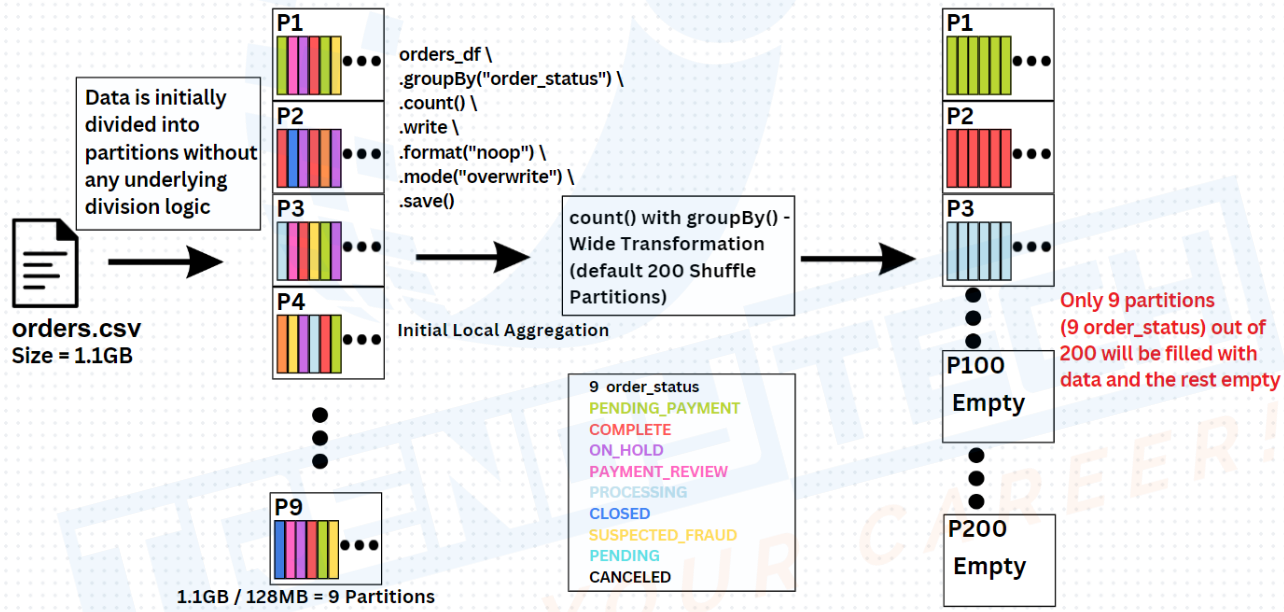
How a groupBy() works

Consider an example of orders.csv dataset of 1.1GB

(This dataset has a column named order_status with 9 unique values - CLOSED, PENDING, ON_HOLD, COMPLETE, PENDING_PAYMENT, CANCELLED, PROCESSING, PAYMENT_REVIEW, SUSPECTED_FRAUD)

- When a wide transformation is triggered, by default, 200 shuffle partitions are created.
- Since there are only 9 unique keys (in order_status), there would be at the max only 9 partitions that will have data in it and the remaining 191 partitions will remain empty (as shown in the diagram below)



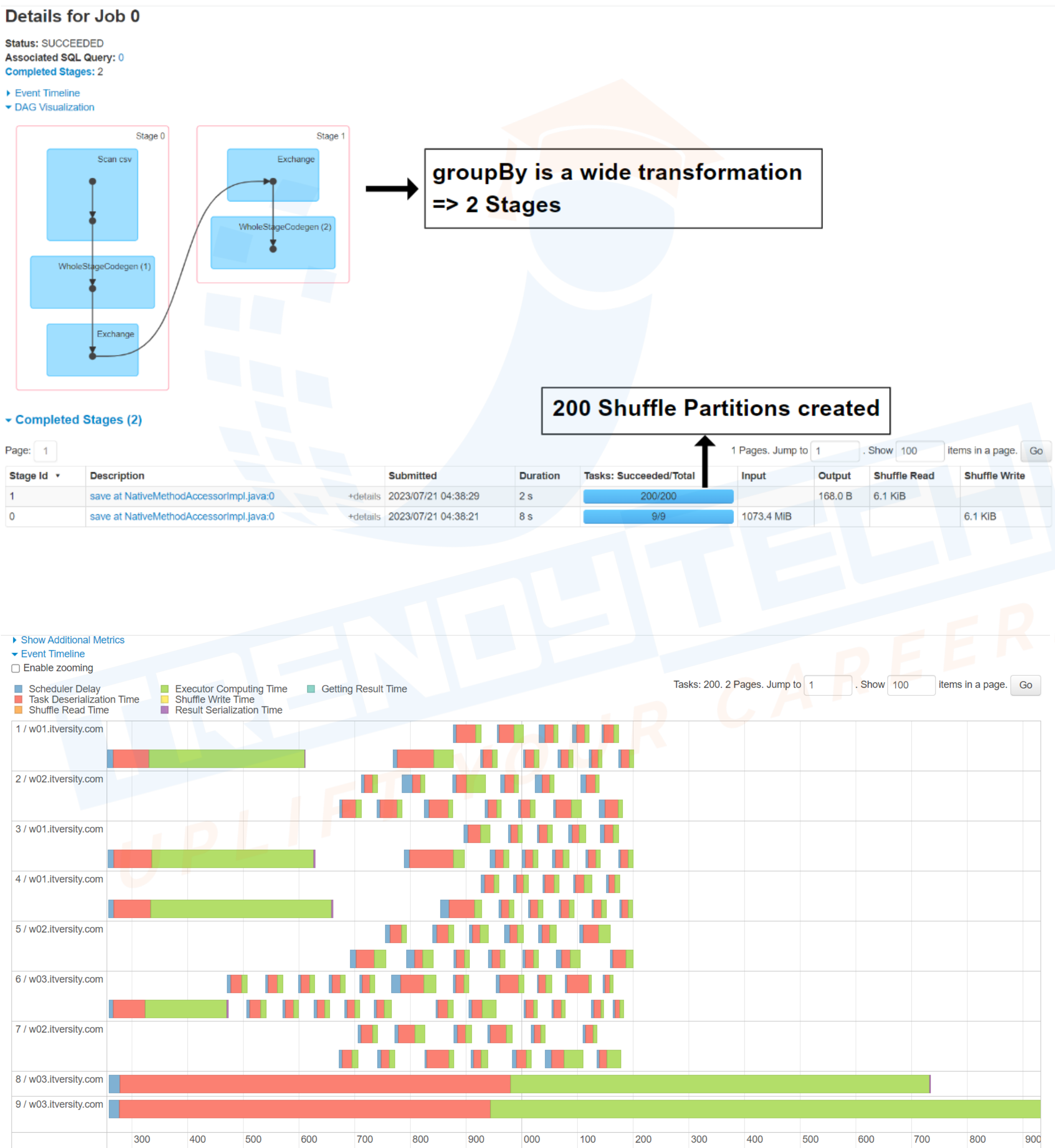
- Higher level APIs always try to optimize the process. In the above example, for aggregation operations like - groupBy().count(), Spark performs initial local aggregation.
- Each partition will have 9 entries as there are 9 different values for order_status.
- The data gets shuffled after the wide transformation - groupBy().count(). All the COMPLETE records coming from the different initial partitions will be shuffled to a single partition (likewise, for the other order_status). Overall, there would be at the max 9 partitions holding data of one

order_status each. (Each partition can also hold data of more than one order_status based on the available memory)

- On executing groupBy as shown below

```
orders_df.groupBy("order_status").count().write.format("csv").mode("overwrite").save("output07")
```

- The Spark UI visualisation on executing the above command



- Here the task scheduler gets overburdened to create tasks that would remain empty without any data and performs no operations. (There would be ~ 191 such tasks for the above scenario)
- orders_df.groupBy("order_status").count().write.format("noop").mode("overwrite").save() This is usually for testing purposes, where you don't intend to write back to the disk each time while testing.

Broadcast Join & Normal Shuffle-Sort-Merge Join

Broadcast join happens when

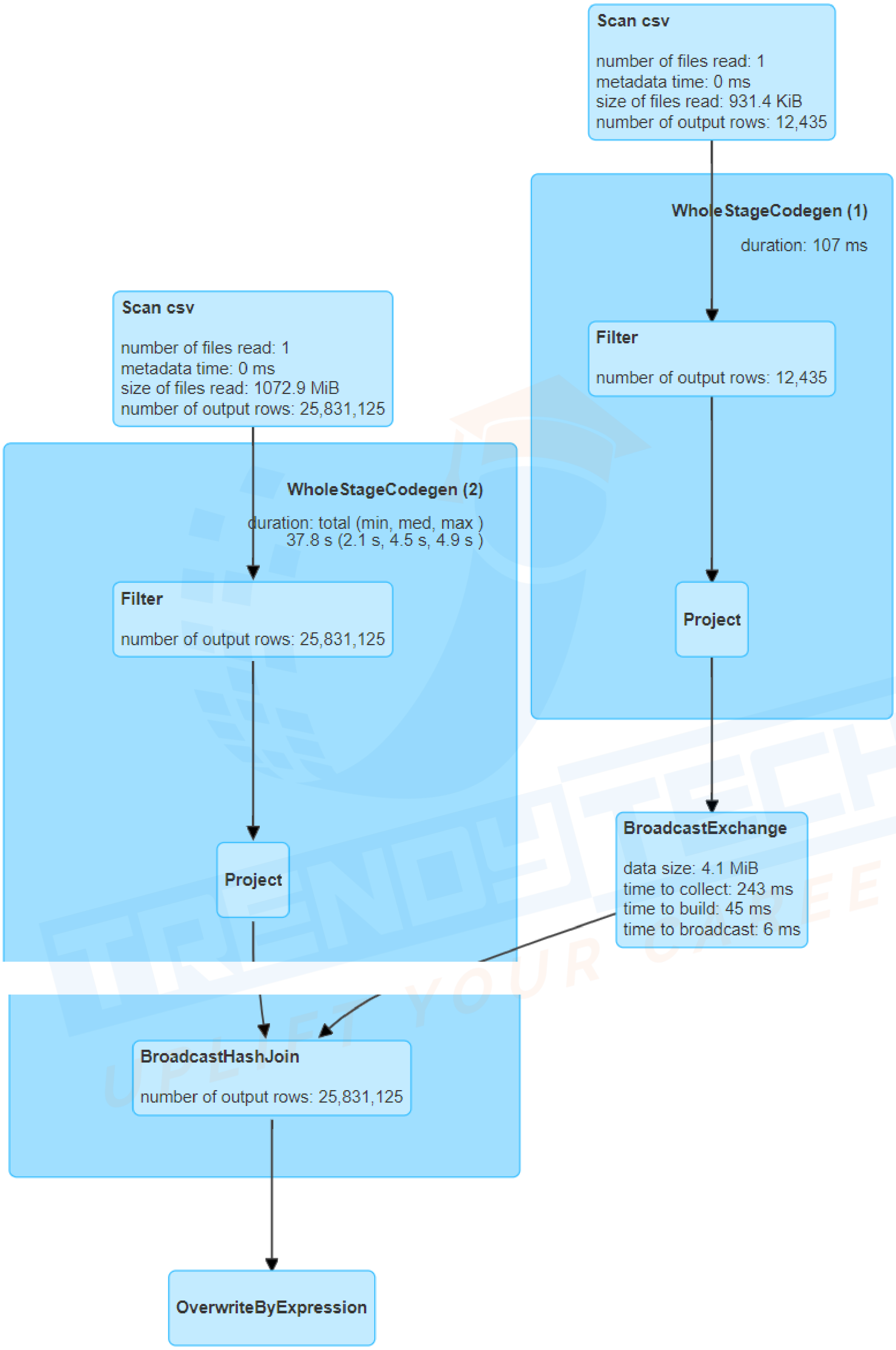
- One of the dataframe/ table is small enough to fit into the driver memory.
- The second Datafame that needs to be joined is large and partitions of this dataframe are distributed across multiple executors in the cluster.

Example :

```
orders_df.join(customers_df, orders_df.cust_id == customers_df.customerid, "inner").write.format("noop").mode("overwrite").save()
```

Key Points :

- Customers dataframe is small and can be broadcasted across executors containing partitions of the large dataframe - Orders.
- Spark by default tries to optimize the above join operation by using broadcast hash join as depicted in the DAG below:


Scan csv
number of files read: 1
metadata time: 0 ms
size of files read: 931.4 KiB
number of output rows: 12,435
WholeStageCodegen (1)
duration: 107 ms
Filter
number of output rows: 12,435
Project
Scan csv
number of files read: 1
metadata time: 0 ms
size of files read: 1072.9 MiB
number of output rows: 25,831,125
WholeStageCodegen (2)
duration: total (min, med, max)
37.8 s (2.1 s, 4.5 s, 4.9 s)
Filter
number of output rows: 25,831,125
Project
BroadcastExchange
data size: 4.1 MiB
time to collect: 243 ms
time to build: 45 ms
time to broadcast: 6 ms
BroadcastHashJoin
number of output rows: 25,831,125
OverwriteByExpression

Normal Shuffle-Sort-Merge join

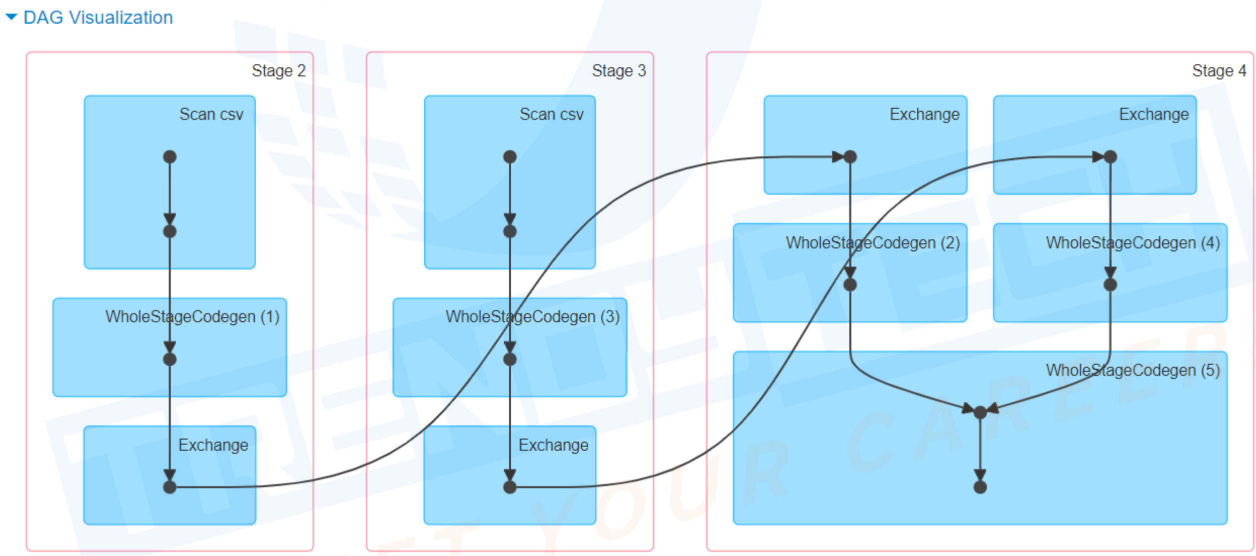
- Spark by default goes for broadcast join when one of the dataframes is small.
- In order to visualize how a normal shuffle-sort-merge join works, broadcast join has to be disabled by setting the following configuration -

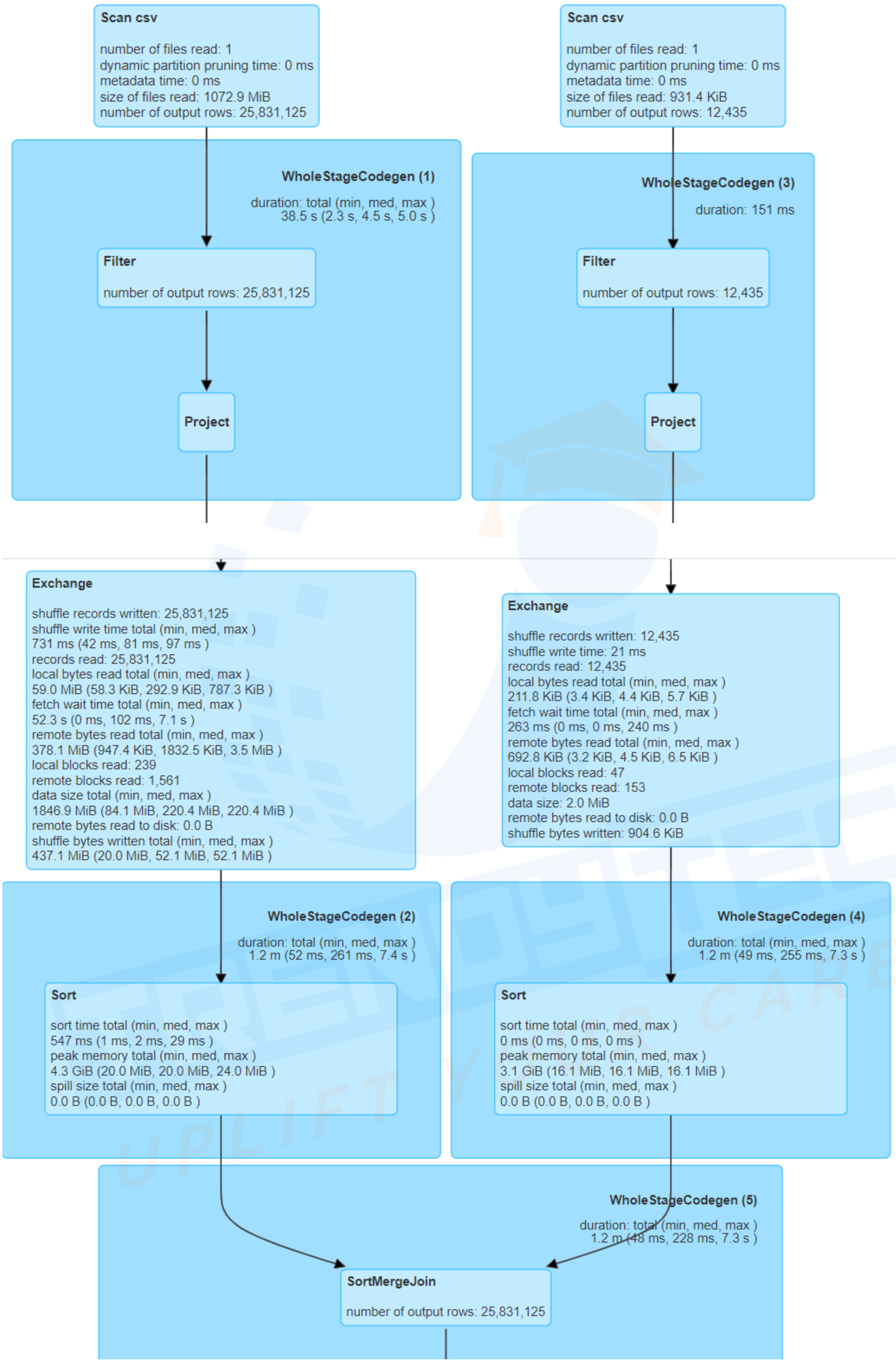
```
spark.conf.set('spark.sql.autoBroadcastJoinThreshold', '-1')
```

- Then we execute the join operation of orders and customers

```
orders_df.join(customers_df, orders_df.cust_id == customers_df.customerid, "inner").write.format("noop").mode("overwrite").save()
```

- DAG of Normal Join after disabling the Broadcast Join




Scan csv
number of files read: 1
dynamic partition pruning time: 0 ms
metadata time: 0 ms
size of files read: 1072.9 MiB
number of output rows: 25,831,125
Scan csv
number of files read: 1
dynamic partition pruning time: 0 ms
metadata time: 0 ms
size of files read: 931.4 KiB
number of output rows: 12,435
WholeStageCodegen (1)
duration: total (min, med, max)
38.5 s (2.3 s, 4.5 s, 5.0 s)
Filter
number of output rows: 25,831,125
Project
WholeStageCodegen (3)
duration: 151 ms
Filter
number of output rows: 12,435
Project
Exchange
shuffle records written: 25,831,125
shuffle write time total (min, med, max)
731 ms (42 ms, 81 ms, 97 ms)
records read: 25,831,125
local bytes read total (min, med, max)
59.0 MiB (58.3 KiB, 292.9 KiB, 787.3 KiB)
fetch wait time total (min, med, max)
52.3 s (0 ms, 102 ms, 7.1 s)
remote bytes read total (min, med, max)
378.1 MiB (947.4 KiB, 1832.5 KiB, 3.5 MiB)
local blocks read: 239
remote blocks read: 1,561
data size total (min, med, max)
1846.9 MiB (84.1 MiB, 220.4 MiB, 220.4 MiB)
remote bytes read to disk: 0.0 B
shuffle bytes written total (min, med, max)
437.1 MiB (20.0 MiB, 52.1 MiB, 52.1 MiB)
Exchange
shuffle records written: 12,435
shuffle write time: 21 ms
records read: 12,435
local bytes read total (min, med, max)
211.8 KiB (3.4 KiB, 4.4 KiB, 5.7 KiB)
fetch wait time total (min, med, max)
263 ms (0 ms, 0 ms, 240 ms)
remote bytes read total (min, med, max)
692.8 KiB (3.2 KiB, 4.5 KiB, 6.5 KiB)
local blocks read: 47
remote blocks read: 153
data size: 2.0 MiB
remote bytes read to disk: 0.0 B
shuffle bytes written: 904.6 KiB
WholeStageCodegen (2)
duration: total (min, med, max)
1.2 m (52 ms, 261 ms, 7.4 s)
Sort
sort time total (min, med, max)
547 ms (1 ms, 2 ms, 29 ms)
peak memory total (min, med, max)
4.3 GiB (20.0 MiB, 20.0 MiB, 24.0 MiB)
spill size total (min, med, max)
0.0 B (0.0 B, 0.0 B, 0.0 B)
WholeStageCodegen (4)
duration: total (min, med, max)
1.2 m (49 ms, 255 ms, 7.3 s)
Sort
sort time total (min, med, max)
0 ms (0 ms, 0 ms, 0 ms)
peak memory total (min, med, max)
3.1 GiB (16.1 MiB, 16.1 MiB, 16.1 MiB)
spill size total (min, med, max)
0.0 B (0.0 B, 0.0 B, 0.0 B)
WholeStageCodegen (5)
duration: total (min, med, max)
1.2 m (48 ms, 228 ms, 7.3 s)
SortMergeJoin
number of output rows: 25,831,125

Joins

Most common Join Types are :

1. Inner
2. Left Outer
3. Right Outer
4. Full Outer

Example :

Orders

order_id	customer_id	order_status
101	1	Closed
102	2	Complete
103	3	Pending
104	4	Closed

Customers

customer_id	city
1	Bangalore
2	Pune
5	Mumbai

Consider the Join column is customer_id

Inner Join

Gives matching records from both the tables.

Result - 1 , 2

Left Outer Join

Gives all the matching records + non-matching records from the left table.

Result - 1 , 2 (Left table details + Right table details)

3 , 4 (Left table details + NULL)

Right Outer Join

Gives all the matching records + non-matching records from the right table.

Result - 1 , 2 (Left table details + Right table details)

5 (Right table details + NULL)

Full Outer Join

Gives all the matching records + non-matching records from the left table. Union of Left & Right Outer Joins

Result - 1 , 2 (Left table details + Right table details)

3 , 4 (Left table details + NULL)

5 (Right table details + NULL)

Key Points:

- Broadcast join is not possible in case of Right Outer Join for the above scenario.
- Broadcast join is not possible in case of Full Outer Join as it is an union of Left and Right Outer Join.

Converting dataframes to Spark SQL tables to query the data using Spark SQL

Creating Spark Session

```
from pyspark.sql import SparkSession

spark = SparkSession. \
    builder. \
    config('spark.shuffle.useOldFetchProtocol', 'true'). \
    config("spark.sql.warehouse.dir", "/user/{username}/warehouse"). \
    enableHiveSupport(). \
    master('yarn'). \
    getOrCreate()
```

Orders Schema

```
orders_schema = "order_id long, order_date string, cust_id long, order_status string"
```

Creating Orders Dataframe

```
orders_df = spark.read \
.format("csv") \
.schema(orders_schema) \
.load("/public/trendytech/orders/orders_1gb.csv")
```

```
customers_schema = "customerid long, customer_fname string, customer_lname string, username string, password string, address string, city string, state string, pincod
```

Creating Customers Dataframe

```
customers_df = spark.read \
.format("csv") \
.schema(customers_schema) \
.load("/public/trendytech/retail_db/customers")
```

Join on Dataframes

```
orders_df.join(customers_df, orders_df.cust_id == customers_df.customerid, "inner").write.format("noop").mode("overwrite").save()
```

Creating Spark SQL table from orders dataframe

```
orders_df.createOrReplaceTempView("orders")
```

Creating Spark SQL table from customers dataframe

```
customers_df.createOrReplaceTempView("customers")
```

Join on Spark SQL tables

```
spark.sql("select * from orders inner join customers on orders.cust_id == customers.customerid").write.format("noop").mode("overwrite").save()
```

Partition Skew

Partition skew occurs when one of the partitions holds relatively more data as compared to the rest of the partitions.

```
orders_new_df = spark.read \
.format("csv") \
.schema(orders_schema) \
.load("/public/trendytech/retail_db/ordersnew")
```

```
orders_new_df.groupBy("order_status").count().collect()
```

```
[Row(order_status='PENDING_PAYMENT', count=5636250),
 Row(order_status='COMPLETE', count=46008801),
 Row(order_status='ON_HOLD', count=1424250),
 Row(order_status='PAYMENT_REVIEW', count=273375),
 Row(order_status='PROCESSING', count=3103125),
 Row(order_status='CLOSED', count=2833500),
 Row(order_status='SUSPECTED_FRAUD', count=584250),
 Row(order_status='PENDING', count=2853750),
 Row(order_status='CANCELED', count=535500)]
```

Partition Skew - COMPLETE has more data than other order_status

- Even though there are 200 partitions created after shuffling in case of wide transformation, all the records with order_status 'COMPLETE' will still be moved to one single partition.
- When any transformation is performed on a dataframe with partition skew, one of the tasks will take exceedingly more time to complete and could also lead to out of memory error.
- Partition skew will also lead to reduced parallelism.
- In order to overcome the partition skew problem, a process called Salting can be used.
- Salting is a process which involves adding random numbers to the dominating key causing the skew and generating multiple unique keys for a single key.

3 common problem use cases

1. Wide transformations that lead to 200 shuffle partitions but only few of the partitions have data.
2. A dominating key leading to Partition Skew.
3. Joining dataframes on distinct keys of one of the dataframe.

Key Points :

- In the earlier versions, Spark was not able to analyse the runtime statistics to improve the query performance by providing better optimization options.
- Therefore, it would default to a time consuming shuffle-sort-merge join with 200 shuffle partitions.
- However, this behaviour of spark was changed from Spark version 3, wherein it can now analyse the runtime statistics.

Adaptive Query Execution (AQE)

Is a feature that is available from Spark Major Version 3.0 onwards. This feature provides the following benefits to improve query performance.

1. **Dynamically Coalescing the number of shuffle partitions**
2. **Dynamically handling Partition Skew**
3. **Dynamically Switching Join Strategies**

   Note: The above 3 enhancements solve the 3 common problem use cases discussed earlier with older versions of spark.

Working of AQE

1. During execution, Spark calculates and analyses the statics at Run-time
2. Based on the calculation it derives some insights like
   - Number of Records
   - Size of Data
   - Minimum & Maximum of each column
   - Number of occurrences of each key …

How to check if AQE is enabled

```
spark.conf.get("spark.sql.adaptive.enabled")
```

```
'false'
```

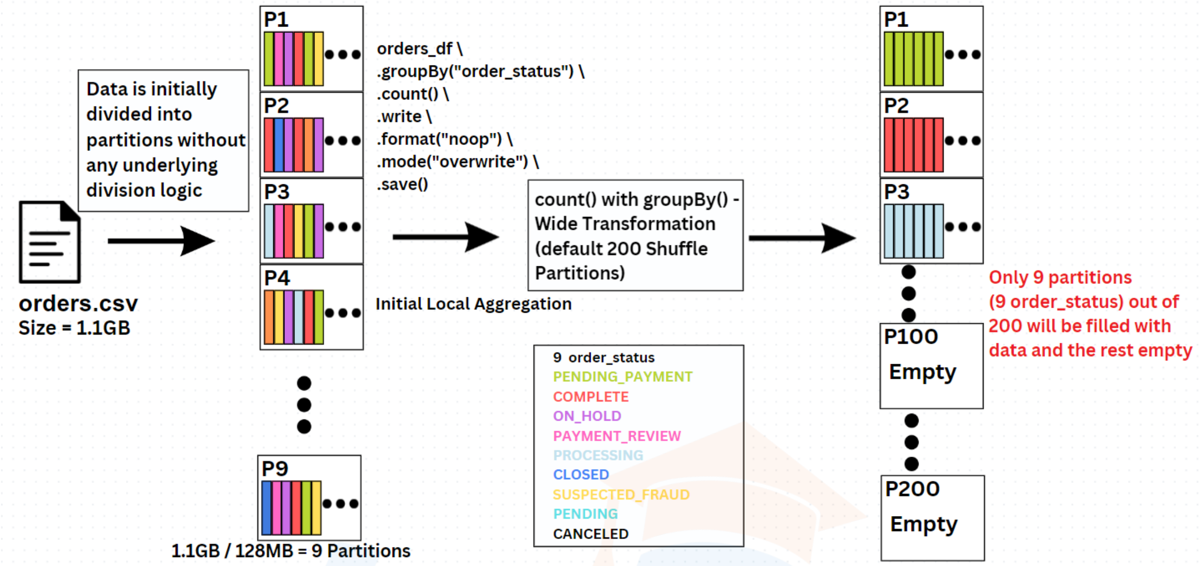
```
spark.conf.set("spark.sql.adaptive.enabled",True)
```

```
spark.conf.get("spark.sql.adaptive.enabled")
```

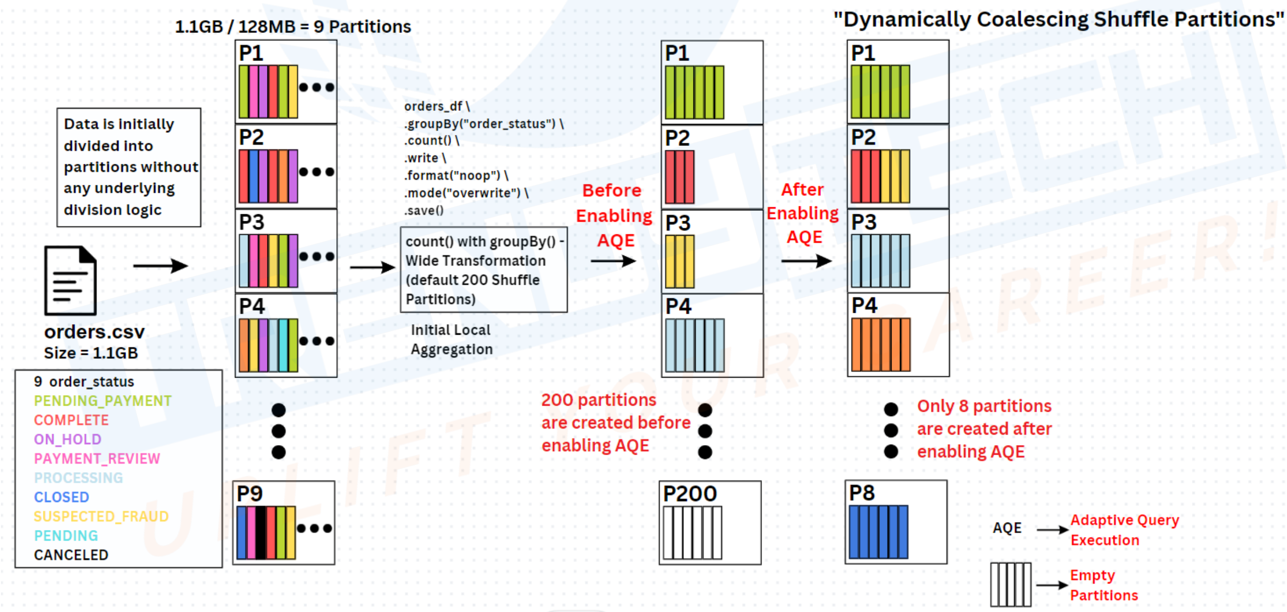
```
'true'
```

By default AQE is disabled in Spark 3.0 and has to be enabled. However, with versions 3.2 onwards, it is enabled by default.

Consider the following example orders dataframe

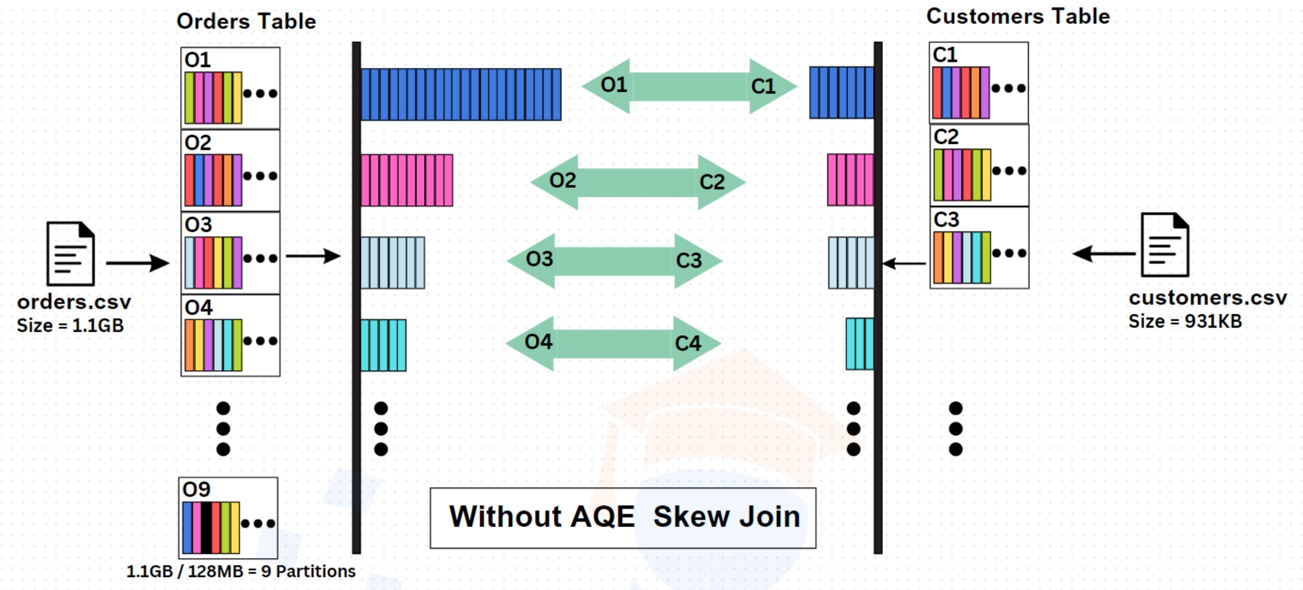


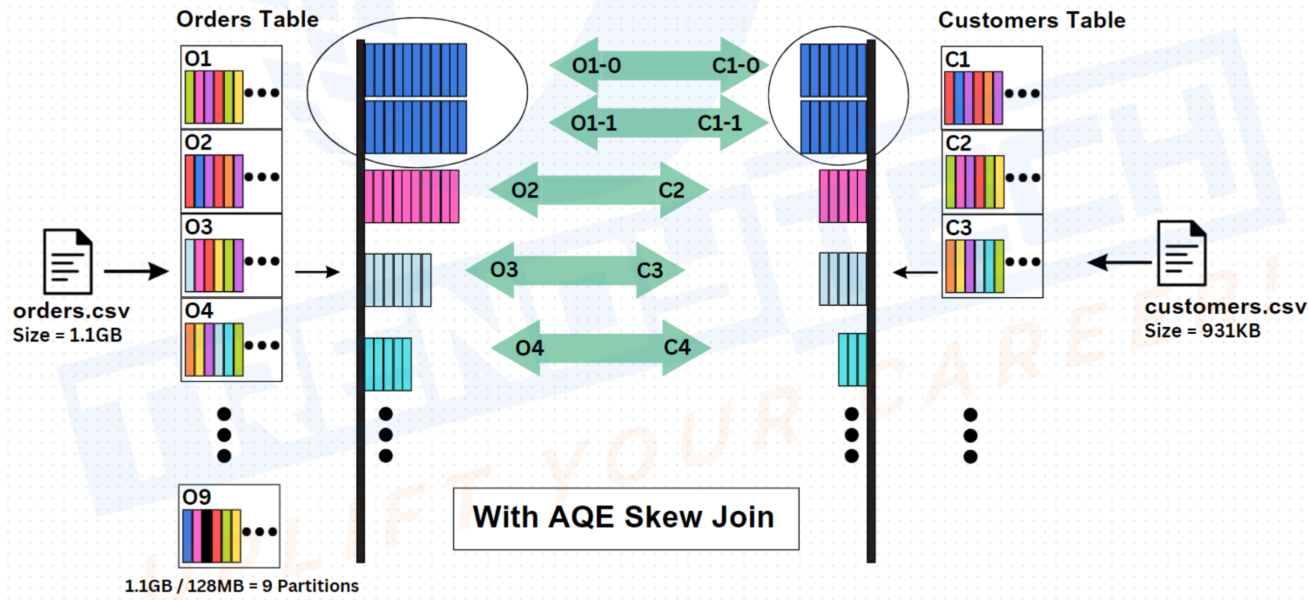
Demonstration of Dynamically Coalescing Shuffle Partitions



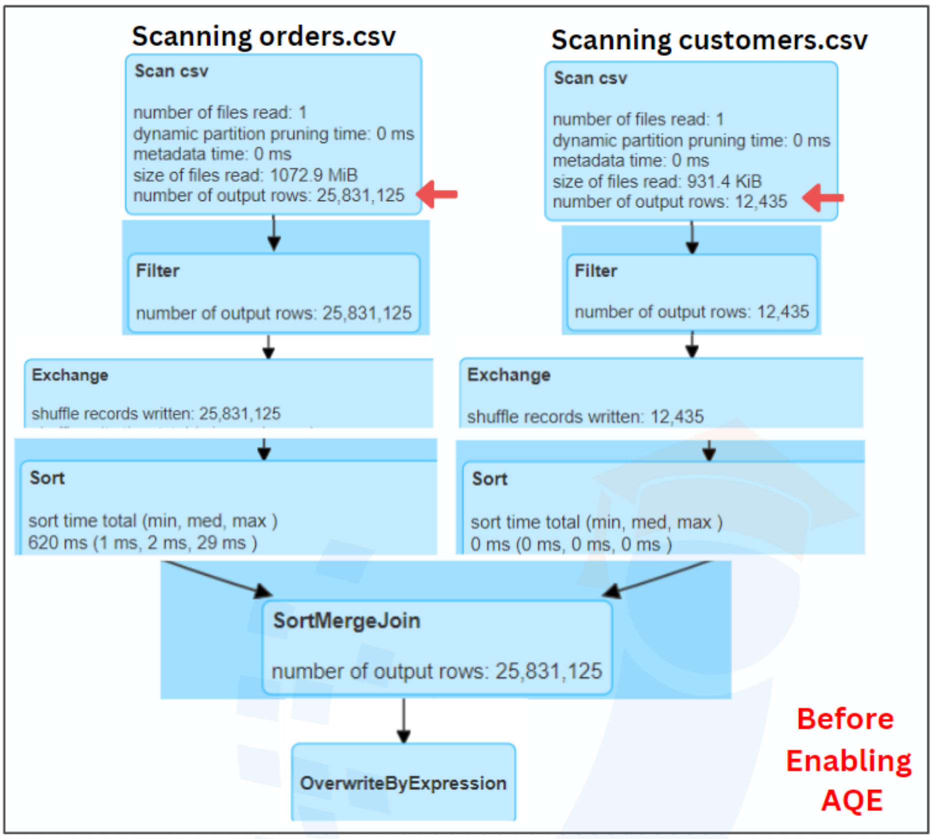
NOT FOR DISTRIBUTION ©Sumit Mittal www.trendytech.in

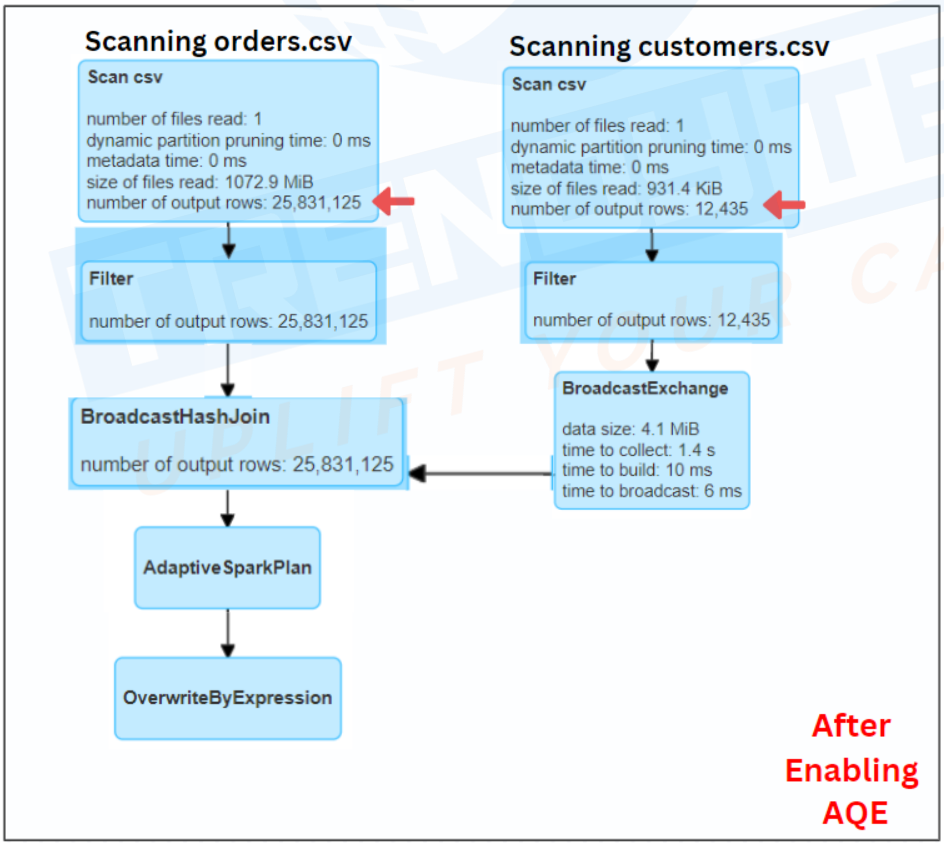
Demonstration of Dynamically handling Partition Skew





Demonstration of Dynamically Switching Join Strategies





Consider the following Example dataframes with 20 records each

Orders Dataframe

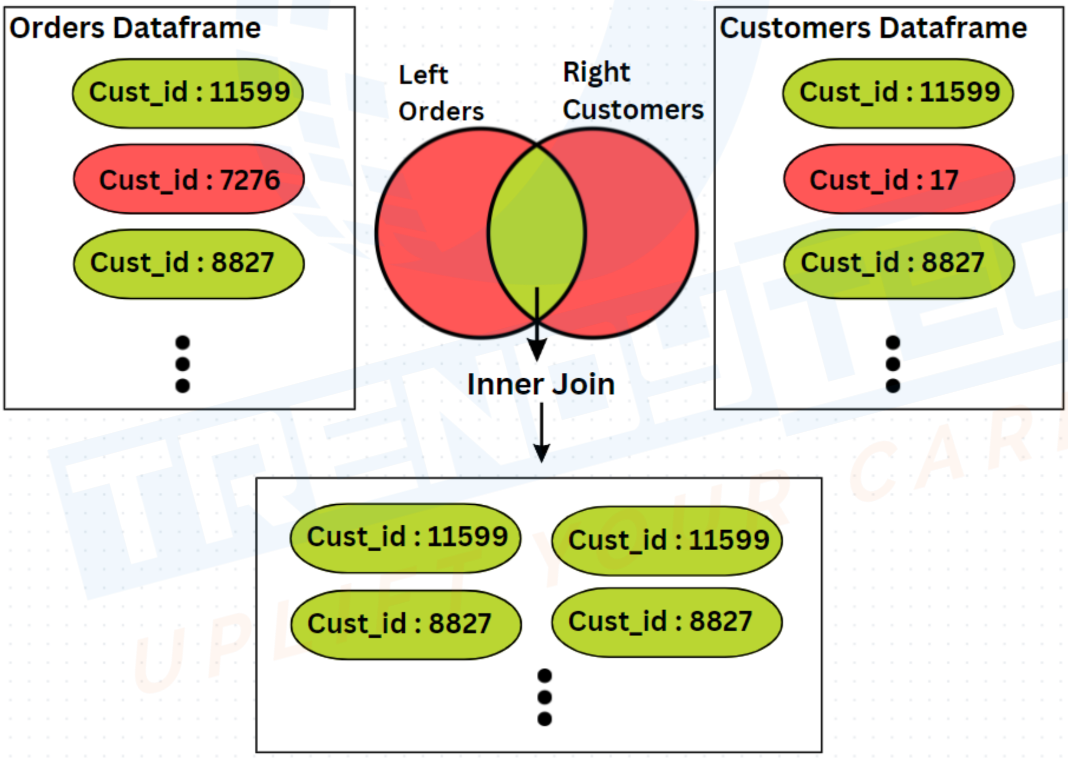
```
order_id, order_date, cust_id, order_status
1,2013-07-25 00:00:00.0,11599,CLOSED
2,2013-07-25 00:00:00.0,256,PENDING_PAYMENT
3,2013-07-25 00:00:00.0,12111,COMPLETE
4,2013-07-25 00:00:00.0,8827,CLOSED
5,2013-07-25 00:00:00.0,11318,COMPLETE
6,2013-07-25 00:00:00.0,7130,COMPLETE
7,2013-07-25 00:00:00.0,4530,COMPLETE
8,2013-07-25 00:00:00.0,2911,PROCESSING
9,2013-07-25 00:00:00.0,5657,PENDING_PAYMENT
10,2013-07-25 00:00:00.0,5648,PENDING_PAYMENT
11,2013-07-25 00:00:00.0,918,PAYMENT_REVIEW
12,2013-07-25 00:00:00.0,1837,CLOSED
13,2013-07-25 00:00:00.0,9149,PENDING_PAYMENT
14,2013-07-25 00:00:00.0,9842,PROCESSING
15,2013-07-25 00:00:00.0,2568,COMPLETE
16,2013-07-25 00:00:00.0,7276,PENDING_PAYMENT
17,2013-07-25 00:00:00.0,2667,COMPLETE
18,2013-07-25 00:00:00.0,1205,CLOSED
19,2013-07-25 00:00:00.0,2667,PENDING_PAYMENT
20,2013-07-25 00:00:00.0,1205,PROCESSING
```

Customers Dataframe

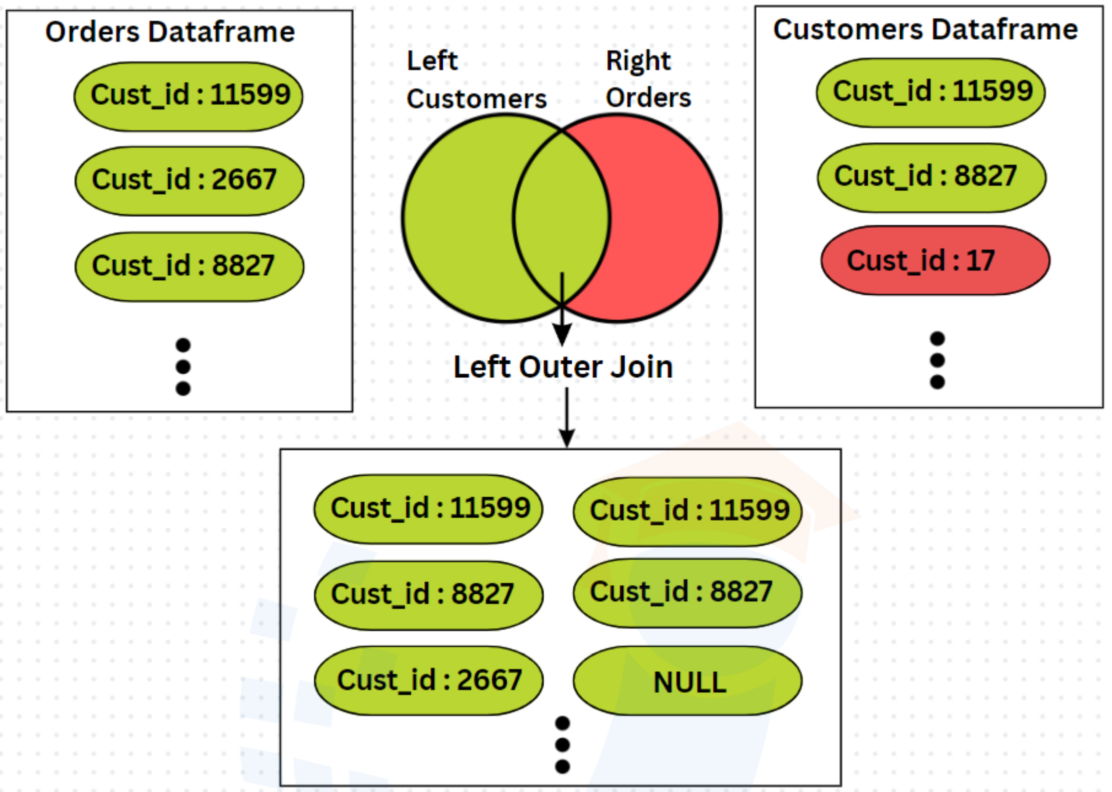
```
cust_id, cust_fname, cust_lame, cust_email, cust_password, cust_street, cust_city, cust_state, cust_zipcode
11599,Richard,Hernandez,XXXXXXXXX,XXXXXXXXX,6303 Heather Plaza,Brownsville,TX,78521
256,Mary,Barrett,XXXXXXXXX,XXXXXXXXX,9526 Noble Embers Ridge,Littleton,CO,80126
12111,Ann,Smith,XXXXXXXXX,XXXXXXXXX,3422 Blue Pioneer Bend,Caguas,PR,00725
8827,Mary,Jones,XXXXXXXXX,XXXXXXXXX,8324 Little Common,San Marcos,CA,92069
11318,Robert,Hudson,XXXXXXXXX,XXXXXXXXX,"10 Crystal River Mall ",Caguas,PR,00725
7130,Mary,Smith,XXXXXXXXX,XXXXXXXXX,3151 Sleepy Quail Promenade,Passaic,NJ,07055
4530,Melissa,Wilcox,XXXXXXXXX,XXXXXXXXX,9453 High Concession,Caguas,PR,00725
2911,Megan,Smith,XXXXXXXXX,XXXXXXXXX,3047 Foggy Forest Plaza,Lawrence,MA,01841
5657,Mary,Perez,XXXXXXXXX,XXXXXXXXX,3616 Quaking Street,Caguas,PR,00725
5648,Melissa,Smith,XXXXXXXXX,XXXXXXXXX,8598 Harvest Beacon Plaza,Stafford,VA,22554
11,Mary,Huffman,XXXXXXXXX,XXXXXXXXX,3169 Stony Woods,Caguas,PR,00725
12,Christopher,Smith,XXXXXXXXX,XXXXXXXXX,5594 Jagged Embers By-pass,San Antonio,TX,78227
13,Mary,Baldwin,XXXXXXXXX,XXXXXXXXX,7922 Iron Oak Gardens,Caguas,PR,00725
14,Katherine,Smith,XXXXXXXXX,XXXXXXXXX,5666 Hazy Pony Square,Pico Rivera,CA,90660
15,Jane,Luna,XXXXXXXXX,XXXXXXXXX,673 Burning Glen,Fontana,CA,92336
16,Tiffany,Smith,XXXXXXXXX,XXXXXXXXX,6651 Iron Port,Caguas,PR,00725
17,Mary,Robinson,XXXXXXXXX,XXXXXXXXX,1325 Noble Pike,Taylor,MI,48180
18,Robert,Smith,XXXXXXXXX,XXXXXXXXX,2734 Hazy Butterfly Circle,Martinez,CA,94553
19,Stephanie,Mitchell,XXXXXXXXX,XXXXXXXXX,3543 Red Treasure Bay,Caguas,PR,00725
20,Mary,Ellis,XXXXXXXXX,XXXXXXXXX,4703 Old Route,West New York,NJ,07093
```

Different Join Types

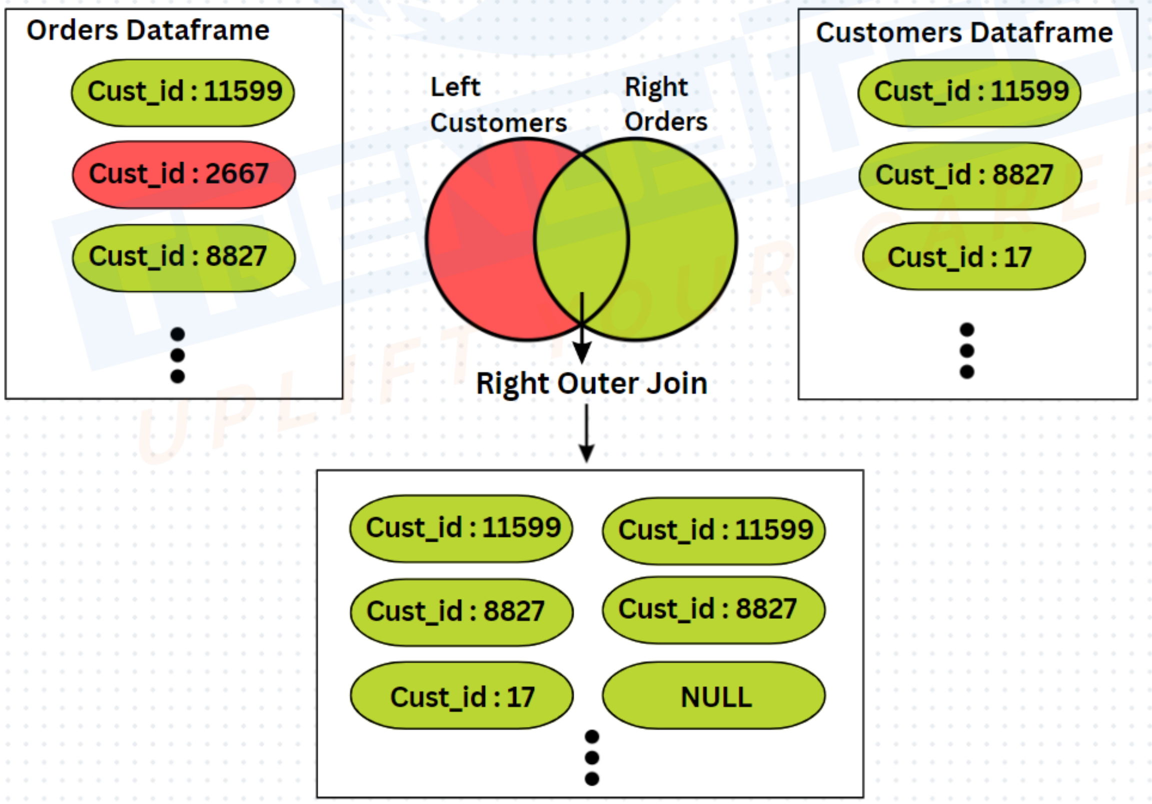
1. Inner Join



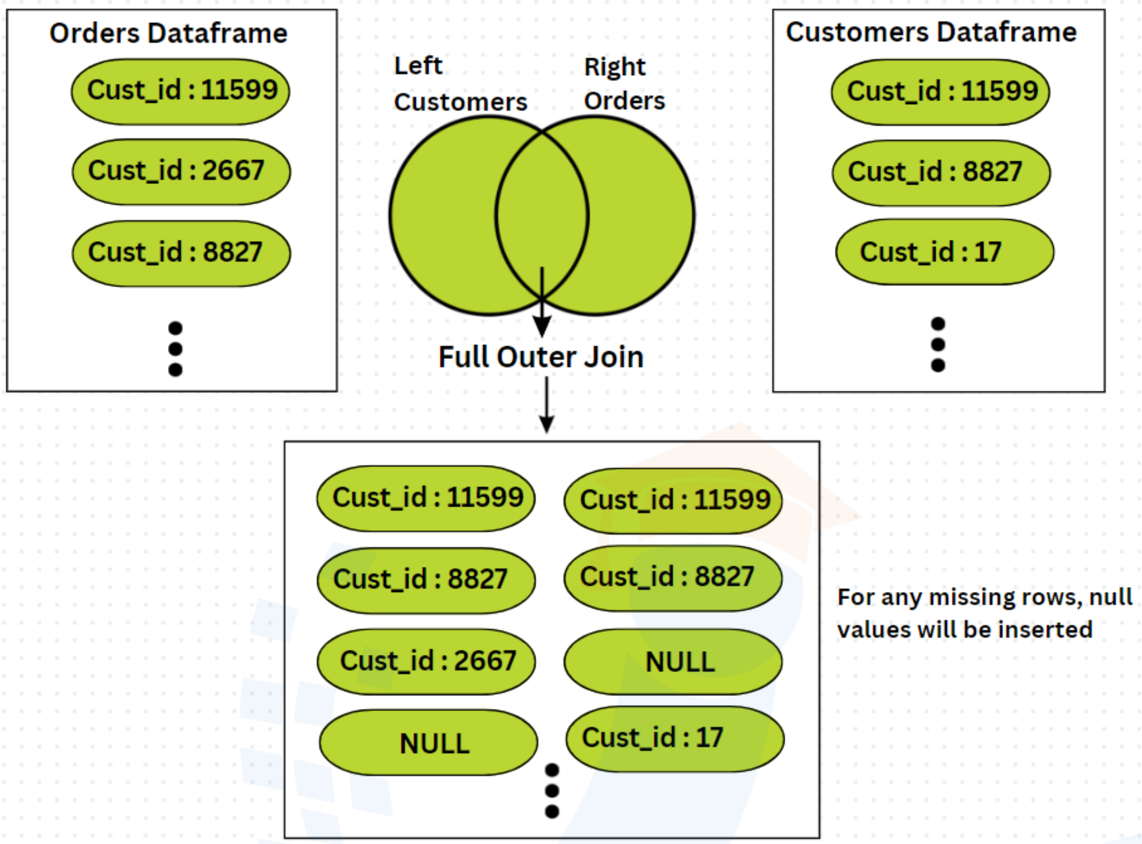
2. Left Outer Join / Left Join



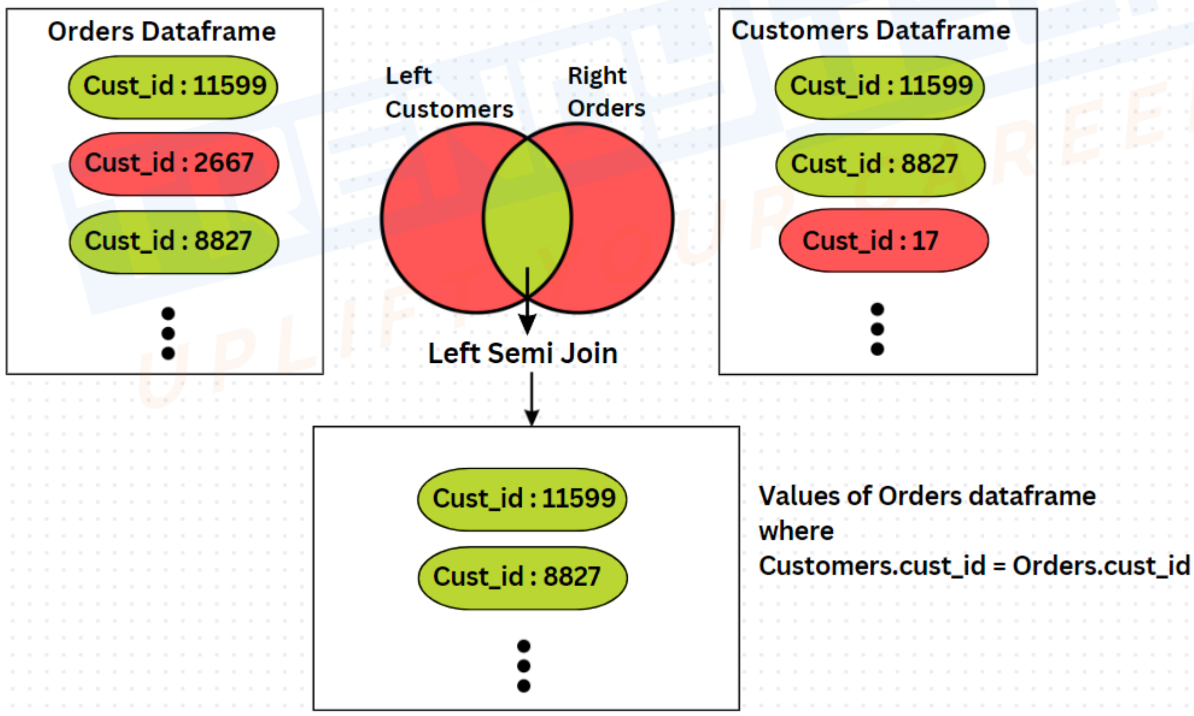
3. Right Outer Join / Right Join



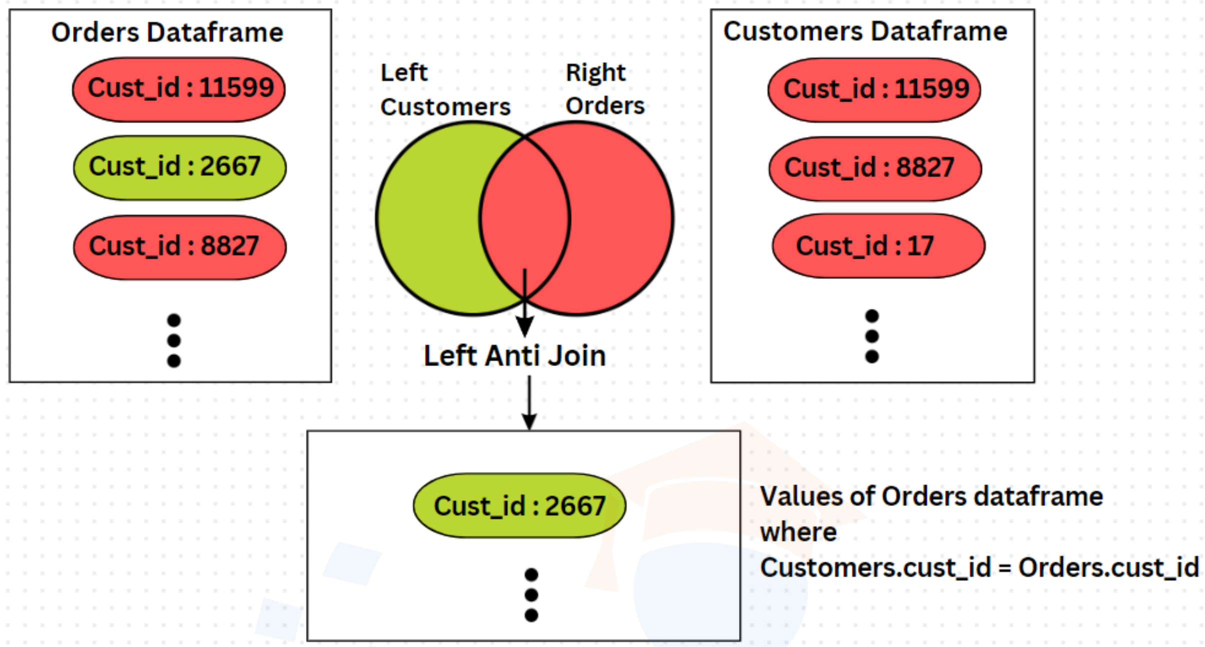
4. Full Join / Full Outer Join



5. Left Semi Join



6. **Left Anti Join**



Different Join Strategies

Broadcast :

This join involves -

a. One large dataframe partitioned and distributed across executors in the cluster.
b. One small dataframe that can fit into the driver memory.

The entire small dataframe is then broadcasted to all the executors containing partitions of the large dataframe.

This avoids shuffling data between the executors, which is an expensive operation.

Shuffle :

Every executor will interact with other executors to share the data partitions. Eventually after the Shuffle / data movement, all the records with the same join keys will then be moved to the same executor.

- Consider the following use case of orders and customers data

Orders Dataframe
Large Dataframe

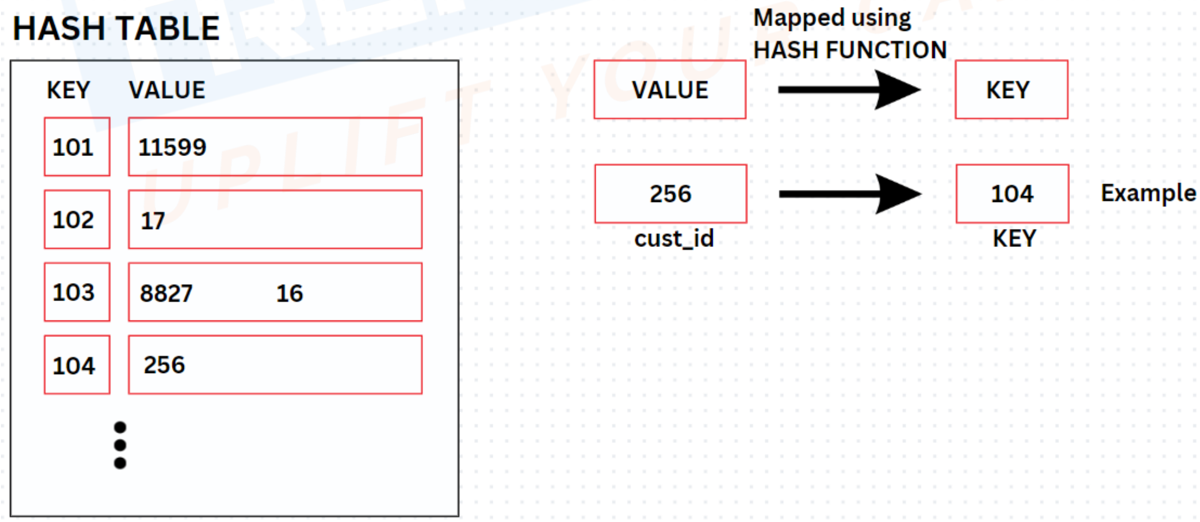
-Cannot fit into a single executor memory
-Partitioned and distributed across the cluster.

Customers Dataframe
Small Dataframe < 10MB

-Can fit into a single executor memory
-Spark creates a Hash Table for this small dataframe.

Hash Table

Hash Table Creation
-Hash Table is a Data Structure that will increase the search efficiency to O(1) i.e., Constant time.
-Hash Function is used to Map the Input Value(Join column value) to a Key

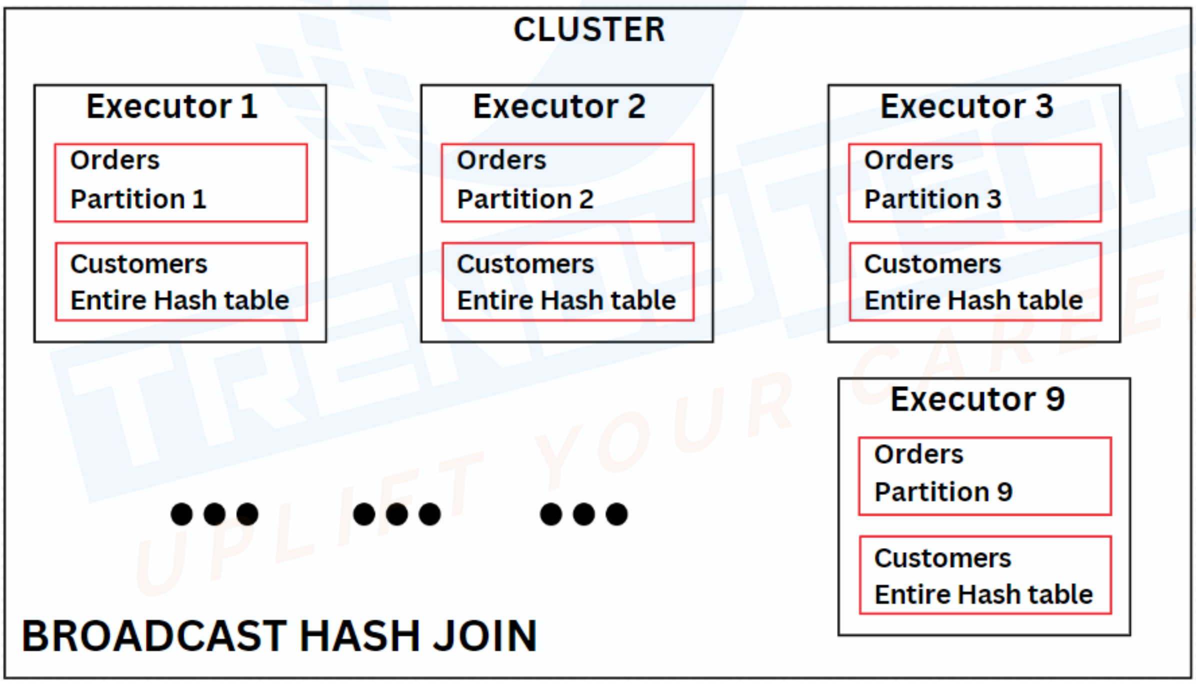


1. **Broadcast Hash Join**

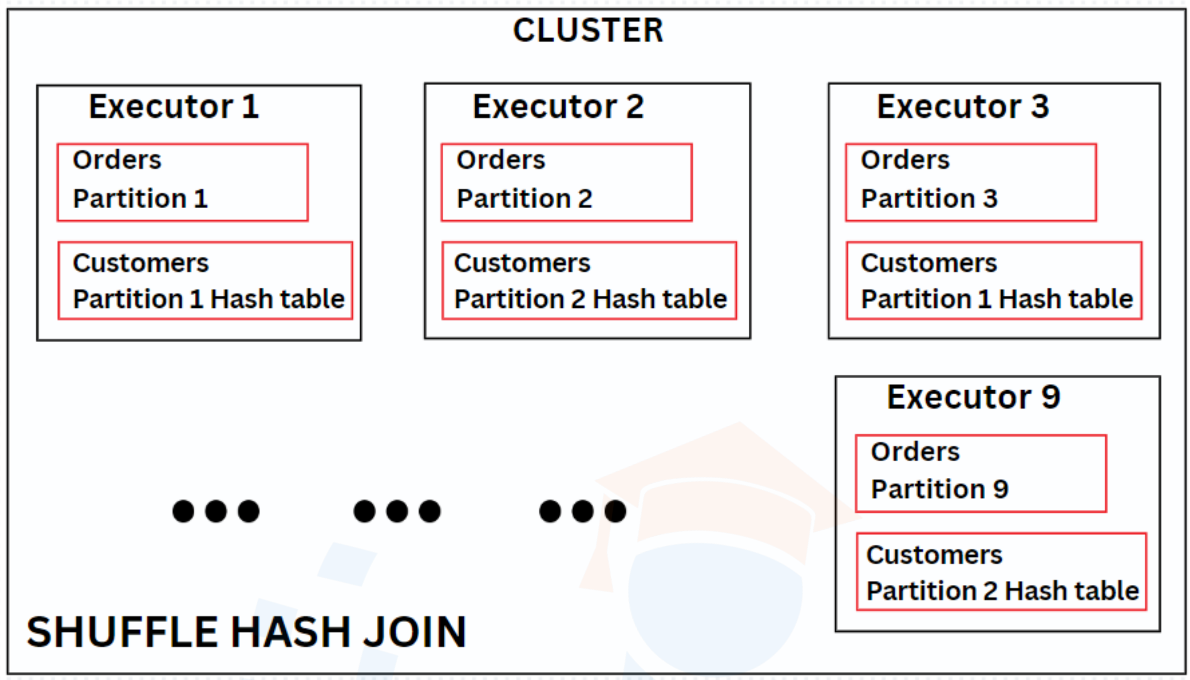
- Since Orders dataframe is large (~1.1GB) it is divided into 9 partitions and distributed among multiple executors on the cluster. (aka : PROBE Table)

  Note: 2 different partitions can be present on the same executor as well. This depends on the availability of resources

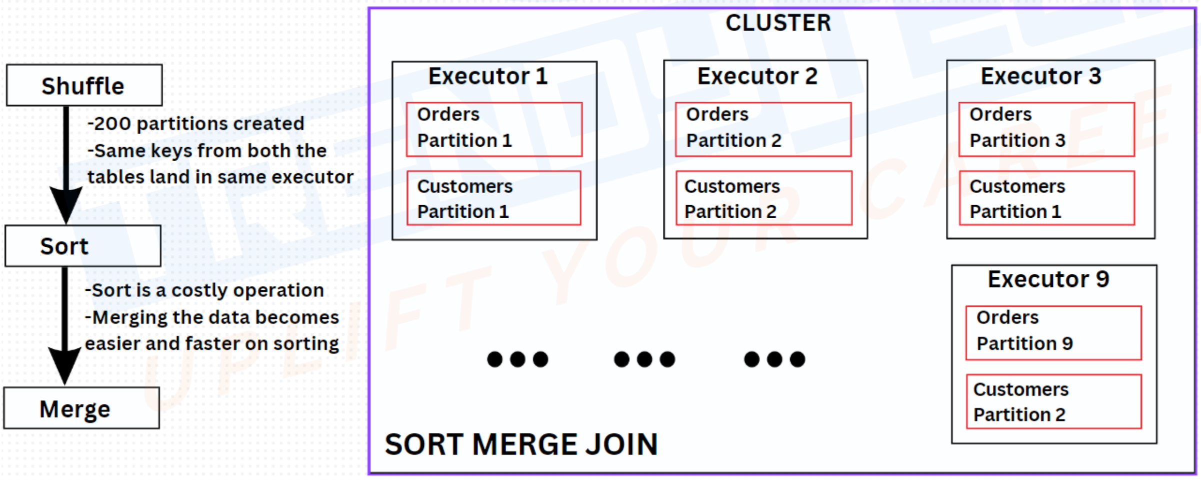
- The Customers dataframe is small (~938MB), therefore Spark creates a Hash Table for this small dataframe. (aka : HASH Table)
- The entire hash table of the customers dataframe is broadcasted to all the executors consisting of partitions of the large orders dataframe.



2. Shuffle Hash Join



3. Shuffle Sort Merge Join



Optimization of joining 2 Large tables : Bucketing

Consider orders dataset, one way of optimising the query performance while querying orders data is

Partitioning

- While creating the orders dataframe, it is partitioned based on the order_status as a partition column
- This will create 9 different folders, each folder will contain data records related to one order status.
- The queries will be more performant while querying the data based on the partition column "order_status"
- Only one folder will be searched to fetch the desired records based on the query as each folder is associated to one order_status.
- Rest of the folders will not be searched leading to significant performance gains. This process of searching only one folder and eliminating the other folders is called Partition **Pruning**.
- Partitioning helps in queries involving filtering operations.

Bucketing

- Suppose that the most frequently executed query is

  select * from orders where order_id = ' ### '

- For such queries, bucketing will be a better choice
- In case of bucketing, a fixed number of buckets have to be predefined and the data will be segregated into these buckets based on a **Hash function**.

```
orders_df.write \
.mode("overwrite") \
.bucketBy(8, "cust_id") \
.sortBy("cust_id") \
.option("path", "orderspath")
.saveAsTable("ordersschematable")
```

```
orders_df.write \
.mode("overwrite") \
.bucketBy(8, "cust_id") \
.sortBy("cust_id") \
.option("path", "orderspath") \
.saveAsTable("orderstable")
```

```
spark.sql("describe formatted orderstable").show(50,False)
```

```
+----------------------------+--------------------------------------------------------------+-------+
|col_name                    |data_type                                                     |comment|
+----------------------------+--------------------------------------------------------------+-------+
|order_id                    |bigint                                                        |null   |
|order_date                  |string                                                        |null   |
|cust_id                     |bigint                                                        |null   |
|order_status                |string                                                        |null   |
|                            |                                                              |       |
|# Detailed Table Information|                                                              |       |
|Database                    |default                                                       |       |
|Table                       |orderstable                                                   |       |
|Owner                       |itv006753                                                     |       |
|Created Time                |Sat Jul 22 04:39:28 EDT 2023                                  |       |
|Last Access                 |UNKNOWN                                                       |       |
|Created By                  |Spark 3.0.1                                                   |       |
|Type                        |EXTERNAL                                                      |       |
|Provider                    |parquet                                                       |       |
|Num Buckets                 |8                                                             |       |
|Bucket Columns              |[`cust_id`]                                                   |       |
|Sort Columns                |[`cust_id`]                                                   |       |
|Statistics                  |20197478 bytes                                                |       |
|Location                    |hdfs://m01.itversity.com:9000/user/itv006753/orderspath       |       |
|Serde Library               |org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe   |       |
|InputFormat                 |org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat |       |
|OutputFormat                |org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat|       |
+----------------------------+--------------------------------------------------------------+-------+
```

- Example : Both orders and customers tables were created with 8 buckets each.

On performing a join on these two tables, 8 tasks were launched as there were 8 buckets in each table.