

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.

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

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

Not required

Just now (3s)

spark

SparkSession - hive

SparkContext

[Spark UI](#)

Version v3.3.2

Master local[8]

AppName Databricks Shell

Click on Spark UI

Jobs Stages Storage Environment **Executors** SQL / DataFrame JDBC/ODBC Server Structured Streaming

Executors

Show Additional Metrics

Summary

	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

Executors

Show 20 entries

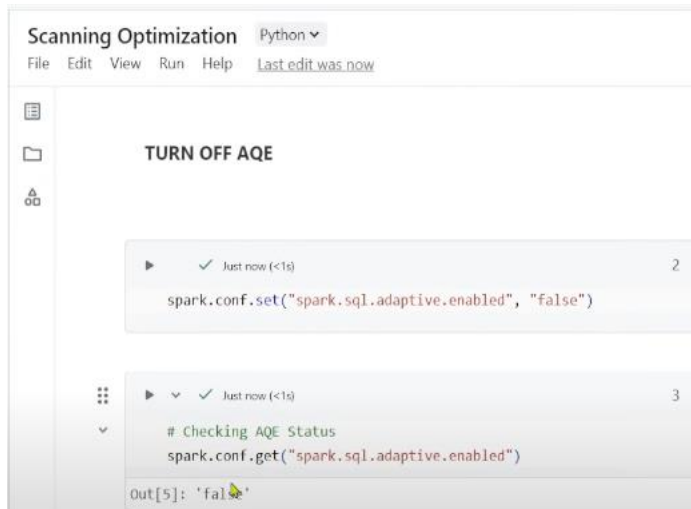
Search:

Executor	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)
----------	---------	--------	------------	----------------	-----------	-------	--------------	--------------	----------------	-------------	---------------------

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.



```
Scanning Optimization Python
File Edit View Run Help Last edit was now

TURN OFF AQE

▶ ✓ Just now (<1s) 2
spark.conf.set("spark.sql.adaptive.enabled", "false")

▶ ✓ Just now (<1s) 3
# Checking AQE Status
spark.conf.get("spark.sql.adaptive.enabled")
Out[5]: 'false'
```



```
Data Reading

▶ ✓ 11:22 AM (17s) 6
df = spark.read.format("csv")\
    .option("inferSchema", True)\
    .option("header", True)\
    .load("/FileStore/rawdata/BigMart_Sales.csv")
▶ (2) Spark Jobs
```

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



```
Get No. of Partitions

▶ ✓ Just now (<1s)
df.rdd.getNumPartitions()
Out[9]: 1
```

Since we have peanut amount of data

Changing DEFAULT Partition Size to 128KB



```
▶ ✓ Just now (<1s) 11
# Changing the default partition size to 128KB
spark.conf.set("spark.sql.files.maxPartitionBytes", "131072")

▶ ✓ Just now (<1s) 12
df.rdd.getNumPartitions()
Out[13]: 7
```

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_Identifier: string, Item_Weight: double ... 10 more fields]

```
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

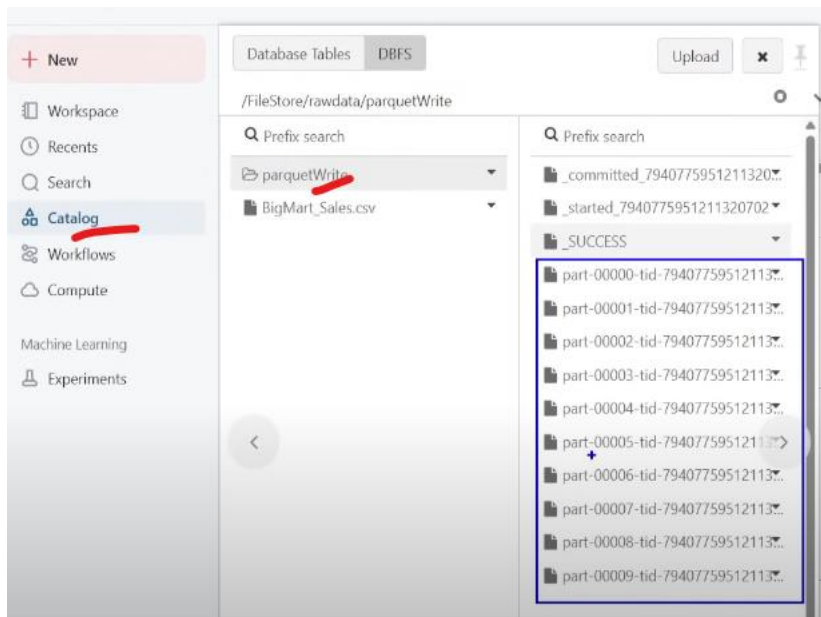
```
df.withColumn("partition_id", spark_partition_id()).display()
```

(2) Spark Jobs

Data Writing

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

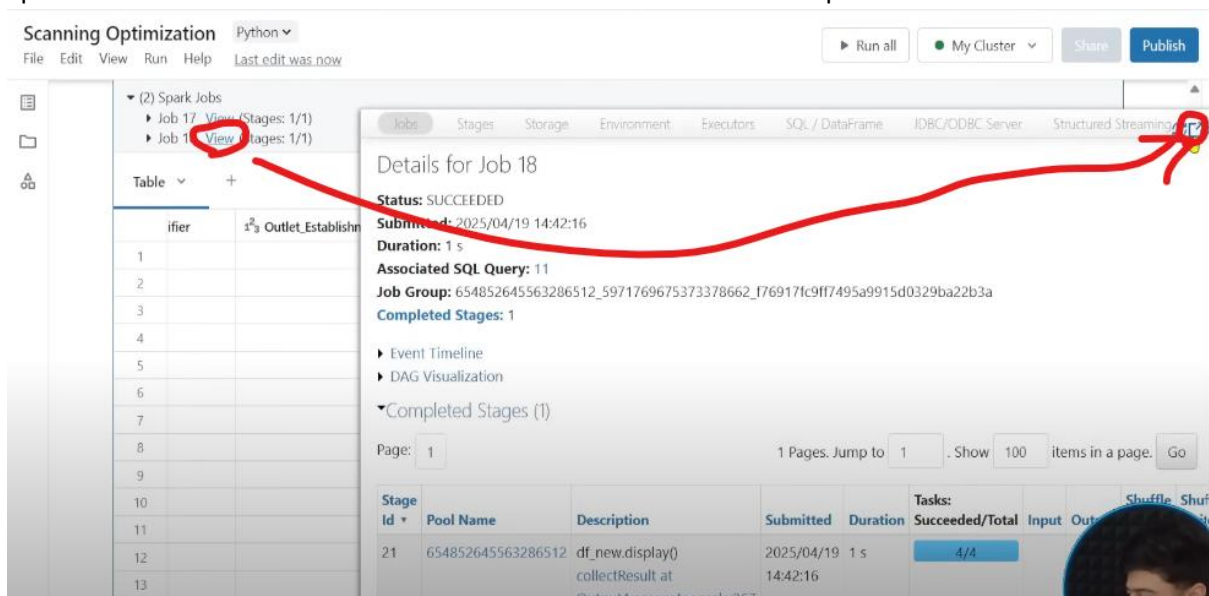
(2) Spark Jobs



New Data Reading



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



Click on Jobs → Expand → SQL dataframe

Jobs Stages Storage Environment Executors SQL / DataFrame JDBC/ODBC Server Structured Streaming					
SQL / DataFrame					
Completed Queries: 12					
Completed Queries (12)					
Page: 1		1 Pages. Jump to 1. Show 100 items in a pag			
ID	Description	Submitted	Duration	Job IDs	Sub Execution IDs
11	df_new.display()	2025/04/19 14:42:15	2 s	[17][18]	
10	df_new.display()	2025/04/19 14:41:10	9 s	[14][15]	
9	show tables in 'default'	2025/04/19 14:40:49	17 ms		
8	show tables in 'default'	2025/04/19 14:40:49	0.2 s		
7	show databases	2025/04/19 14:40:48	13 ms		
6	show databases	2025/04/19 14:40:34	14 s		

Jobs Stages Storage Environment Executors SQL / DataFrame JDBC/ODBC Server Structured Streaming	
cluster's current parallelism	0
corrupt files	0
estimated repeated reads high size total (min, med, max)	0.0 B (0.0 B, 0.0 B, 0.0 B)
estimated repeated reads low size total (min, med, max)	0.0 B (0.0 B, 0.0 B, 0.0 B)
estimated size of a row of all columns in the relation	176.0 B
estimated size of a row of scanned columns	176.0 B
file sorting by size time	0 ms
filesystem read data size (sampled) total (min, med, max)	590.4 KiB (114.7 KiB, 118.7 KiB, 122.0 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)	1.0 s (169 ms, 205 ms, 232 ms)
max distance between the positions of scanned columns in the relation	11
max partition size chosen	0.0 B
metadata time	0 ms
missing files	0
number of columns in the relation	12
number of files read	10
number of parquet row groups read	10
number of scanned columns	12
relative skew in total splits sizes distribution	0.0 B

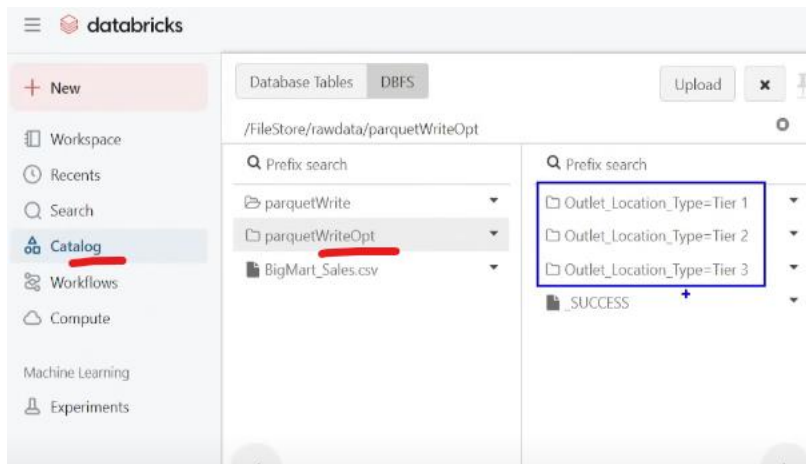
SCANNING OPTIMIZATION

Solution →

```

1 df.write.format("parquet")\
2   .mode("append")\
3   .partitionBy("Outlet_Location_Type")\
4   .option("path", "/FileStore/rawdata/parquetwriteOpt")\
5   .save()
  
```

▶ (1) Spark Jobs

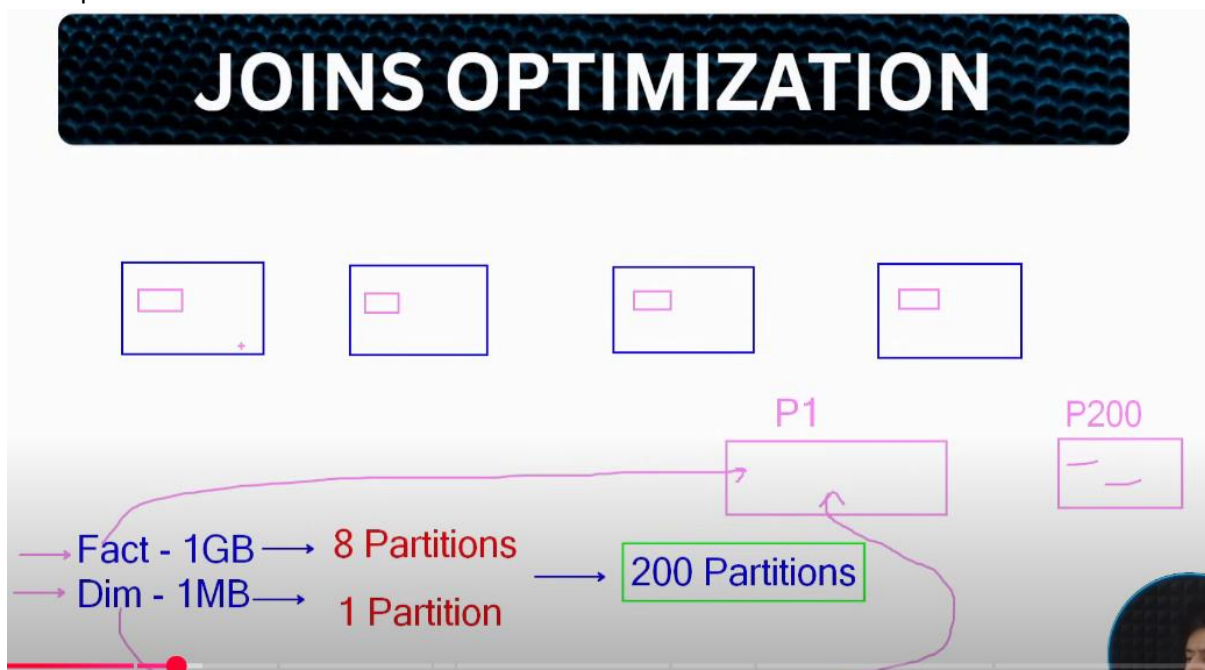


Generally do Partitions on date columns → best column → year, month, date

2. Join Optimizations

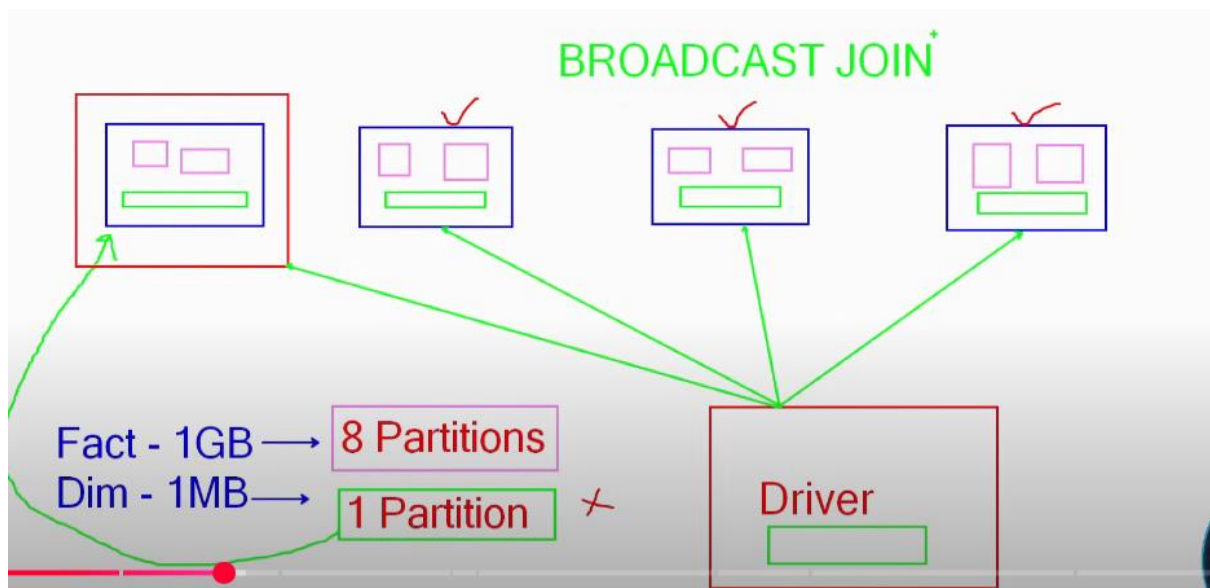
Lets say we have 4 Executors, 1 Fact of size 1GB and 1 Dimension of 1 MB

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



Optimize Joins

Python

File Edit View Run Help Last edit was 21 minutes ago

Waiting

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

```
# 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"])
```

Just now (<1s)

5

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

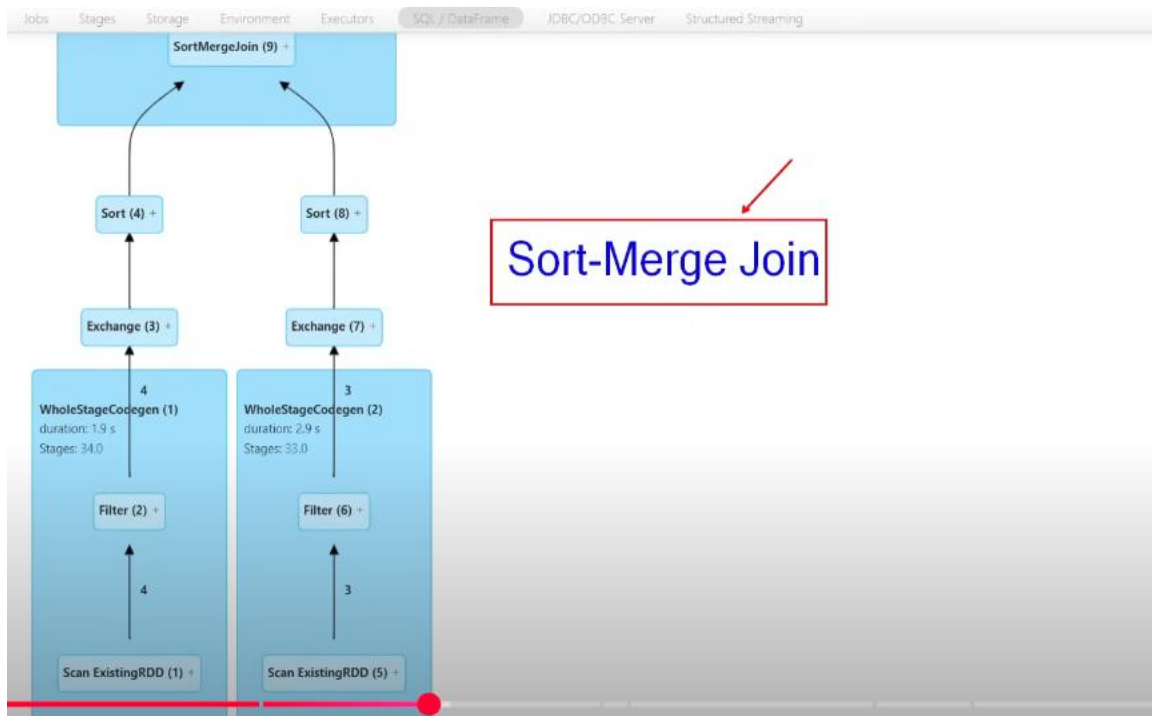
df_join: pyspark.sql.dataframe.DataFrame = [id: long, country_code: string ... 3 more fields]

Interrupt 00:01

6

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

Exchange (3) +
Stages: 34.0 35.0 38.0 41.0 44.0 47.0

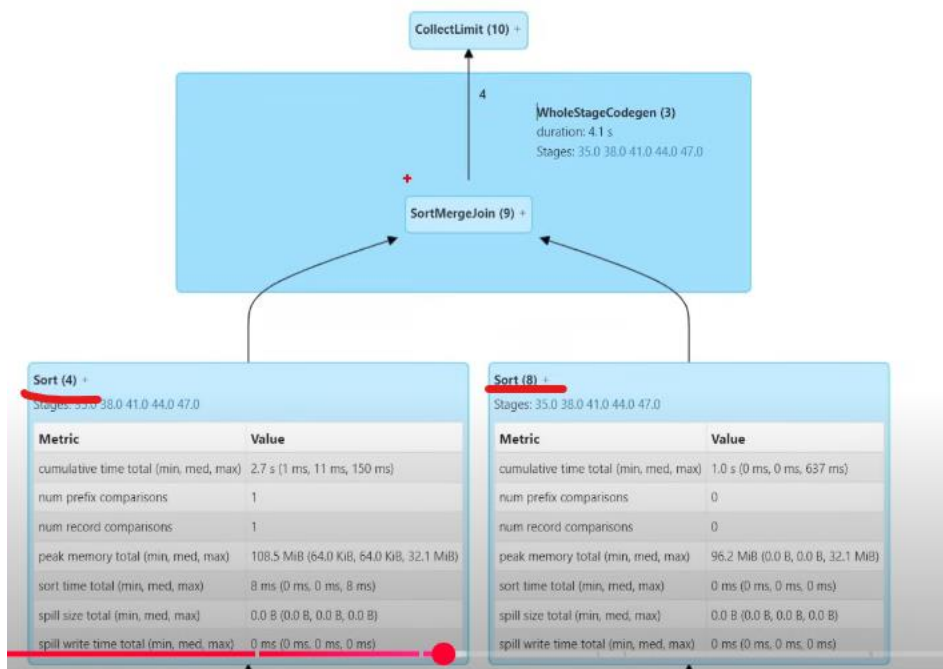
Metric	Value
data size total (min, med, max)	160.0 B (0.0 B, 40.0 B, 40.0 B)
fetch wait time total (min, med, max)	0 ms (0 ms, 0 ms, 0 ms)
local blocks read	4
local bytes read total (min, med, max)	296.0 B (0.0 B, 0.0 B, 150.0 B)
number of partitions	200
records read	4
remote blocks read	0
remote bytes read	0.0 B
remote bytes read to disk	0.0 B
shuffle bytes written total (min, med, max)	296.0 B (0.0 B, 71.0 B, 75.0 B)
shuffle records written	4
shuffle write time total (min, med, max)	515 ms (0 ms, 114 ms, 136 ms)

Exchange (7) +
Stages: 33.0 35.0 38.0 41.0 44.0 47.0

Metric	Value
data size total (min, med, max)	136.0 B (0.0 B, 0.0 B, 48.0 B)
fetch wait time total (min, med, max)	0 ms (0 ms, 0 ms, 0 ms)
local blocks read	3
local bytes read total (min, med, max)	261.0 B (0.0 B, 0.0 B, 90.0 B)
number of partitions	200
records read	3
remote blocks read	0
remote bytes read	0.0 B
remote bytes read to disk	0.0 B
shuffle bytes written total (min, med, max)	261.0 B (0.0 B, 0.0 B, 90.0 B)
shuffle records written	3
shuffle write time total (min, med, max)	718 ms (0 ms, 0 ms, 261 ms)

Exchange
hashpartitioning(country_code#1198, 200), ENSURE_REQUIREMENTS.
[plan_id=430]

200 partitons for 3-4 records, this is crazy



Then it performs sorting and finally sort merge join

Optimize Joins Python

File Edit View Run Help [Last edit was now](#)

```

from pyspark.sql.functions import *

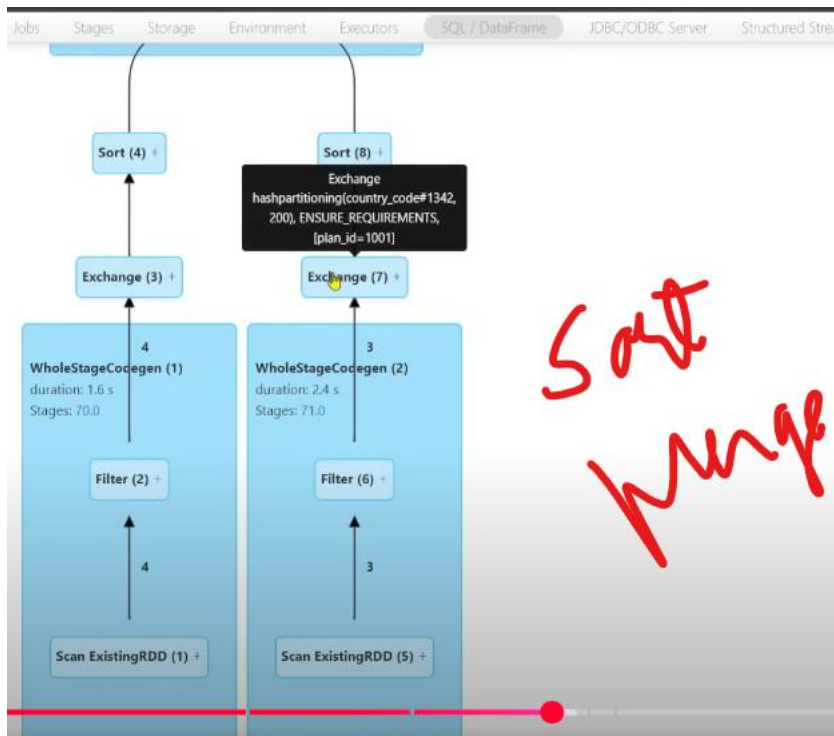
```

```

df_join_opt = df_transactions.join(broadcast(df_countries),df_transactions['country_code']==df_countries['country_code'],'inner')

```

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



```

df_sql_opt = spark.sql(
    '''SELECT * /* broadcast(c) */
FROM transactions t
JOIN countries c
ON t.country_code = c.country_code'''
)
df_sql_opt: pyspark.sql.dataframe.DataFrame = [id: long, country_code:

```

```

df_sql_opt.display()

(5) Spark Jobs
  ▶ Job 46 View (Stages: 3/3)
  ▶ Job 47 View (Stages: 1/1, 2 skipped)
  ▶ Job 48 View (Stages: 1/1, 2 skipped)
  ▶ Job 49 View (Stages: 1/1, 2 skipped)
  ▶ Job 50 View (Stages: 1/1, 2 skipped)

```

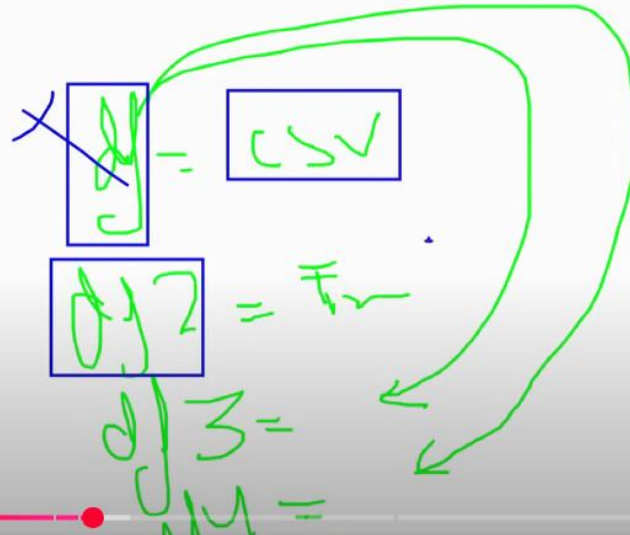
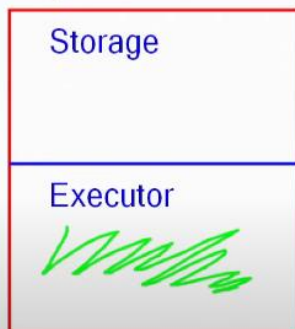
It's just an advice to Spark, not a guarantee, it may consider may not.

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

CACHING & PERSISTENCE

Spark Pool 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.persist(storageLevel.DISK_AND_MEMORY) == df.cache()`

+

DISK_ONLY
MEMORY_ONLY

```
2 minutes ago (2s) 2
df = spark.read.format("csv")\
    .option("inferSchema",True)\
    .option("header",True)\
    .load("/FileStore/rawdata/BigMart_Sales.csv")
(2) Spark Jobs
df: pyspark.sql.dataframe.DataFrame = [Item_Identifier: string, Item_Weight: double ... 10 more fields]

Just now (<1s) 3
df2 = df.filter(col("Outlet_Location_Type") == 'Tier 1')
df2: pyspark.sql.dataframe.DataFrame = [Item_Identifier: string, Item_Weight: double ... 10 more fields]

Just now (<1s) 4
df3 = df.filter(col("Outlet_Location_Type") == 'Tier 2')
df3: pyspark.sql.dataframe.DataFrame = [Item_Identifier: string, Item_Weight: double ... 10 more fields]
```

df will be computed multiple times

JobsStagesStorageEnvironmentExecutorsSQL / DataFrameJDBC/ODBC ServerStructured Streaming

Storage

Parquet IO Cache

Data Read from External Filesystem (All Formats)	Data Read from IO Cache (Cache Hits, Compressed)	Data Written to IO Cache (Compressed)	Cache Misses (Compressed)	True Cache Misses	Partial Cache Misses	Rescheduling Cache Misses	Cache Hit Ratio	Number of Local Scan Tasks	Number of Rescheduled Scan Tasks	Cache Metadata Manager Peak Disk Usage
0.0 B	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B	0 %	0	0	0.0 B

▼RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
224	FileScan csv [Item_Identifier#1891,Item_Weight#1892,Item_Fat_Content#1893,Item_Visibility#1894,Item_Type#1895,Item_MRP#1896,Outlet_Identifier#1897,Outlet_Establishment_Year#1898,Outlet_Size#1899,Outlet_Location_Type#1900,Outlet_Type#1901,Item_Outlet_Sales#1902] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 paths)[dbfs:/FileStore/rawdata/BigMart_Sales.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<Item_Identifier:string,Item_Weight:double,Item_Fat_Content:string,Item_Visibility:double,L...	Disk Memory Deserialized 1x Replicated	1	100%	406.6 KIB	0.0 B

To finally remove the cached data

2,785 rows | 1.58s runtime
Refreshed 1 minute ago

```
df.unpersist()
```

Out[10]: DataFrame[Item_Identifier: string, Item_Weight: double, Item_Fat_Content: string, Item_Visibility: double, Item_Type: string, Item_MRP: double, Outlet_Identifier: string, Outlet_Establishment_Year: int, Outlet_Size: string, Outlet_Location_Type: string, Outlet_Type: string, Item_Outlet_Sales: double]

```
from pyspark.storagelevel import StorageLevel
```

```
df.persist(StorageLevel.MEMORY_ONLY)
```

df: pyspark.sql.dataframe.DataFrame = [Item_Identifier: string, Item_Weight: double, Item_Fat_Content: string, Item_Visibility: double, Item_Type: string, Item_MRP: double, Outlet_Identifier: string, Outlet_Establishment_Year: int, Outlet_Size: string, Outlet_Location_Type: string, Outlet_Type: string, Item_Outlet_Sales: double]

5. DYNAMIC RESOURCE ALLOCATION

Now auto enabled

DYNAMIC RESOURCE ALLOCATION

```
spark-submit \
  --conf spark.dynamicAllocation.enabled=true \
  --conf spark.shuffle.service.enabled=true \
  --conf spark.dynamicAllocation.minExecutors=2 \
  --conf spark.dynamicAllocation.initialExecutors=4 \
  --conf spark.dynamicAllocation.maxExecutors=10 \
  my_spark_job.py
```


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 can't 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, It's like Spark Pool in Synapse, DB and Fabric.

6. AQE (Adaptive Query Execution)

AQE ADAPTIVE QUERY EXECUTION

- Dynamically Coalesce the partitions
- Optimizing the Join Strategy during runtime
- Optimizing the skewness

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

Turning OFF the AQE

The screenshot shows a Jupyter Notebook interface with the following components:

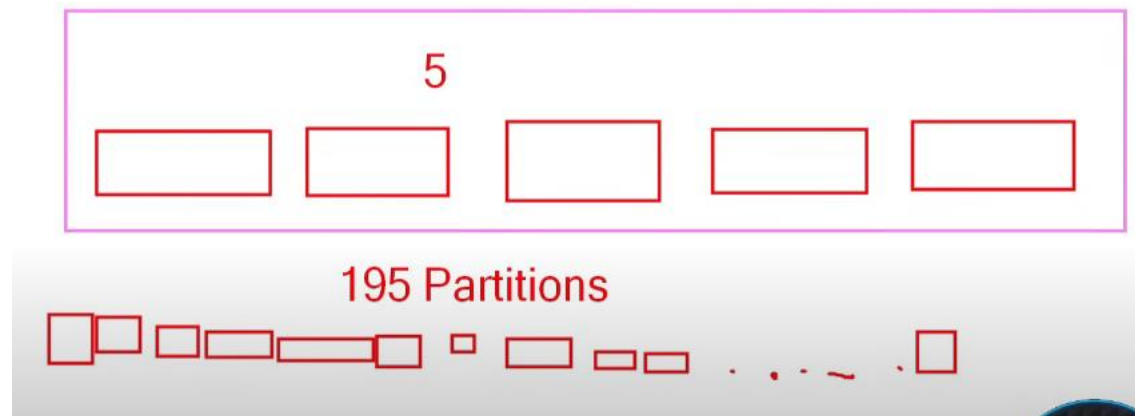
- Code Cell 1:** `spark.conf.set("spark.sql.adaptive.enabled", "false")`
- Code Cell 2:** `from pyspark.sql.functions import *`
- Code Cell 3:** `df = spark.read.format("csv")\n .option("inferSchema", True)\n .option("header", True)\n .load("/FileStore/rawdata/BigMart_Sales.csv")`
- Code Cell 4:** `df_new = df.groupBy("Item_fat_Content").count()\n df_new.display()`
- Output Cell 1:** `df.rdd.getNumPartitions()` returns `Out[4]: 1`
- Table View:** A table showing the grouped data with 5 partitions.

	Item_fat_Content	count
1	low fat	112
2	Low Fat	5089
3	LF	316
4	Regular	2889
5	reg	117

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

☐ Show experimental metrics

Before AQE



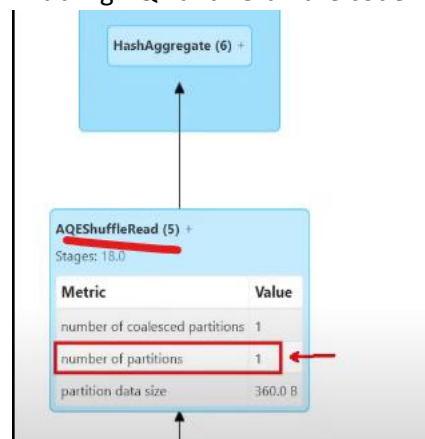
Exchange
hashpartitioning(Item_fat_Content#48
,200), ENSURE_REQUIREMENTS,
[plan_id=67]

Exchange (3) +
Stages: 3.0 4.0 6.0 8.0 10.0 12.0

Metric	Value
data size	160.0 B
fetch wait time total (min, med, max)	0 ms (0 ms, 0 ms, 0 ms)
local blocks read	5
local bytes read total (min, med, max)	358.0 B (0.0 B, 0.0 B, 72.0 B)
number of partitions	200
records read	5
remote blocks read	0
remote bytes read	0.0 B
remote bytes read to disk	0.0 B
shuffle bytes written	358.0 B
shuffle records written	5
shuffle write time	66 ms

```
✓ Just now (<1s)  
spark.conf.set("spark.sql.adaptive.enabled", "true")
```

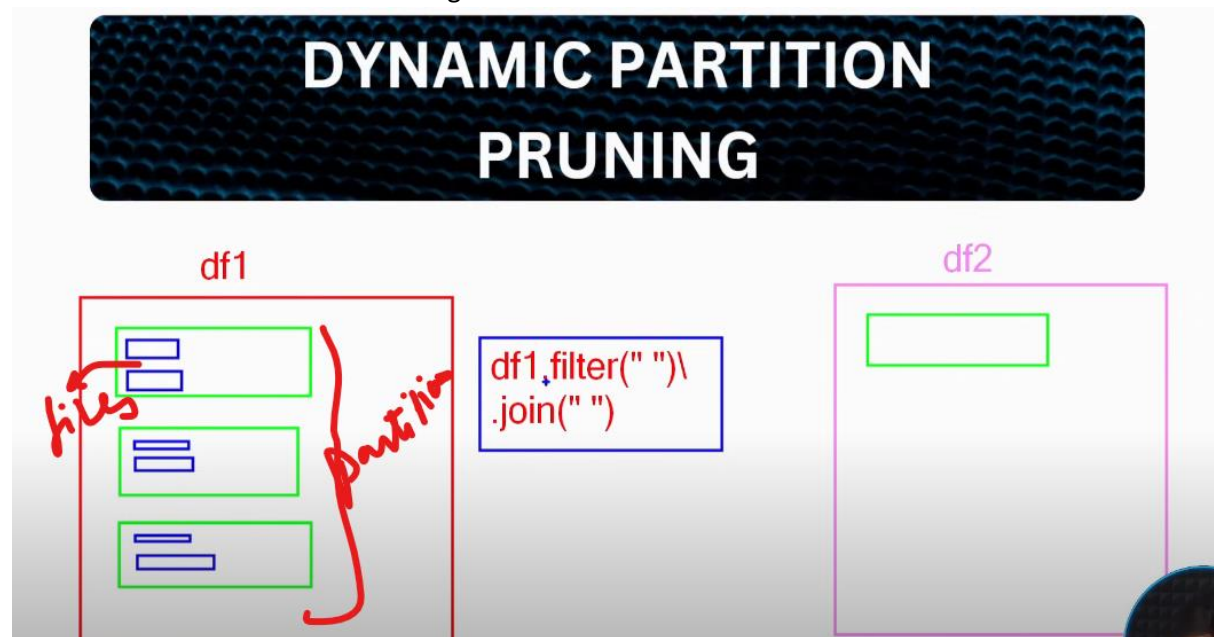
Enabling AQE and rerun the code



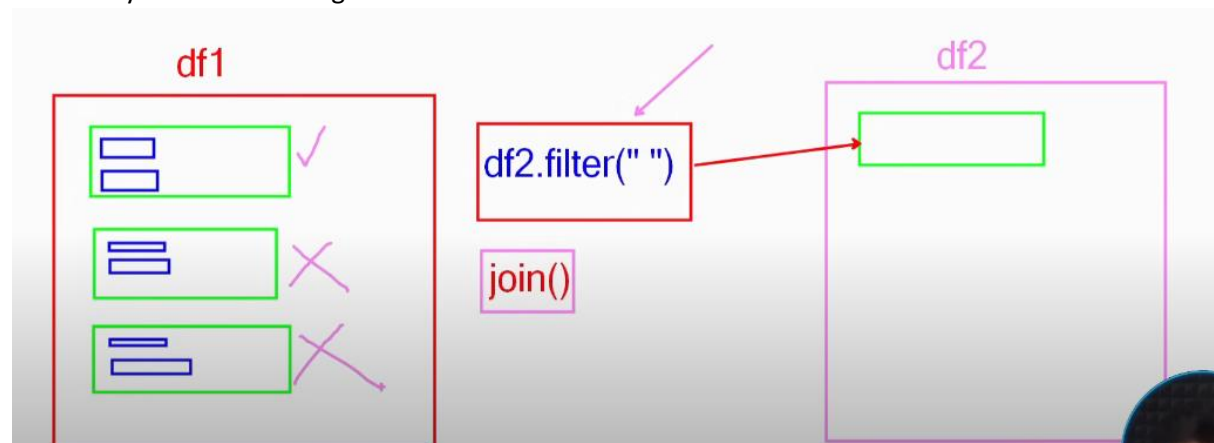
from 200 partitions to 1 partition, AQE firstly removes all the 195 empty partitions and then coalesces the partitions from 5 to 1.

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.

⚙️ Turning OFF AQE and DPP and AutoBroadcast

```
04:47 PM (<1s) 2
spark.conf.set("spark.sql.adaptive.enabled", "false")
spark.conf.set("spark.sql.optimizer.dynamicPartitionPruning.enabled", "false")
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
```

```
Waiting 3
1 df = spark.read.format("csv")\
2   |   |   |   |   |   |   |   |   |
3   |   |   |   |   |   |   |   |   |
4   |   |   |   |   |   |   |   |   |
5   |   |   |   |   |   |   |   |   |
6 df.limit(8).display()
```

Preparing the Partitioned Data

```
Interrupt 0001 6
1 df.write.format("parquet")\
2   |   |   |   |   |   |   |   |   |
3   |   |   |   |   |   |   |   |   |
4   |   |   |   |   |   |   |   |   |
5   |   |   |   |   |   |   |   |   |
6 df.save()
```

Non Partitioned Data

```
Just now (2s) 8
df.write.format("parquet")\
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
df.save()
```

Dataframes

```
Just now (2s) 10
df1 = spark.read.format("parquet")\
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
df1: pyspark.sql.dataframe.DataFrame = [Item_Identifier: string, Item_Weight: double ... 10 more fields]

Waiting 11
df2 = spark.read.format("parquet")\
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
df2: pyspark.sql.dataframe.DataFrame = [Item_Identifier: string, Item_Weight: double ... 10 more fields]
```

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

Scan parquet (6) +

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_partitionednew")\
  .save()

```

(1) Spark Jobs

Dataframes

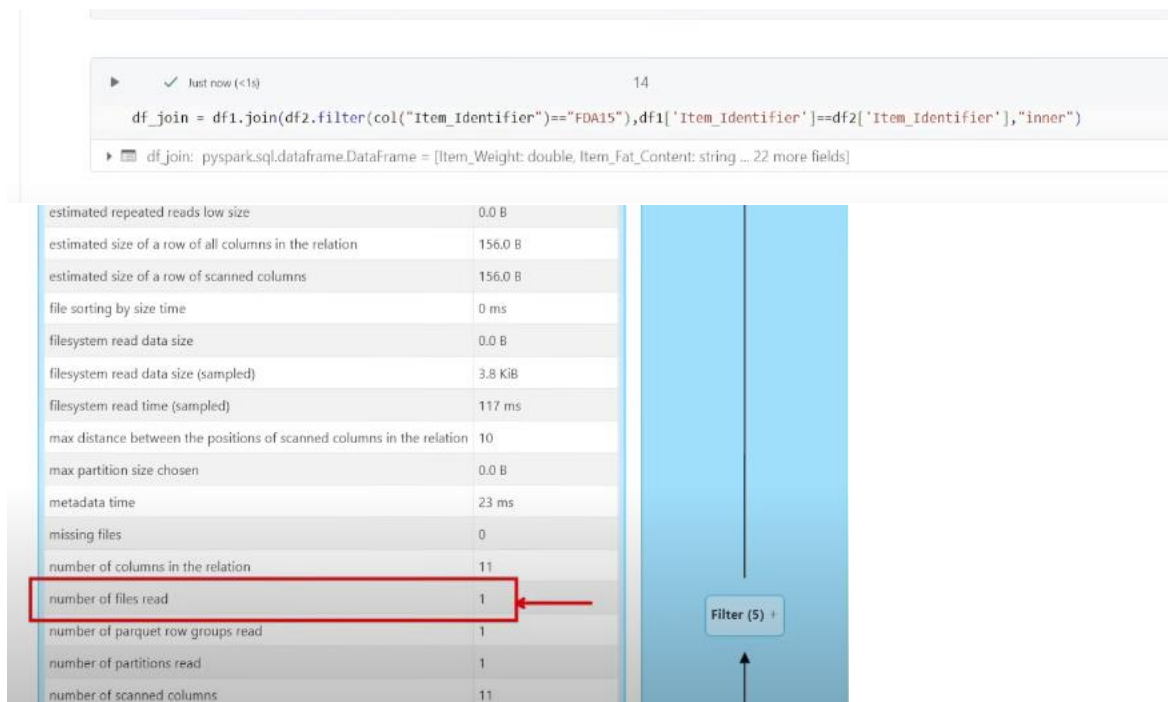
```

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

```

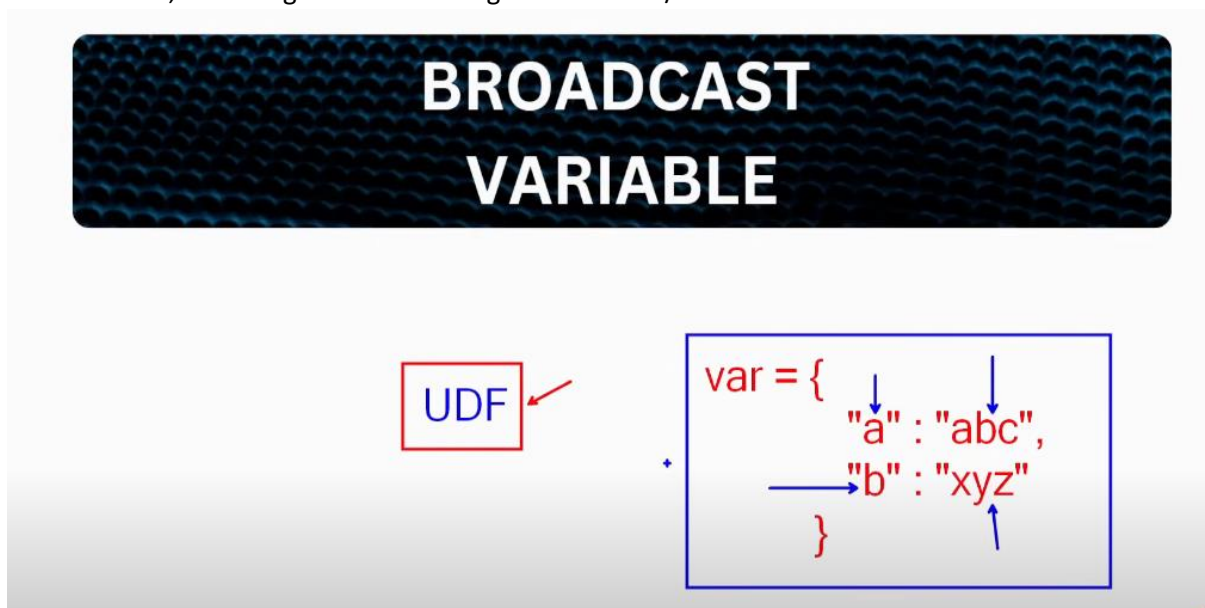
(1) Spark Jobs

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



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 retrieved from the memory

Broadcast Variable Python

```
File Edit View Run Help Last edit was 10 minutes ago
```

```
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]

```
# Broadcasting the dictionary variable

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

```
broad_vr.value
```

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

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

Out[12]: 'iPhone'

```
# Our Function

def mymap(x):
    return broad_vr.value.get(x)
```

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

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

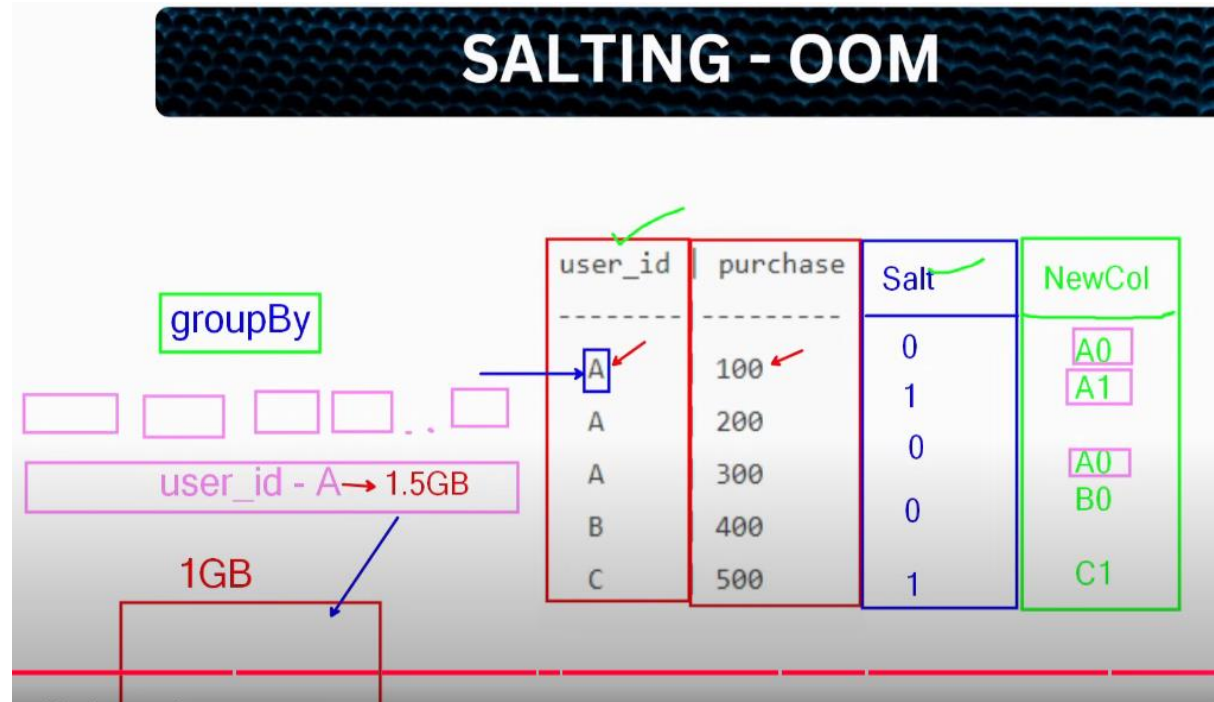
df_with_names: pyspark.sql.dataframe.DataFrame = [product_id: string, product_name: string]

```
df_with_names.display()
```

(3) Spark Jobs

	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

```
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

Python

```
1 df = df.withColumn("user_id_salt", concat(col("user_id"), lit("-"), col("salt_column")))
2
3 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

```
1 df = df.groupBy("user_id_salt").agg(sum("purchase"))
2
3 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