/file-formats-big-data-world-part-1-ankur-ranjan/

Apache Parquet is an open-source, column-oriented data file format designed for efficient data storage and retrieval. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk.

Some of the characteristics of this file format are as follows.

- **Column-based format** - files are organized by column, rather than by row, which saves storage space and speeds up analytics queries.
- **Supports complex data types -** It supports advanced nested data structures like Array, Struct etc. We can very nested values in the column and it will work like charm.
- **Very good predicate pushdown filters support -** if you apply a filter on column A to only return records with value V, the predicate pushdown will make parquet read-only blocks that may contain values V.
- **Advance Compression Support** –
- By default, it comes with Snappy compression codec which gives it very much balance between compression ratio and speed.
- So internally Parquet will store these data as bytes instead of Int. It saves storage and space. It uses a complex mechanism to store the right byte size. Apache Parquet uses snappy compression by default when it is used with computing engines like Apache Spark. This ensures the right balance b/w compression ratio and speed.