Spark Optimization

In Databricks we don't have to submit Spark jobs or submit driver and executors, everything is being taken care off by DB cluster.

Even we don't have to create Spark Session, its already running for you. The variable spark is by default ready.

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder.master("local[*]")\
        .appName("AnshLamba")\
        .getOrCreate()
```

Not required

```
spark
```

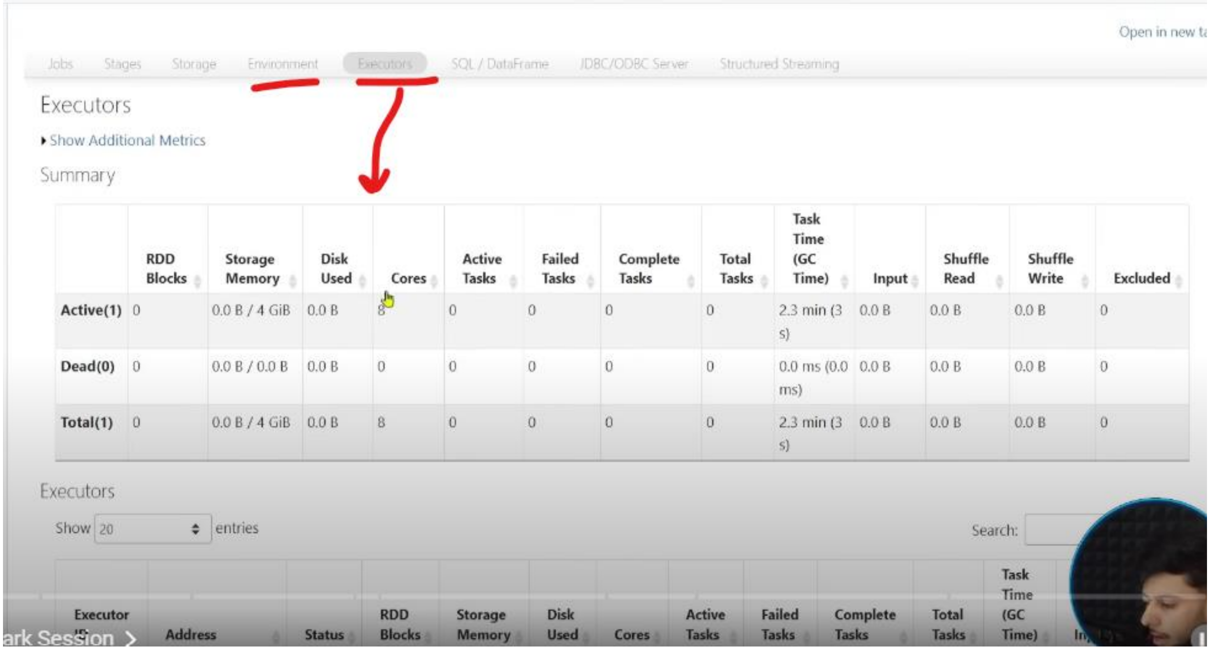
SparkSession - hive

SparkContext

Spark UI

Version
v3.3.2
Master
local[8]
AppName
Databricks Shell

Click on Spark UI

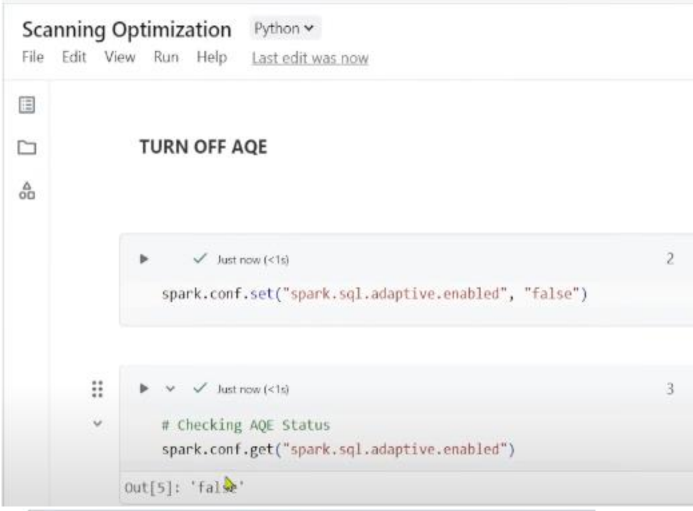


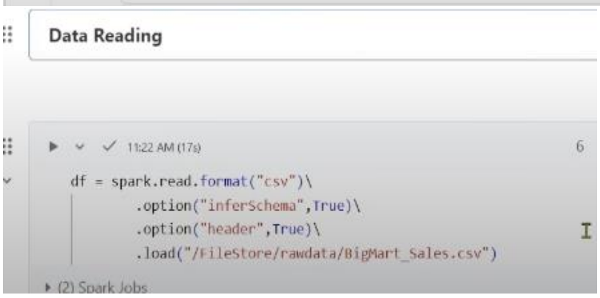
	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(1)	0	0.0 B / 4 GiB	0.0 B	8	0	0	0	0	2.3 min (3 s)	0.0 B	0.0 B	0.0 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(1)	0	0.0 B / 4 GiB	0.0 B	8	0	0	0	0	2.3 min (3 s)	0.0 B	0.0 B	0.0 B	0

You can play with spark session using spark.conf.set() but don't have to create Spark Session in DB. DB removes all the overheads.

1. Scanning Optimisation → Partition Pruning

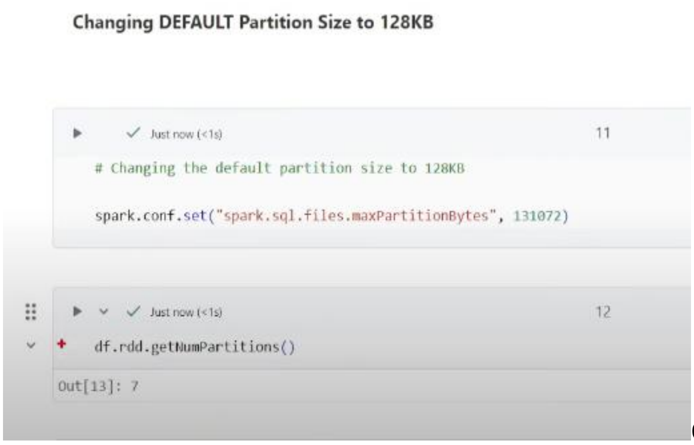
Partition Pruning : Don't have to scan the entire data for a specified condition to avoid reading the entire data, so Partition Pruning helps optimising the partitions.





Spark creates Logical partitions/small chunks on top of data (by default Block size is 128MB)

Since we have peanut amount of data



Our file size is 850kb

Changing the default partition size to 128MB

```
spark.conf.set("spark.sql.files.maxPartitionBytes", 134217728)
```

Repartitioning

```
df = df.repartition(10)
```

df: pyspark.sql.dataframe.DataFrame = [Item_Identifier: string, Item_Weight: double ... 10 more fields]

Repartitioning

```
df = df.repartition(10)
```

df: pyspark.sql.dataframe.DataFrame = [Item_Ide

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

(1) Spark Jobs

Out[7]: 10

Why partitions are required? → to apply parallelism, each cores in executor can perform the parallelism tasks.

Get Partition Info

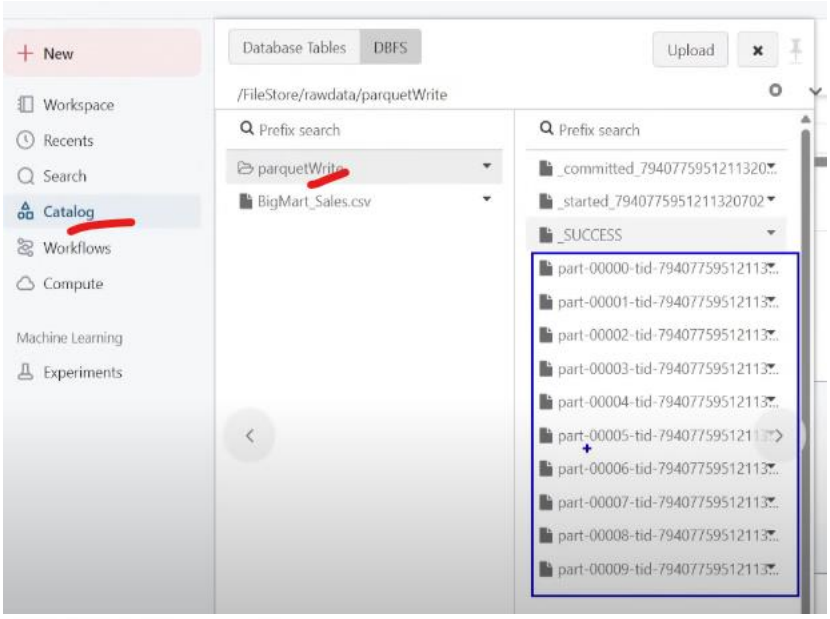
```
df.withColumn("partition_id",spark_partition_id()).display()
```

(2) Spark Jobs

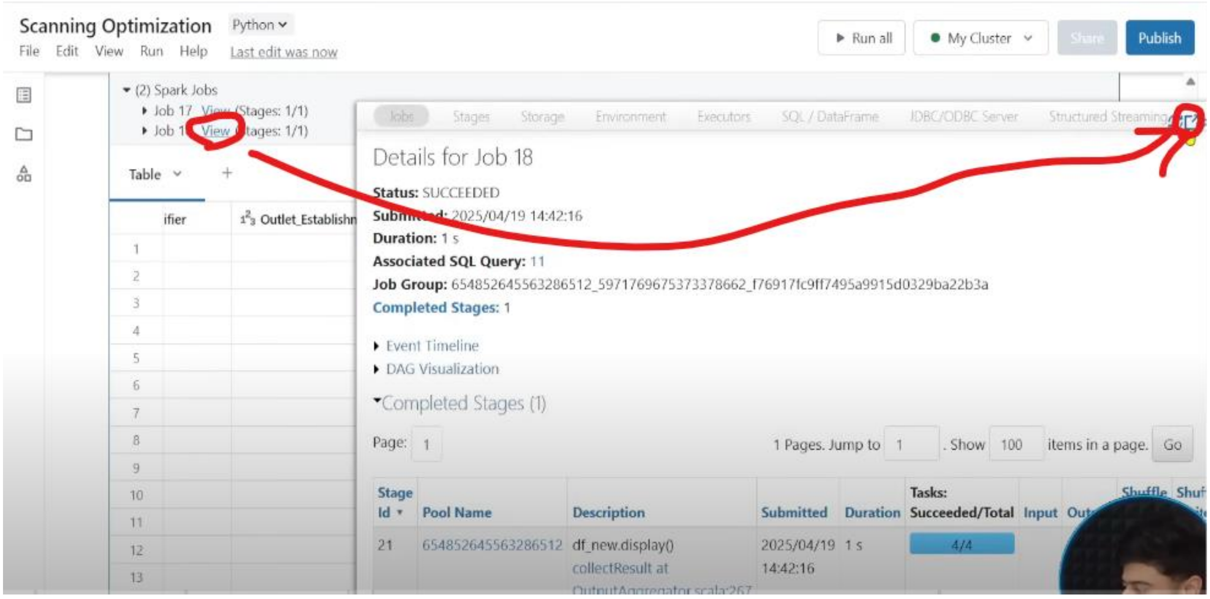
Data Writing

```
df.write.format("parquet")\
    .mode("append")\
    .option("path","/FileStore/rawdata/parquetWrite")\
    .save()
```

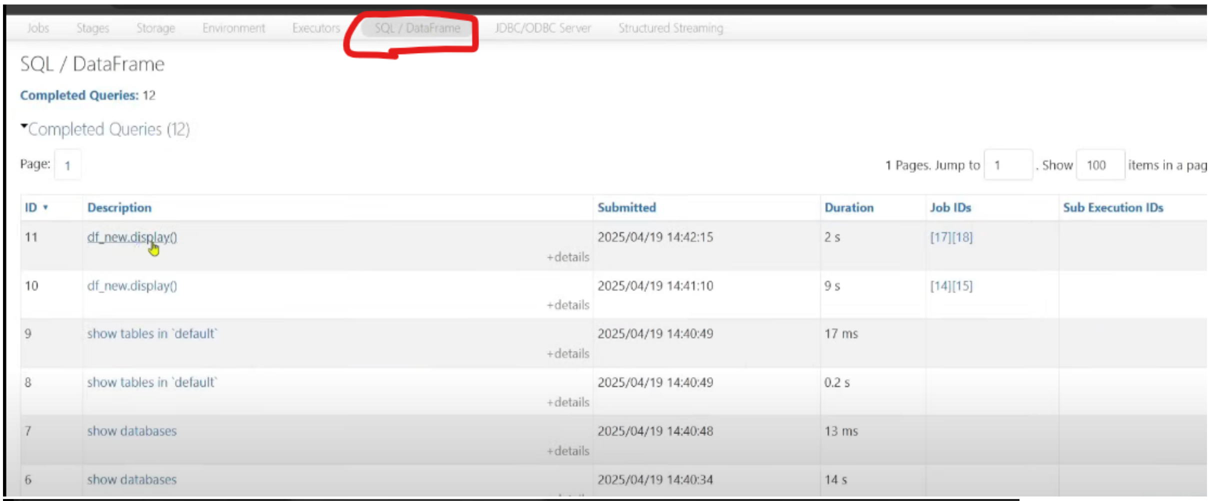
(2) Spark Jobs

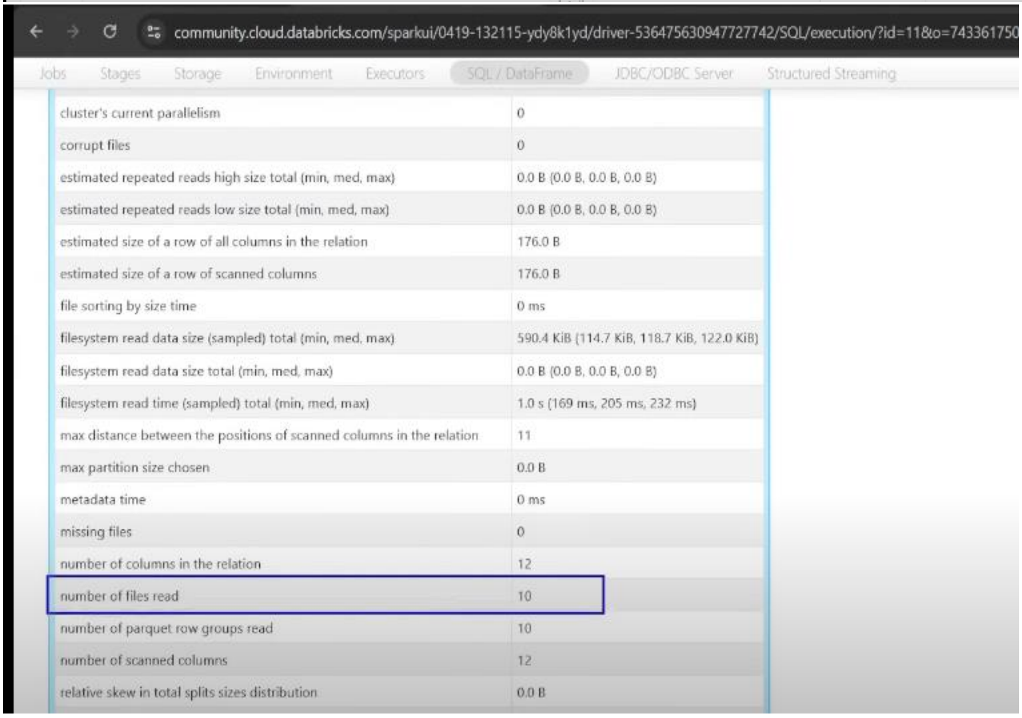


Spark will read all the 10 files as it doesnot know where Tier1 data is present



Click on Jobs → Expand → SQL dataframe

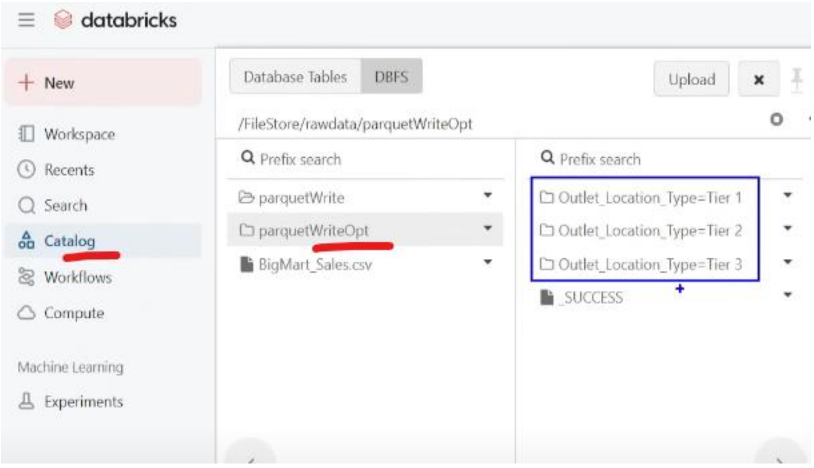




SCANNING OPTIMIZATION

Solution →


```
df.write.format("parquet")\
      .mode("append")\
      .partitionBy("Outlet_Location_Type")\
      .option("path","/FileStore/rawdata/parquetWriteOpt")\
      .save()
```

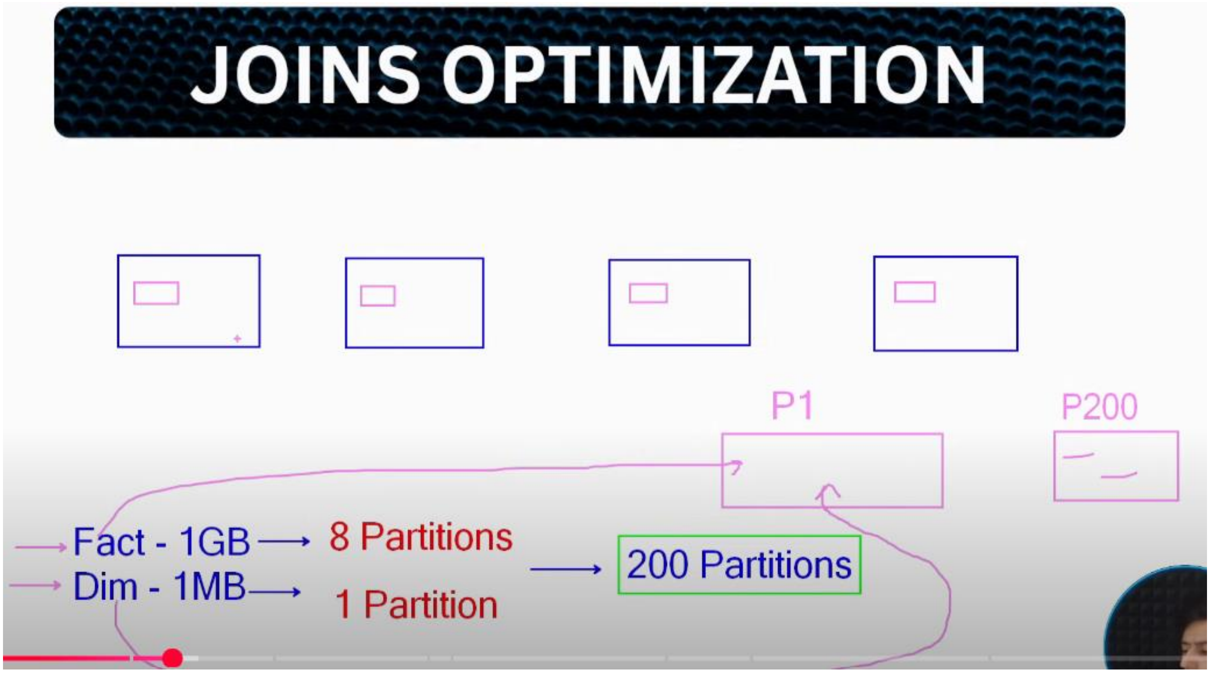


Generally do Partitions on date columns→ best column→ year, month, date

2. Join Optimizations

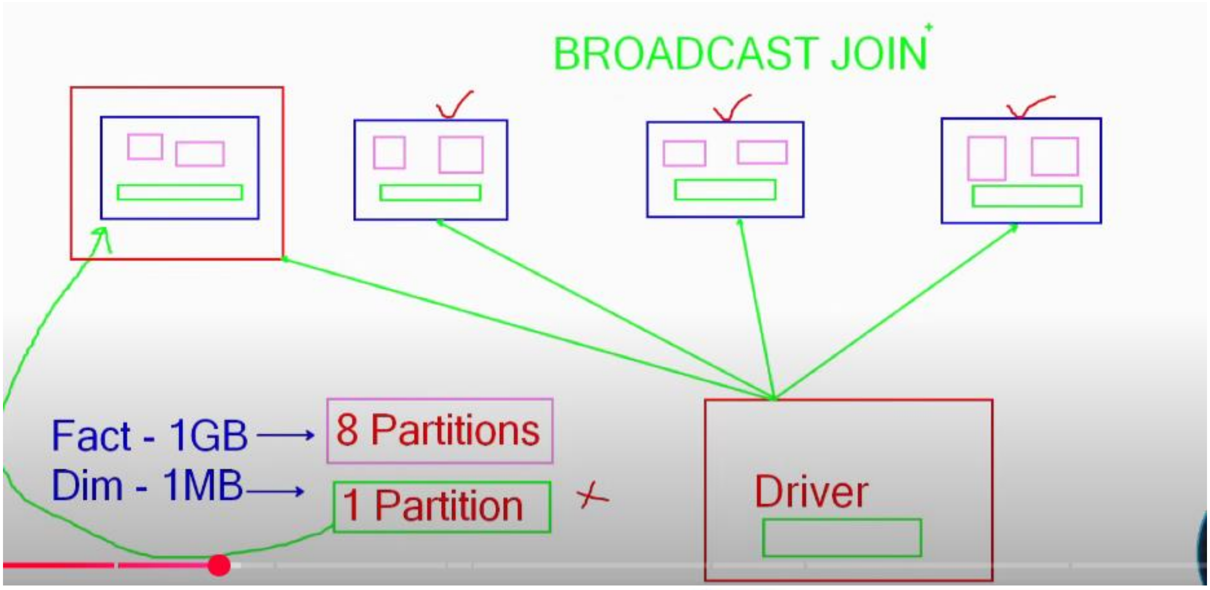
Lets say we have 4 Executors, 1 Fact of size 1GB and 1 Dimension of 1 MB

Join is a wide transformation, if AQE is disabled then Spark creates 200 Partitions , so Partiton1 will be having some data of fact and dimension joined with dimension key, similarly till P200, and all these partitions will be sent to Executors for execution.



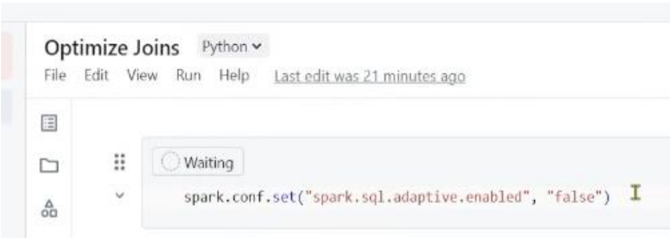
To avoid Shuffling, we will do Broadcast join

Driver will broadcast the smaller table to all the executors



```python
# Big DataFrame
df_transactions = spark.createDataFrame([
    (1, "US", 100),
    (2, "IN", 200),
    (3, "UK", 150),
    (4, "US", 80),
], ["id", "country_code", "amount"])

# Small DataFrame
df_countries = spark.createDataFrame([
    ("US", "United States"),
    ("IN", "India"),
    ("UK", "United Kingdom"),
], ["country_code", "country_name"])
```



Optimize Joins Python

File Edit View Run Help Last edit was 21 minutes ago

Waiting

```python
spark.conf.set("spark.sql.adaptive.enabled", "false")
```

Just now (<1s) 5

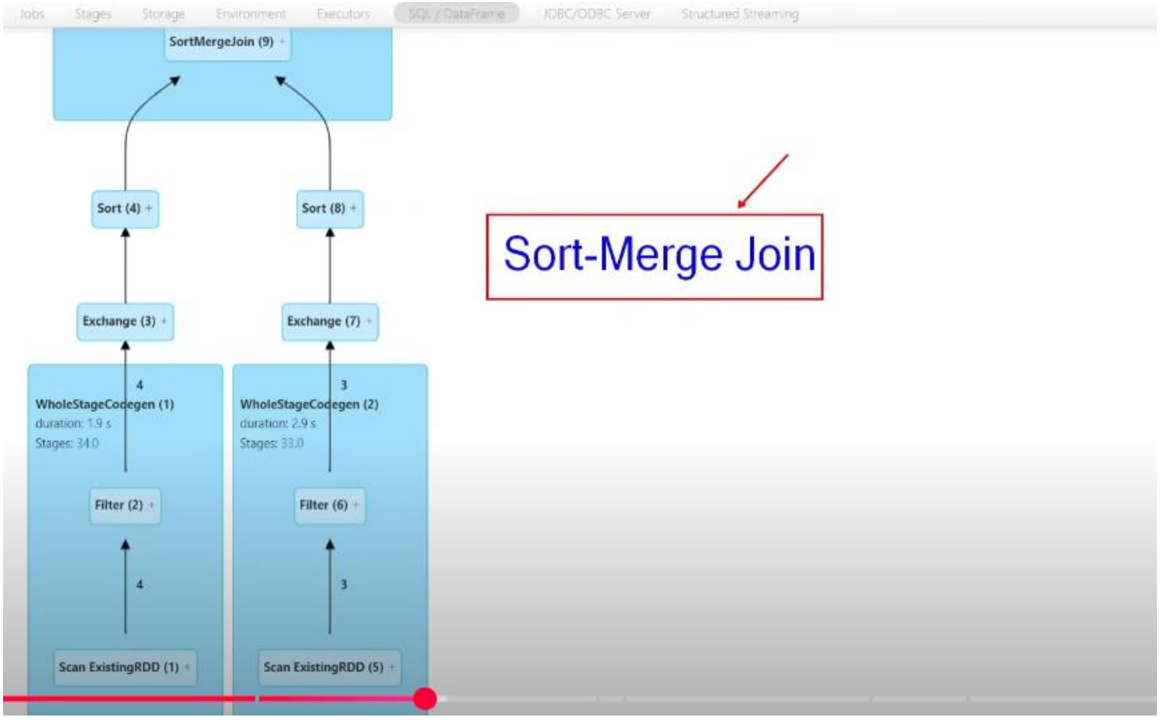
```python
df_join = df_transactions.join(df_countries,df_transactions['country_code']==df_countries['country_code'],"inner")
```

df_join: pyspark.sql.dataframe.DataFrame = [id: long, country_code: string ... 3 more fields]

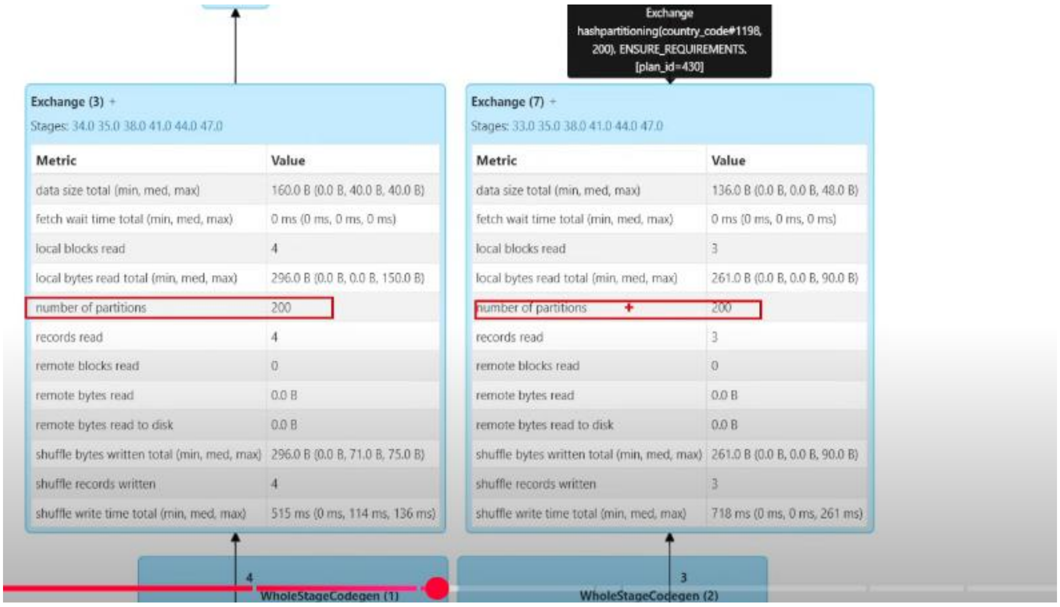
Interrupt 00:01 6

```python
df_join.display()
```

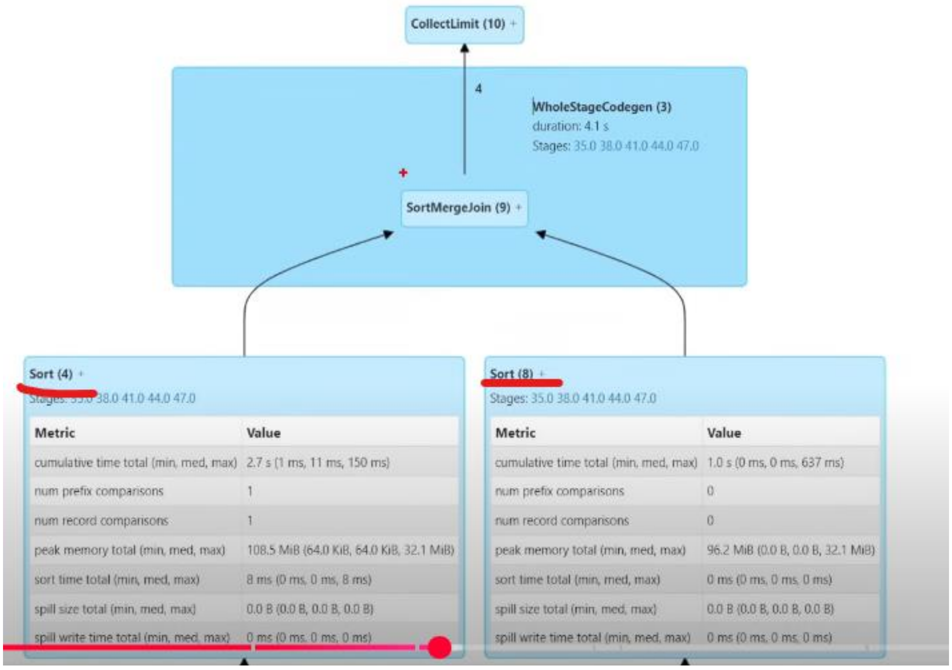
(1) Spark Jobs



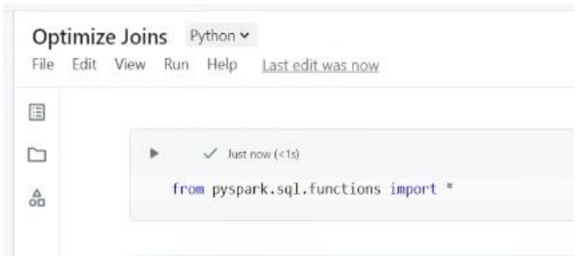
Sort-Merge join has been performed, default join, sorting within 200 partitions & then join.
Filter to remove nulls, Exchange is the step where shuffling happens

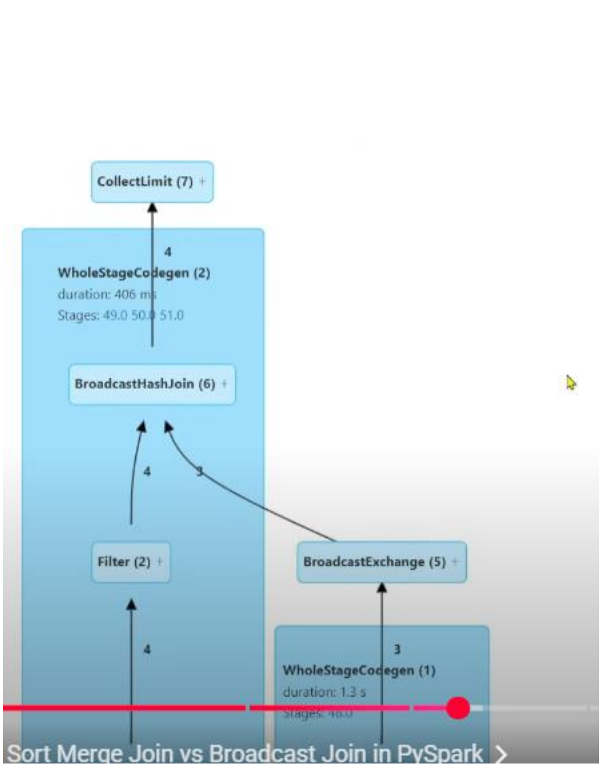


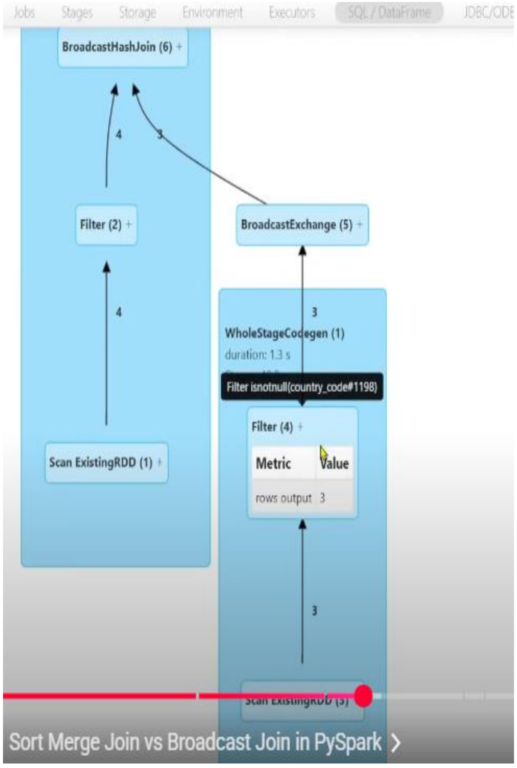
200 partitons for 3-4 records, this is crazy



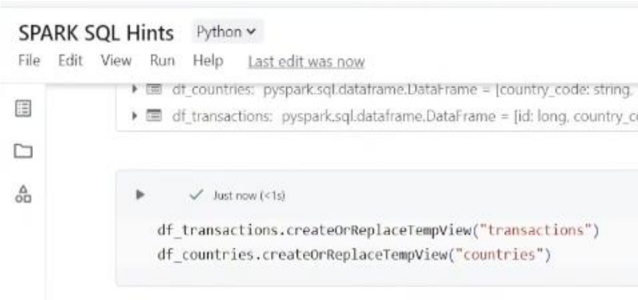
Then it performs sorting and finally sort merge join

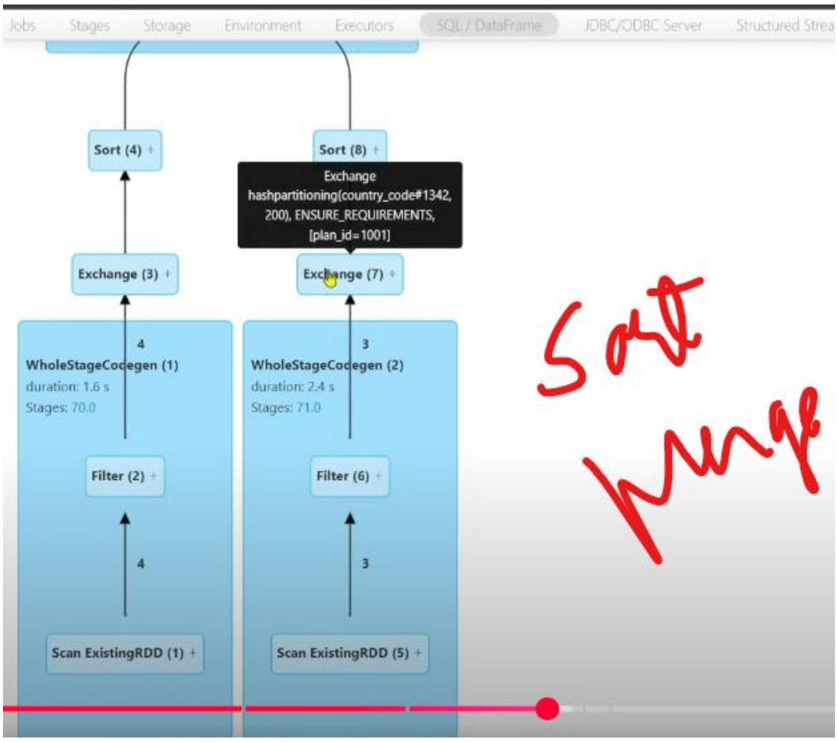


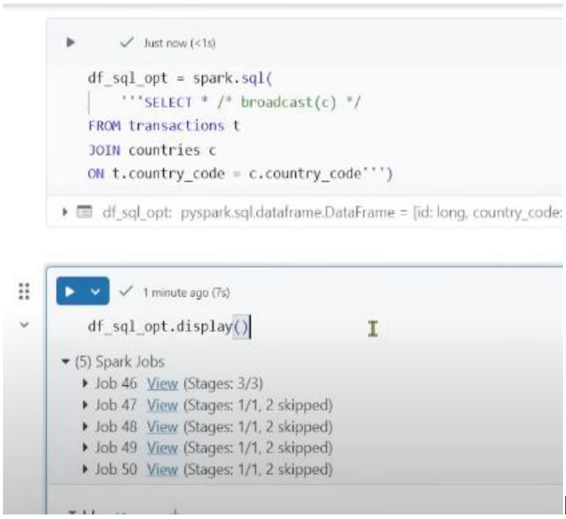




3. SQL Hints (Optimizations on Spark SQL)



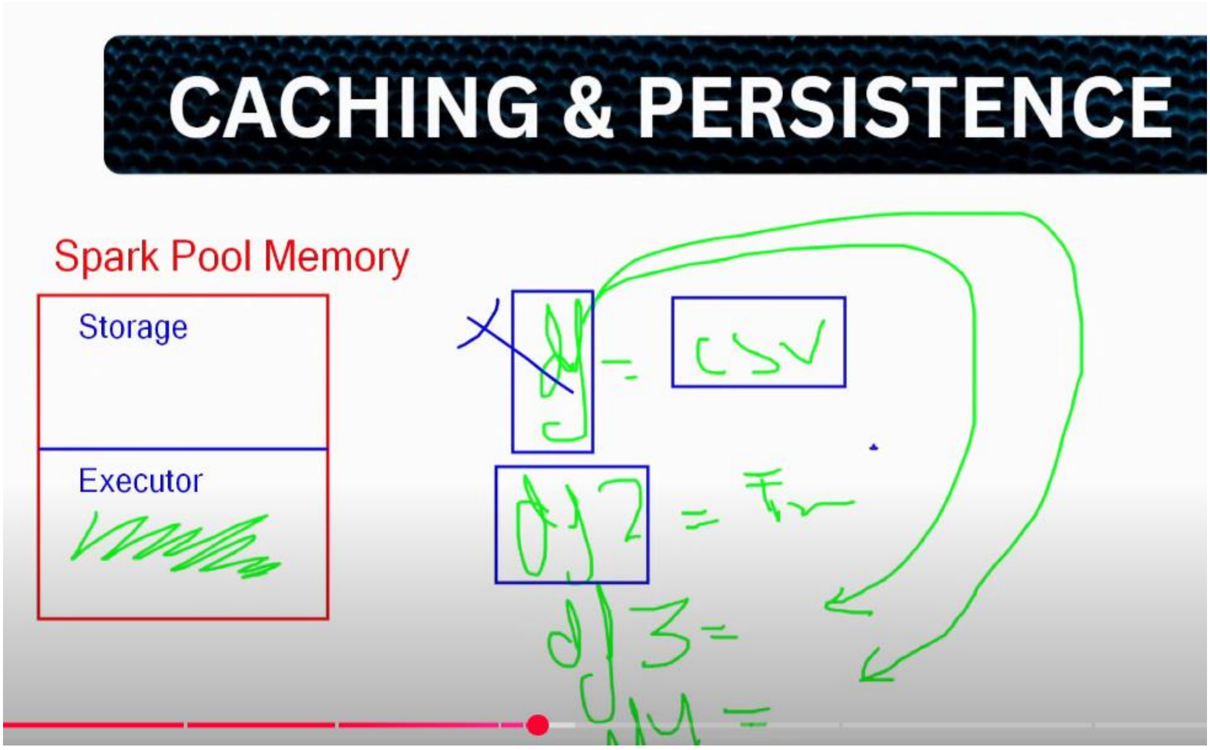




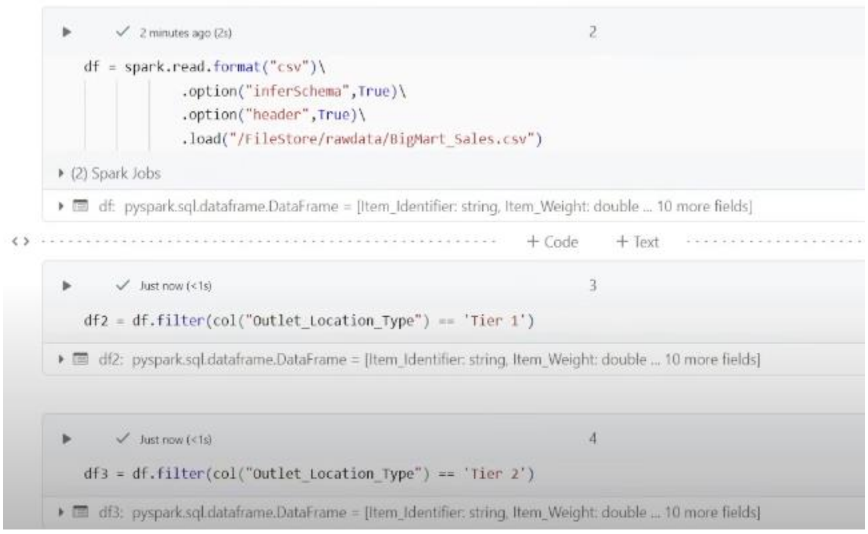
Its just an advice to Spark, not a guarantee, it may consider may not.

4. CACHING & PERSISTENCE

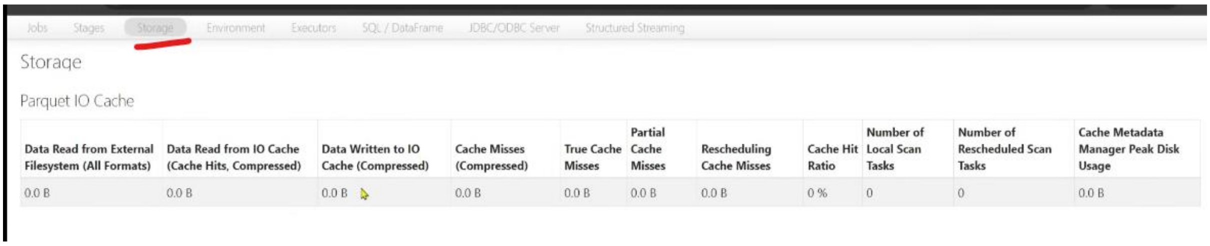
Spark Pool Memory is 60% of overall memory, in which we have Storage memory & Executor memory. Executor memory is used to perform all the transformations, and it's a short lived memory



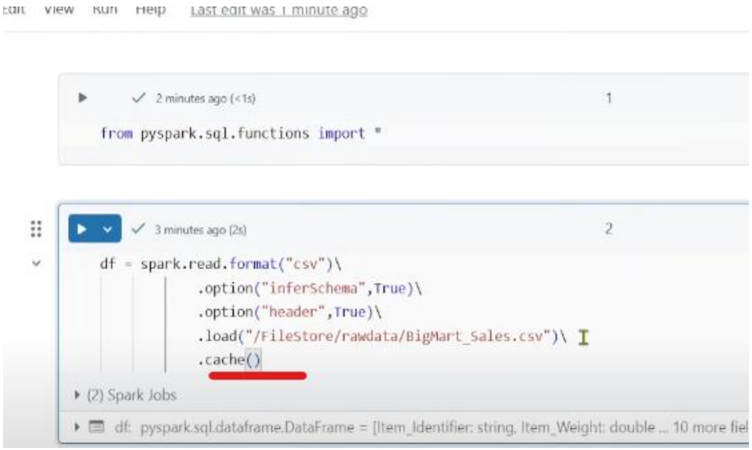
Lets say in df we are reading a csv, in df2 we are doing some transformations, in df3 and df4 we are again using df, so every time it will be recalculated as df has not been saved anywhere, its happening in Executor memory which is a short lived memory. We need to store the data in Long Term memory i.e. Storage memory.

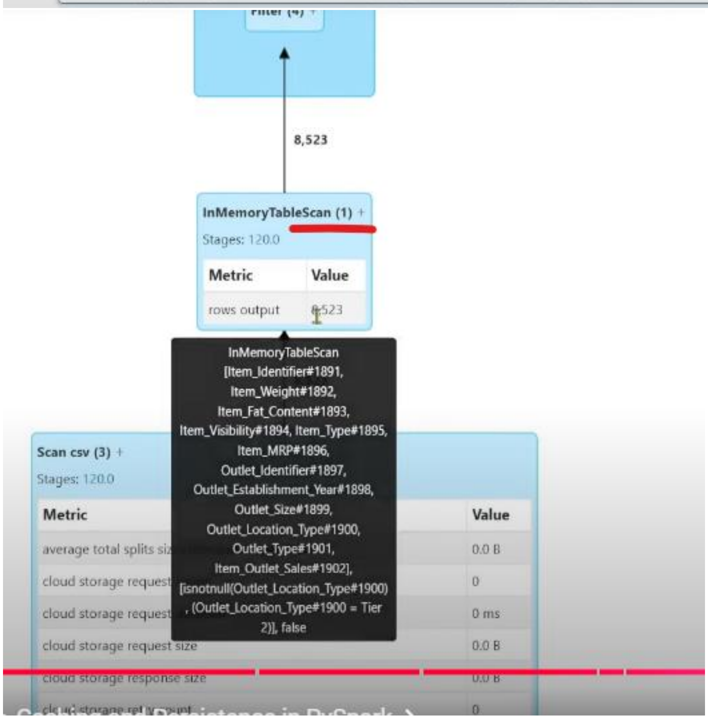


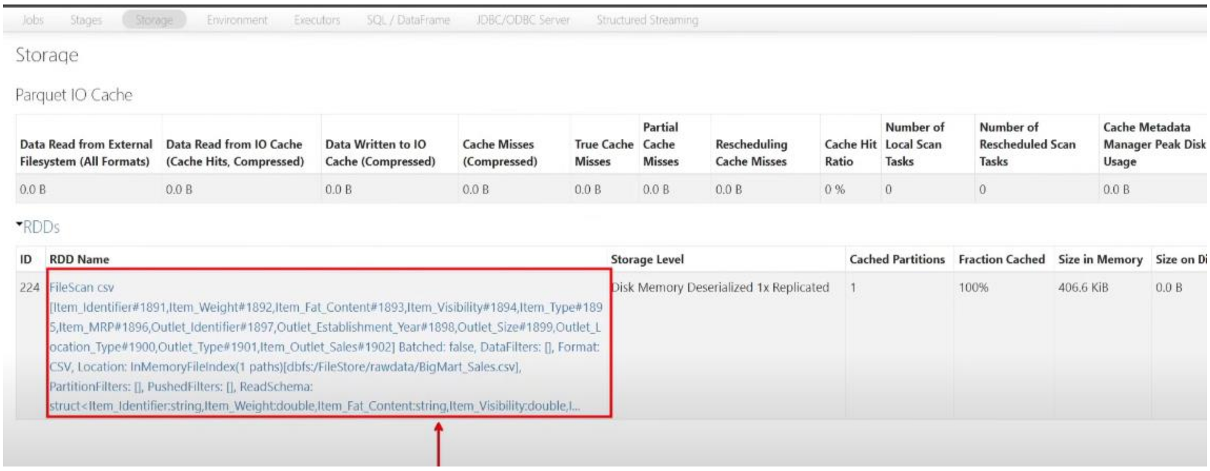
df will be computed multiple times



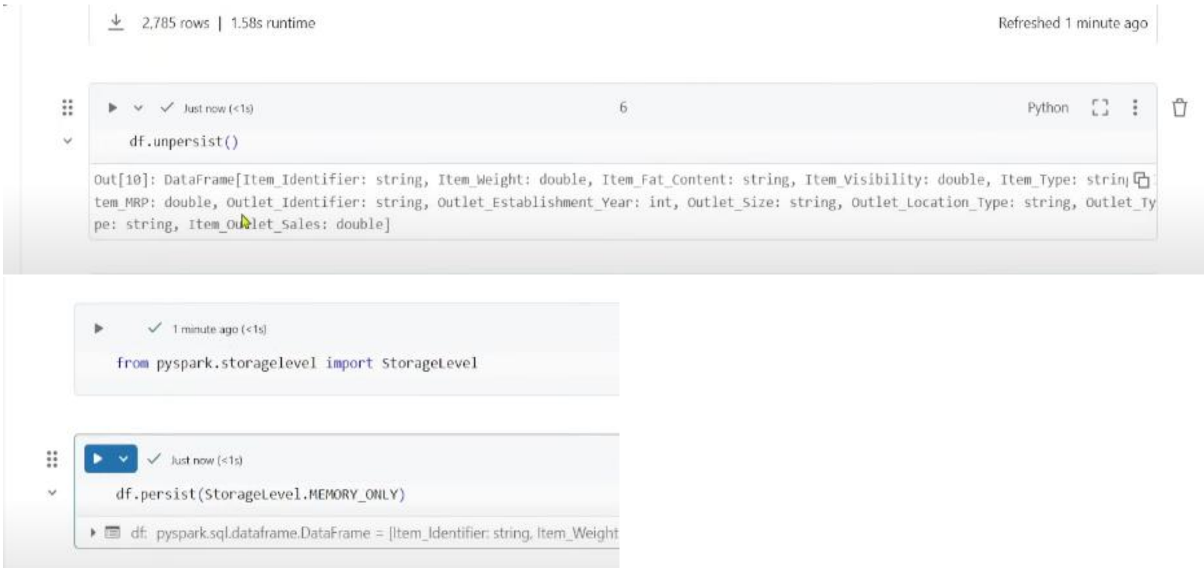
Now let's cache it





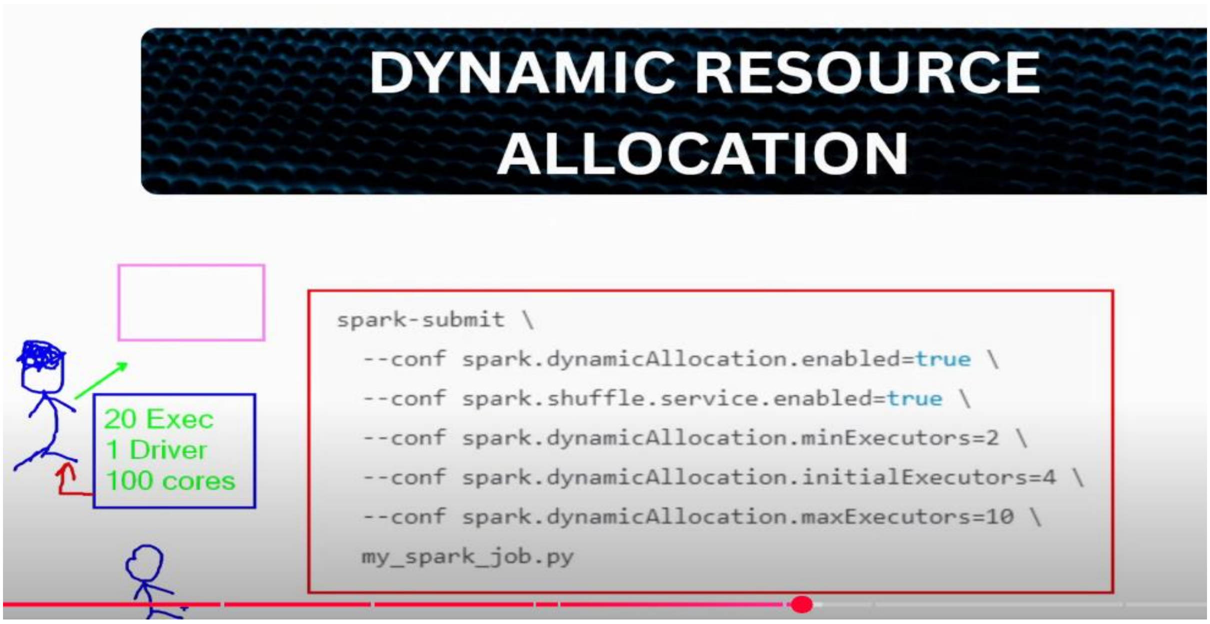


To finally remove the cached data



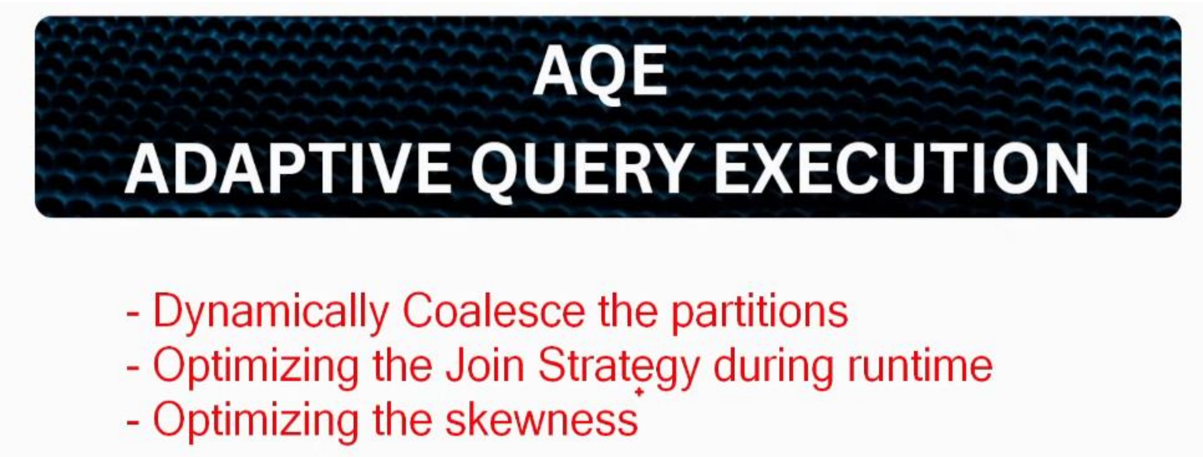
5. DYNAMIC RESOURCE ALLOCATION

Now auto enabled

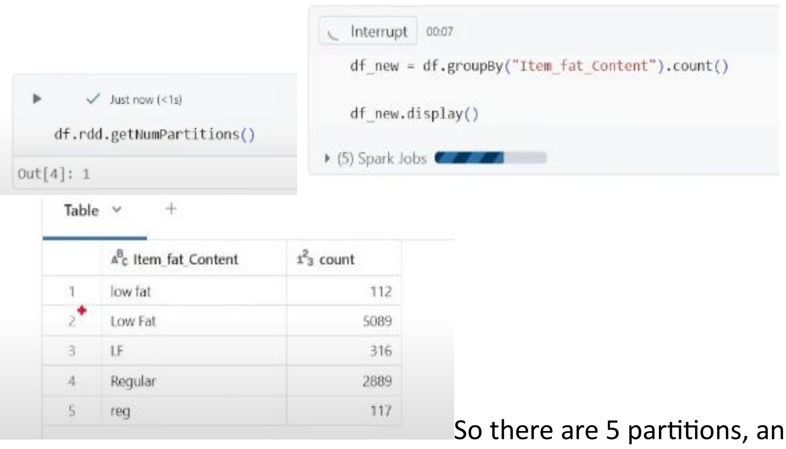


Requesting resources from Cluster Manager, all these resources will be locked with you till you kill the application even if you are not utilising all the resources.
Now there is another developer who cant submit any application because those resources are already in use.
Best solution is **DYNAMIC RESOURCE ALLOCATION,** resources will not be locked, if executors are in idle state, those executors will be released, Its like Spark Pool in Synapse, DB and Fabric.

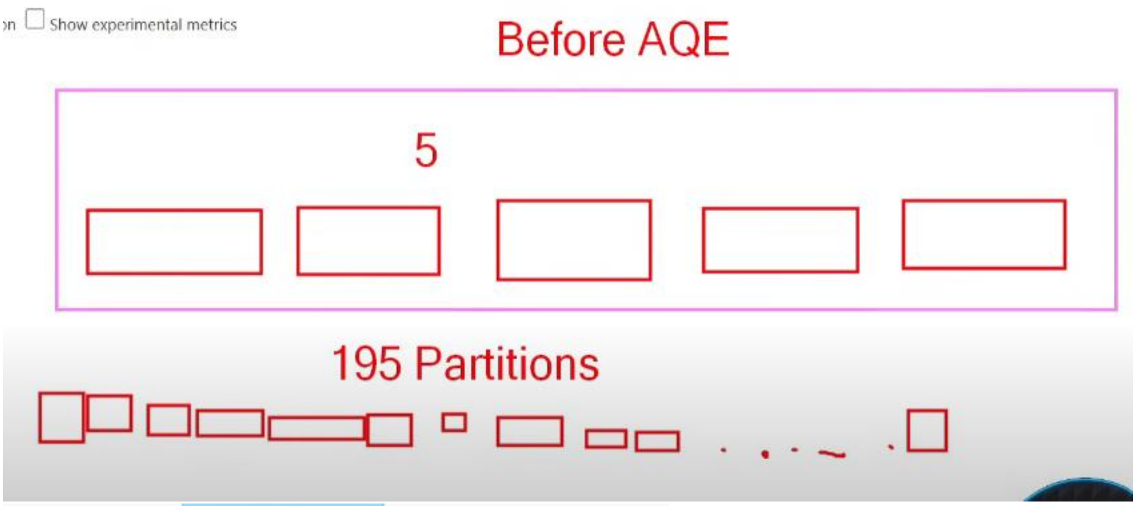
6. AQE (Adaptive Query Execution)

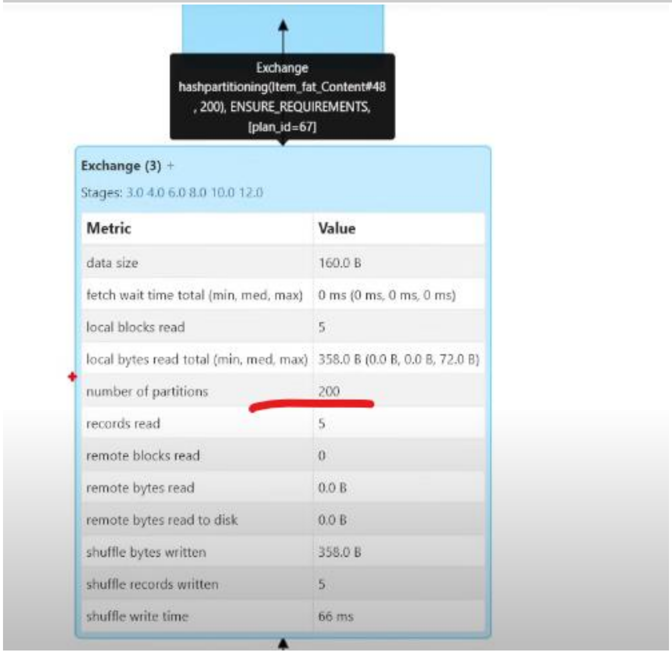


Dynamically coalesce the number of partitions from 200 to required number
Optimizes the Joins whether a Broadcast or a Sort Merge
Optimizes the skewness → when a particular partition is very big then it breaks down into smaller partitions

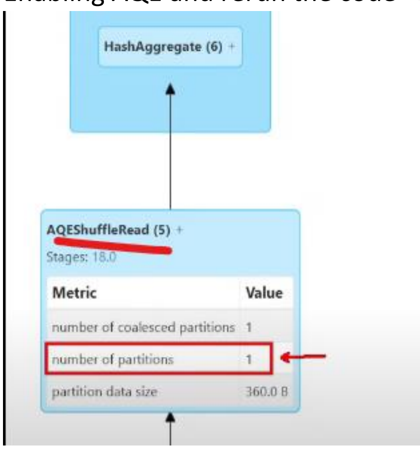


So there are 5 partitions, and rest 195 partitions will be empty, wating all these resources





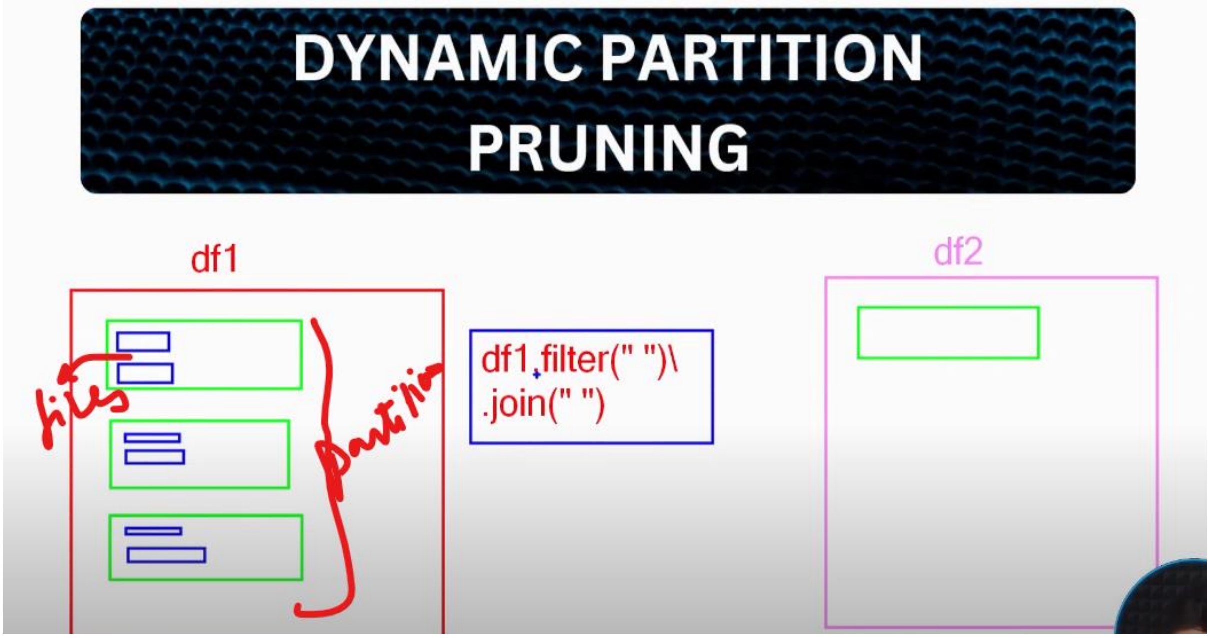
Enabling AQE and rerun the code



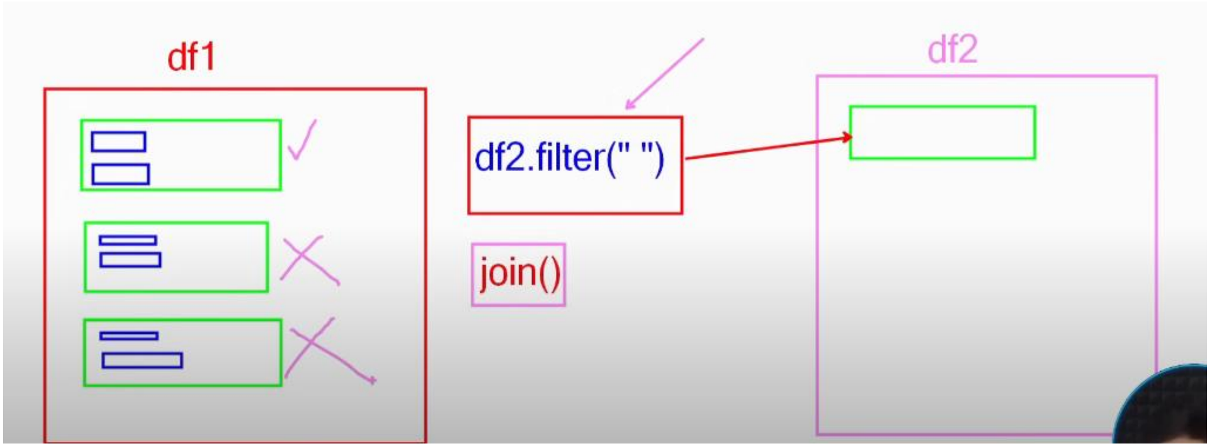
from 200 partitions to 1 partition, AQE firstly removes all the 195 empty partitions and then coalesces the partitions from 5 to 1.

7. DYNAMIC PARTITION PRUNING

Advanced version of Partiton Pruning



Lets say df1 has 3 partitions & df2 has 1, if I want to join df1 and df2 by applying filter on df1 → this is basically Partition Pruning



Now lets say we are joining df1 and df2, by applying filter on df2, so it should read all the partitions of df1 right?But lets say based on the filter condition of df2 the matched records are found only in one partition of df1, then it don't have to read the other partitions, this is Dynamic Partition Pruning. DPP pass the filter of df2 to df1 during runtime, simply broadcast the value of df2 to df1.

Waiting 3

```
df = spark.read.format("csv")\
        .option("header",True)\
        .option("inferSchema",True)\
        .load("/FileStore/rawdata/BigMart_Sales.csv")

df.limit(8).display()
```

Preparing the Partitioned Data

Interrupt 00:01 6

```
df.write.format("parquet")\
        .mode("append")\
        .partitionBy("Outlest_Type")\
        .option("path","/FileStore/rawdata/dpp_partioned")\
        .save()
```

Non Partitioned Data

Just now (2s) 8

```
df.write.format("parquet")\
        .mode("append")\
        .option("path","/FileStore/rawdata/dpp_nonpartioned")\
        .save()
```

(1) Spark Jobs

Dataframes

Just now (2s) 10

```
df1 = spark.read.format("parquet")\
        .load("/FileStore/rawdata/dpp_partioned")
```

(1) Spark Jobs

df1: pyspark.sql.dataframe.DataFrame = [Item_Identifier: string, Item_Weight: double ... 10 more fields]

Waiting 11

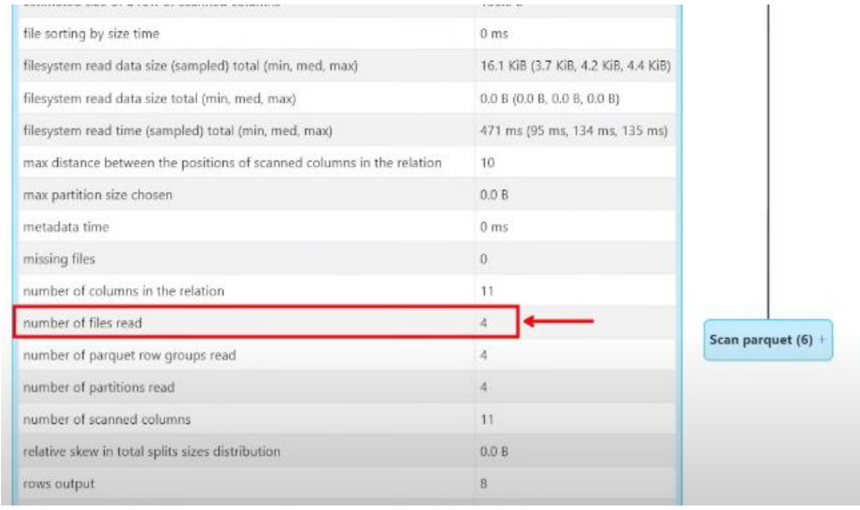
```
df2 = spark.read.format("parquet")\
        .load("/FileStore/rawdata/dpp_nonpartioned")
```

JOINS

Just now (<1s) 13

```
df_join = df1.join(df2.filter(col("Outlet_Type")=="Grocery Store"),df1['Item_Identifier']==df2['Item_Identifier'],"inner")
```

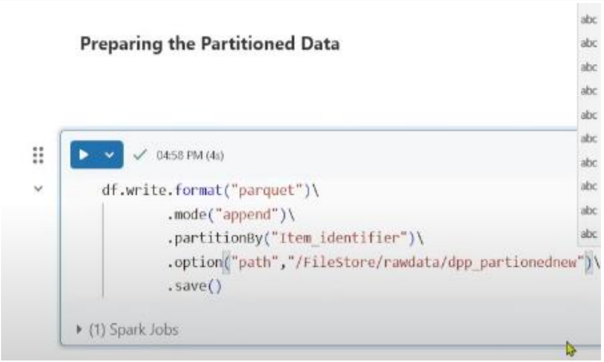
df_join: pyspark.sql.dataframe.DataFrame = [Item_Identifier: string, Item_Weight: double ... 22 more fields]



file sorting by size time	0 ms
filesystem read data size (sampled) total (min, med, max)	16.1 KiB (3.7 KiB, 4.2 KiB, 4.4 KiB)
filesystem read data size total (min, med, max)	0.0 B (0.0 B, 0.0 B, 0.0 B)
filesystem read time (sampled) total (min, med, max)	471 ms (95 ms, 134 ms, 135 ms)
max distance between the positions of scanned columns in the relation	10
max partition size chosen	0.0 B
metadata time	0 ms
missing files	0
number of columns in the relation	11
number of files read	4
number of parquet row groups read	4
number of partitions read	4
number of scanned columns	11
relative skew in total splits sizes distribution	0.0 B
rows output	8

Now enable DPP

```
spark.conf.set("spark.sql.adaptive.enabled","true")
spark.conf.set("spark.sql.optimizer.dynamicPartitionPruning.enabled", "true")
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 5 * 1024 * 1024)
```



Preparing the Partitioned Data

```
df.write.format("parquet")\
        .mode("append")\
        .partitionBy("Item_identifier")\
        .option("path","/FileStore/rawdata/dpp_partionednew")\
        .save()
```

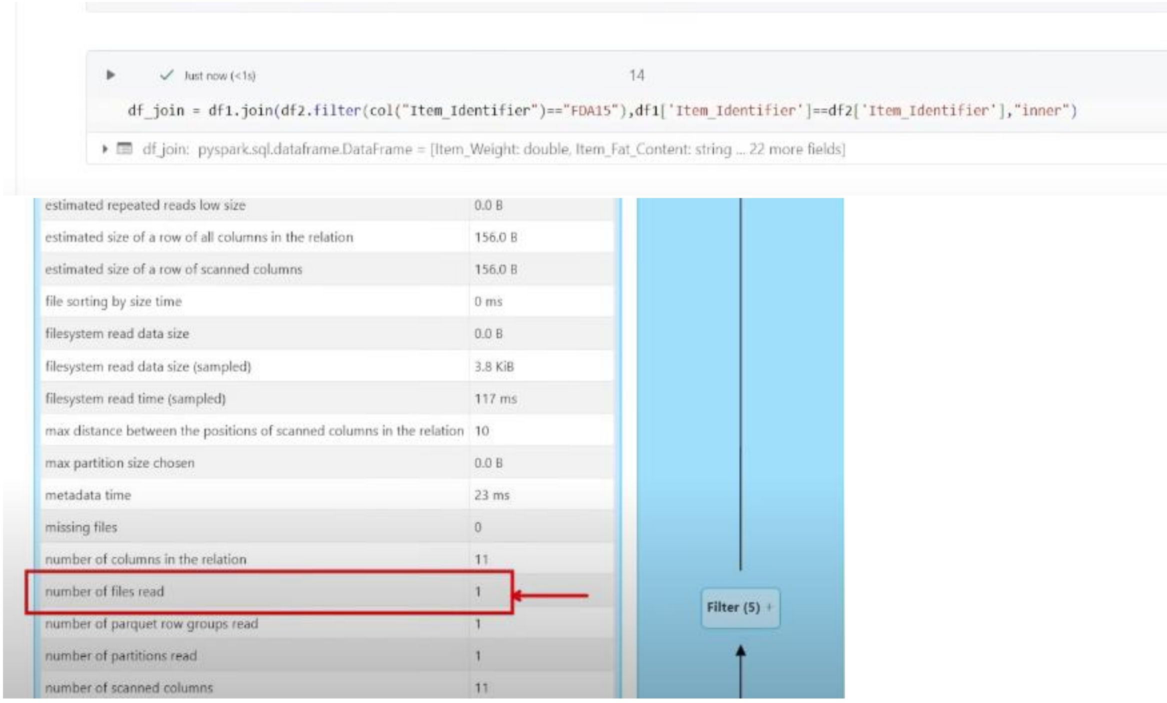
(1) Spark Jobs

Dataframes

```
df1 = spark.read.format("parquet")\
        .load("/FileStore/rawdata/dpp_partionednew")
```

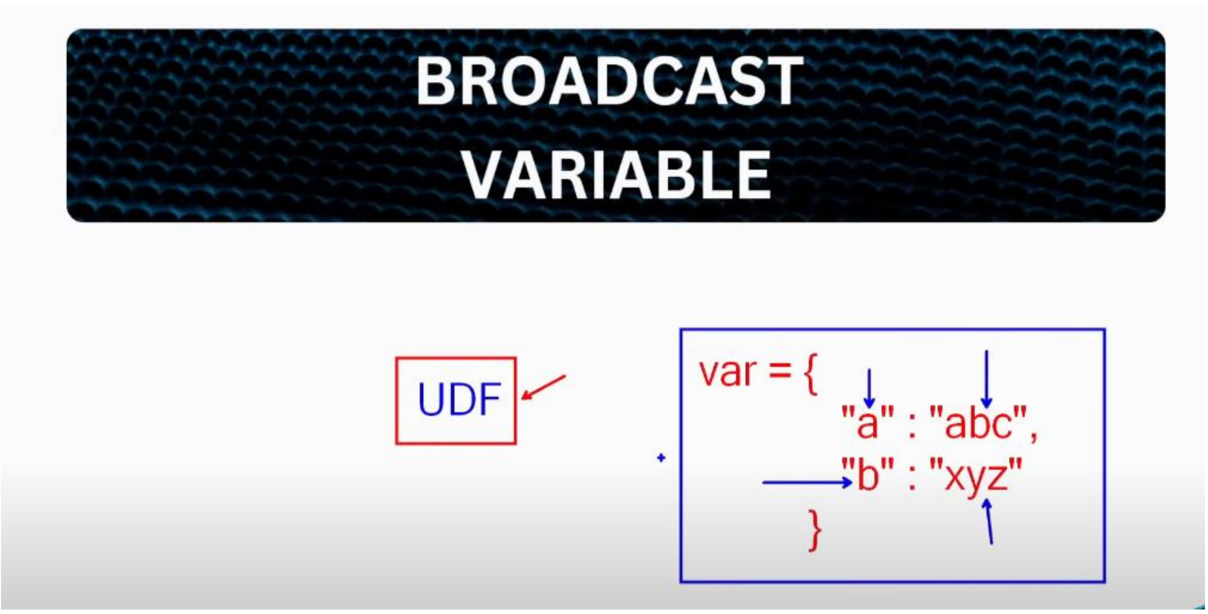
(1) Spark Jobs

df1: pyspark.sql.dataframe.DataFrame = [Item_Identifier: string, Item_Weight: dou



8. BROADCAST VARIABLE

Lets say if you are having a dictionary inside a UDF (like a mapping), then we pass the variable to all the Executors, but doing it over & over again causes a n/w overhead



Simply broadcast the variable to all the executors, so that everytime this variable is needed, it can be retreived from the memory

Broadcast Variable

Python

File Edit View Run Help Last edit was 10 minutes ago

06:30 PM (3s) 1

```python
df = spark.createDataFrame([
    ("1001",),
    ("1002",),
    ("1004",)
], ["product_id"])

# Lookup dictionary (small)
product_dict = {
    "1001": "iPhone",
    "1002": "Samsung",
    "1003": "Pixel"
}
```

df: pyspark.sql.dataframe.DataFrame = [product_id: string]

Just now (<1s) 3

```python
# Broadcasting the dictionary variable

broad_vr = spark.sparkContext.broadcast(product_dict)
```

Just now (<1s) 4

```python
broad_vr.value
```

Out[11]: {'1001': 'iPhone', '1002': 'Samsung', '1003': 'Pixel'}

Just now (<1s)

```python
broad_vr.value.get('1001')
```

Out[12]: 'iPhone'

Just now (<1s)

```python
# Our Function

def mymap(x):

    return broad_vr.value.get(x)
```

Just now (<1s)

```python
mymap_udf = udf(mymap)
```

Just now (<1s) 8

```python
df_with_names = df.withColumn("product_name", mymap_udf("product_id"))
```

df_with_names: pyspark.sql.dataframe.DataFrame = [product_id: string, product_name: string]

Just now (1s)

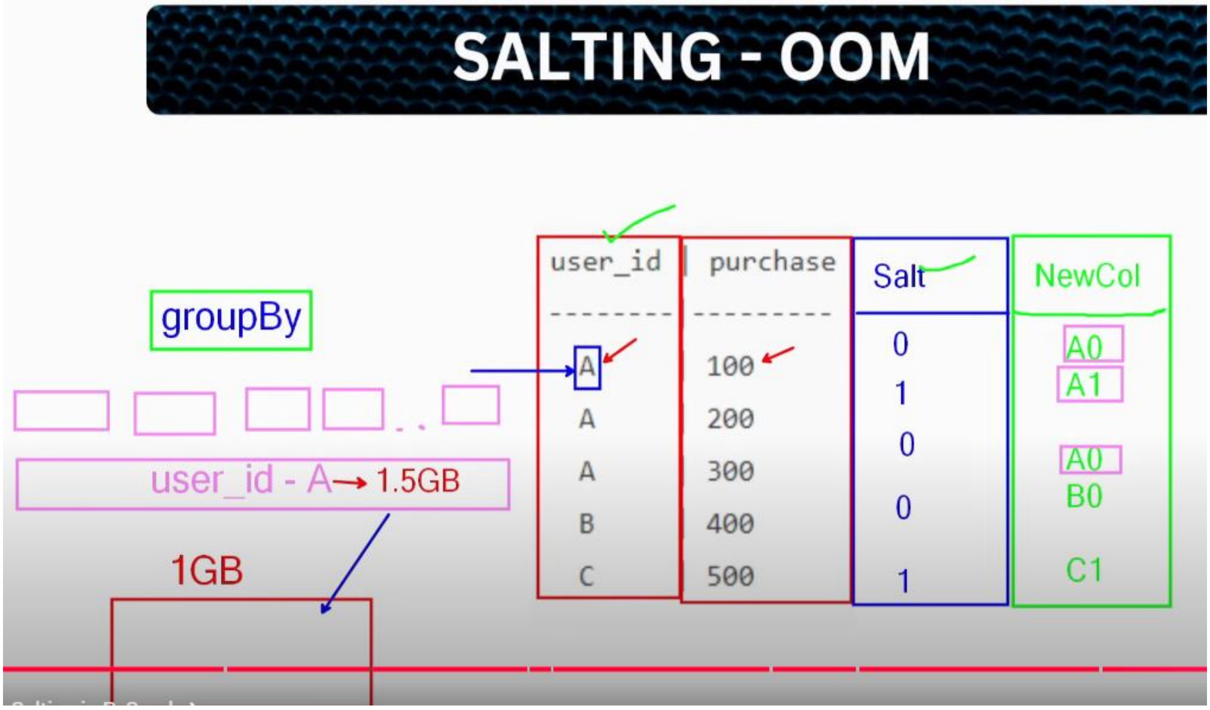
```python
df_with_names.display()
```

(3) Spark Jobs

Table +

	product_id	product_name
1	1001	iPhone
2	1002	Samsung
3	1004	null

9. SALTING



Partition of A is very very huge, lets say 1.5GB whereas Executor has 1GB memory, partition will not memory→ OOM

Introduce a new column with random numbers, break big partition into smaller partitions

SALTING Python

File Edit View Run Help Last edit was now

CREATING A DATAFRAME

```
data = [("A", 100), ("A", 200), ("A", 300), ("B", 400), ("C", 500)]
df = spark.createDataFrame(data, ["user_id", "purchase"])
```

ADDING SALT COLUMN

✓ Just now (<1s)

```
df = df.withColumn("salt_column",floor(rand()*3))
```

df: pyspark.sql.dataframe.DataFrame = [user_id: string, purchase: long ... 1 more field]

	user_id	purchase	salt_column
1	A	100	0
2	A	200	0
3	A	300	1
4	B	400	2
5	C	500	2

Creating Concat Column on original groupBy col and salt_column to create a new groupBy col

Last execution failed 9 Python

```python
df = df.withColumn("user_id_salt",concat(col("user_id"),lit("-"),col("salt_column")))

df.display()
```

	user_id	purchase	salt_column	user_id_salt
1	A	100	0	A-0
2	A	200	0	A-0
3	A	300	1	A-1
4	B	400	2	B-2
5	C	500	2	C-2

Applying Group By on this new col

Interrupt 00:01 11

```python
df = df.groupBy("user_id_salt").agg(sum("purchase"))

df.display()
```

(1) Spark Jobs

Table

	user_id_salt	sum(purchase)
1	A-0	300
2	A-1	300
3	B-2	400
4	C-2	500