What is a Sort Merge Join?

A Sort Merge Join (SMJ) is a type of join used by Spark when:

- Both input datasets are large
- Join keys are sortable
- No table is small enough to be broadcasted

How Sort Merge Join Works (Steps):
Scan and Filter: Spark scans both datasets and applies any WHERE filters.
Shuffle: Both tables are shuffled by the join key using hash partitioning.
Sort: Each partition is sorted on the join key.
Join: Spark performs a merge join on sorted data in each partition.

When to Use Sort Merge Join:

- Both tables are large and cannot be broadcasted
- Join keys are sortable
- You want to avoid memory issues with broadcasting

Characteristics of Sort Merge Join:

Feature	Description
Shuffling	Yes – Both tables are shuffled on join key
Sorting	Yes – Each side is sorted on the join key
Memory Usage	Moderate to High
Parallelism	Good, but can be expensive
Join Key Requirement	Must be sortable
Suitable For	Large-to-large joins where broadcasting is not possible

---

Configuration Tips:

- Use MERGE hint to force this join type.
- You can also disable broadcast joins to force Spark to pick Sort Merge Join:

spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)

Normal join: joined_df.explain

✅ WHAT YOU RAN:

You joined:

- A **big table** transactions_df (id, countrycode, amount)
- A **small table** countries_df (countrycode, countryname)

But Spark did **NOT** use a **broadcast join**.

❗ **Reason: You did not use broadcast() or spark.sql.autoBroadcastJoinThreshold was too small.**

joined_df.explain(True)

```
joined_df.explain(True)

== Parsed Logical Plan ==
'Join UsingJoin(Inner, [countrycode])
:- LogicalRDD [id#0L, countrycode#1, amount#2L], false
+- LogicalRDD [countrycode#6, countryname#7], false

== Analyzed Logical Plan ==
countrycode: string, id: bigint, amount: bigint, countryname: string
Project [countrycode#1, id#0L, amount#2L, countryname#7]
+- Join Inner, (countrycode#1 = countrycode#6)
   :- LogicalRDD [id#0L, countrycode#1, amount#2L], false
   +- LogicalRDD [countrycode#6, countryname#7], false

== Optimized Logical Plan ==
Project [countrycode#1, id#0L, amount#2L, countryname#7]
+- Join Inner, (countrycode#1 = countrycode#6)
   :- Filter isnotnull(countrycode#1)
   :  +- LogicalRDD [id#0L, countrycode#1, amount#2L], false
   +- Filter isnotnull(countrycode#6)
      +- LogicalRDD [countrycode#6, countryname#7], false

== Physical Plan ==
*(5) Project [countrycode#1, id#0L, amount#2L, countryname#7]
+- *(5) SortMergeJoin [countrycode#1], [countrycode#6], Inner
   :- *(2) Sort [countrycode#1 ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(countrycode#1, 200), ENSURE_REQUIREMENTS, [plan_id=93]
   :     +- *(1) Filter isnotnull(countrycode#1)
   :        +- *(1) Scan ExistingRDD[id#0L,countrycode#1,amount#2L]
   +- *(4) Sort [countrycode#6 ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(countrycode#6, 200), ENSURE_REQUIREMENTS, [plan_id=99]
         +- *(3) Filter isnotnull(countrycode#6)
            +- *(3) Scan ExistingRDD[countrycode#6,countryname#7]
```

Breakdown of Physical Plan

```
== Physical Plan ==
*(5) Project [countrycode#1, id#0L, amount#2L, countryname#7]                <-- [Step 5]
+- *(5) SortMergeJoin [countrycode#1], [countrycode#6], Inner                <-- [Step 5]
   :- *(2) Sort [countrycode#1 ASC NULLS FIRST], false, 0                    <-- [Step 2]
   :  +- Exchange hashpartitioning(countrycode#1, 200), ENSURE_REQUIREMENTS   <-- [Step 1]
   :     +- *(1) Filter isnotnull(countrycode#1)                             <-- [Step 1]
   :        +- *(1) Scan ExistingRDD[id#0L,countrycode#1,amount#2L]          <-- [Step 1]
   +- *(4) Sort [countrycode#6 ASC NULLS FIRST], false, 0                    <-- [Step 4]
      +- Exchange hashpartitioning(countrycode#6, 200), ENSURE_REQUIREMENTS  <-- [Step 3]
         +- *(3) Filter isnotnull(countrycode#6)                             <-- [Step 3]
            +- *(3) Scan ExistingRDD[countrycode#6,countryname#7]            <-- [Step 3]
```

✅ Step 1 — Scan & Filter Transactions Table

- Scan ExistingRDD[id#0L,countrycode#1,amount#2L]
  → Read raw data from transactions_df
- Filter isnotnull(countrycode#1)
  → Spark filters out rows with NULL countrycode

✅ Step 2 — Partition + Sort Transactions Table

- Exchange hashpartitioning(countrycode#1, 200)
  → Spark reshuffles transactions table using hash of countrycode
  → Default 200 partitions unless changed
- Sort [countrycode#1 ASC NULLS FIRST]
  → Needed for SortMergeJoin, ensures keys are sorted

✅ Step 3 — Scan & Filter Countries Table

- Scan ExistingRDD[countrycode#6,countryname#7]
  → Read raw data from countries_df
- Filter isnotnull(countrycode#6)
  → Remove nulls

✅ Step 4 — Partition + Sort Countries Table

- Exchange hashpartitioning(countrycode#6, 200)
  → Reshuffle based on countrycode
- Sort [countrycode#6 ASC NULLS FIRST]
  → Sort required for merge join

✅ Step 5 — Join & Project

- SortMergeJoin [countrycode#1], [countrycode#6], Inner
  → Both sides are sorted and partitioned by countrycode
  → Spark performs sort-merge join
- Project [countrycode#1, id#0L, amount#2L, countryname#7]
  → Select only desired columns for final output

Why Sorting Is Required for SMJ

A Sort-Merge Join (SMJ) works efficiently only when both tables are sorted on the join key (countrycode in your case). Spark must guarantee this before it can "merge" the two sides row by row.

🔄 What Happens During the Sort Phase?

For each table (left & right), Spark does:

1. Partitioning: First, Spark ensures both sides are partitioned (via shuffle) by the join key.
   - Example: Exchange hashpartitioning(countrycode, 200)
   - Ensures that rows with the same countrycode go to the same partition.
2. Sort Within Each Partition:
   Spark sorts the rows within each partition by the join key (countrycode).
   - This allows Spark to do a streamed merge similar to a sorted merge in merge sort.

🔧 Internally: Imagine these unsorted partitions:

Imagine these unsorted partitions:

Before sort:

```
less

Partition 1 (countrycode): [IN, FR, US, IN]
Partition 2 (countrycode): [FR, DE, US, IN]
```

After sort:

```
less

Partition 1: [FR, IN, IN, US]
Partition 2: [DE, FR, IN, US]
```

Now that each partition is **sorted**, Spark can efficiently do a **merge scan** across both datasets, like:

vbnet

Copy Edit

```
while (left.hasNext && right.hasNext):
    if left.key == right.key:
        emit join result
    elif left.key < right.key:
        advance left
    else:
        advance right
```

This avoids nested loop joins.

Physical Plan Code You Saw

text

```
:- *(2) Sort [countrycode#1 ASC NULLS FIRST], false, 0
+- Exchange hashpartitioning(countrycode#1, 200)
```

Spark first partitioned the data by hash(countrycode)

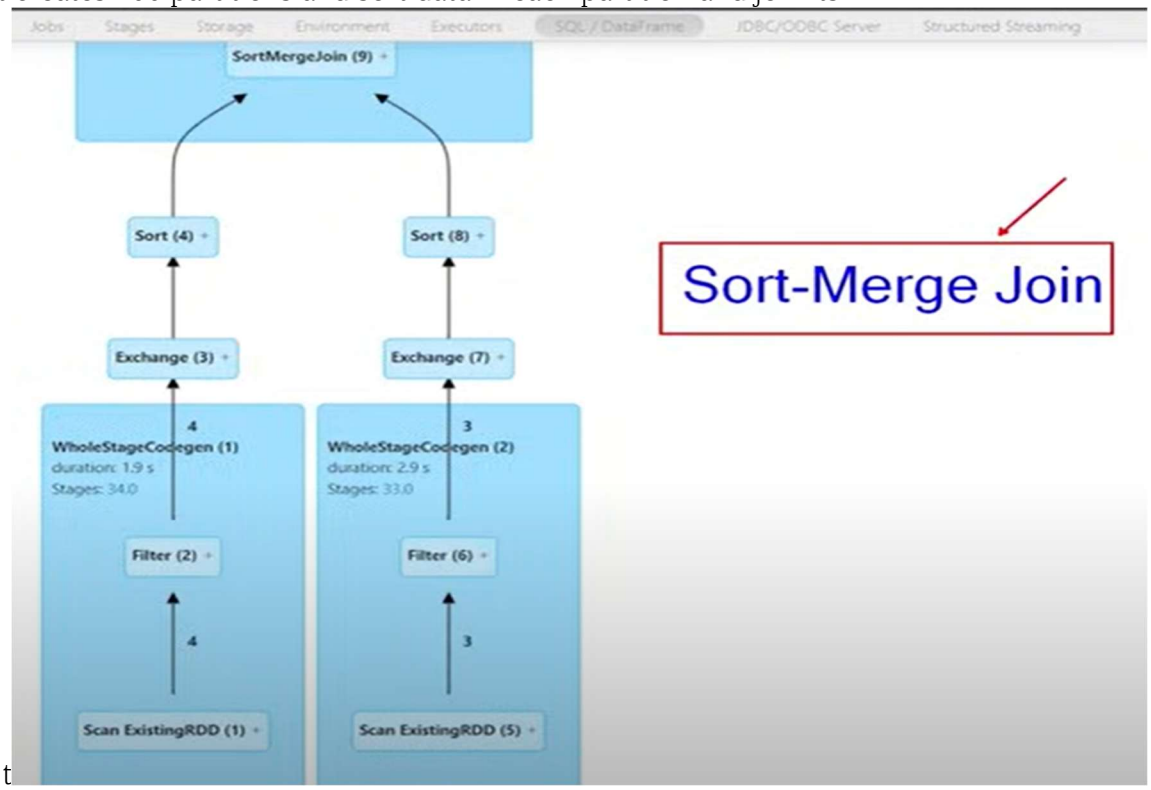
- Then within each partition, it sorted by countrycode ascending

Same happened on the other side (countries_df).

DAG diagram

Sort-merge join it is default join in spark merge join.

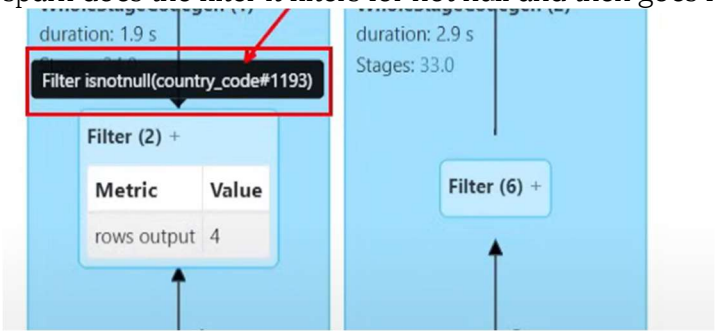
It creates 200 partitions and sort data in each partition and join its



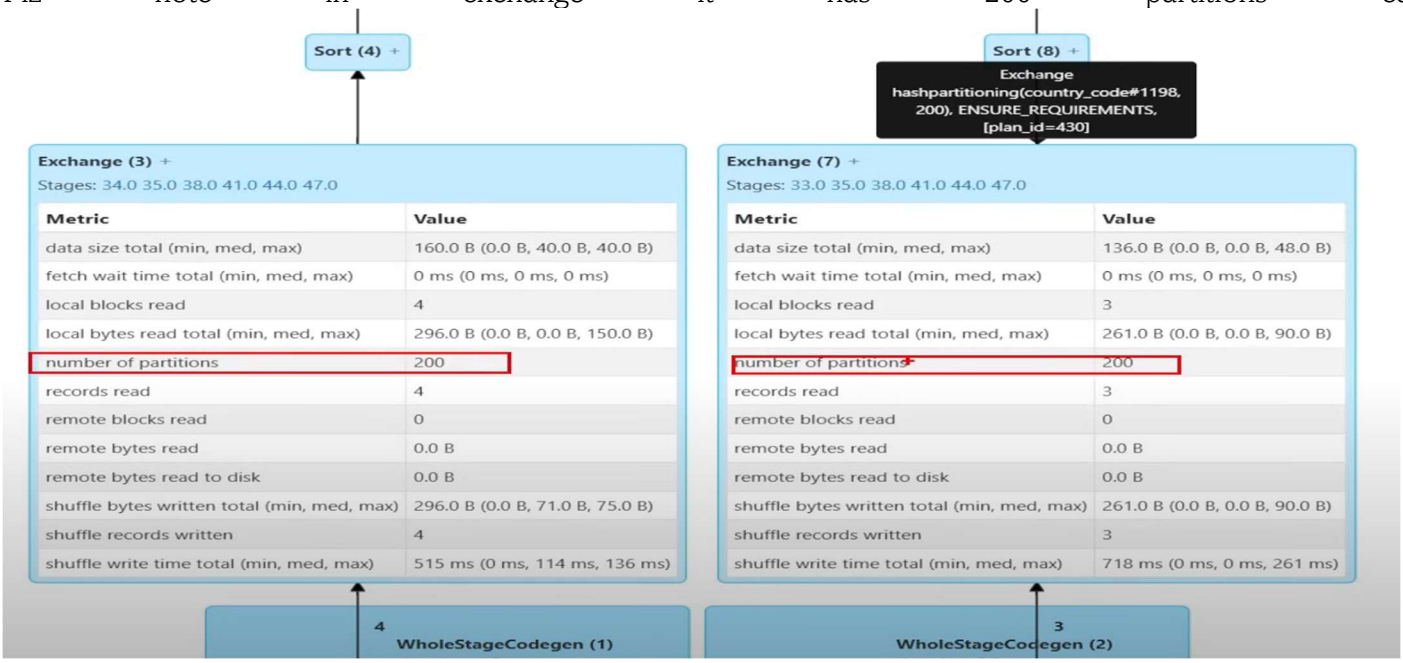
t

Scan existing rdd it has 4 rows existing rdd it has 5 rows

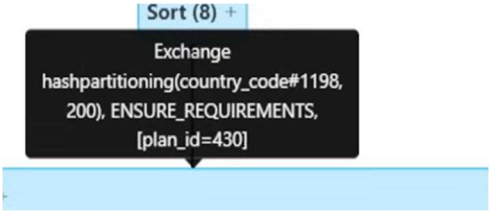
spark does the filter it filters for not null and then goes for exchange.



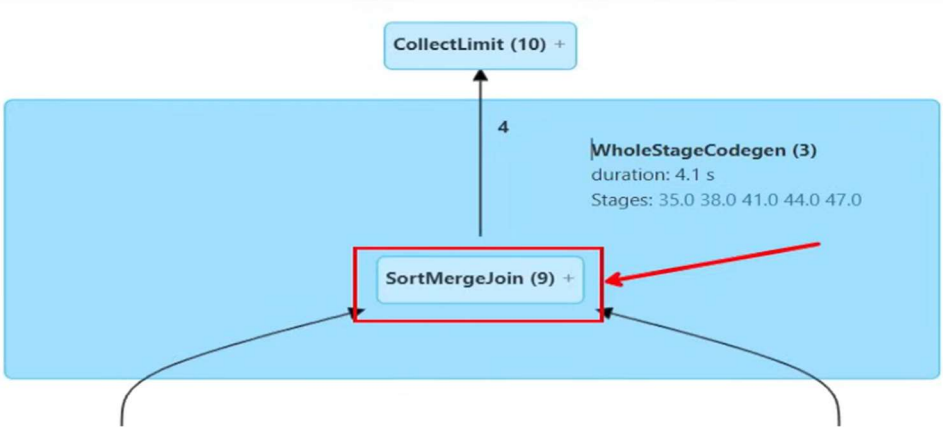
Plz note in exchange it has 200 partitions each



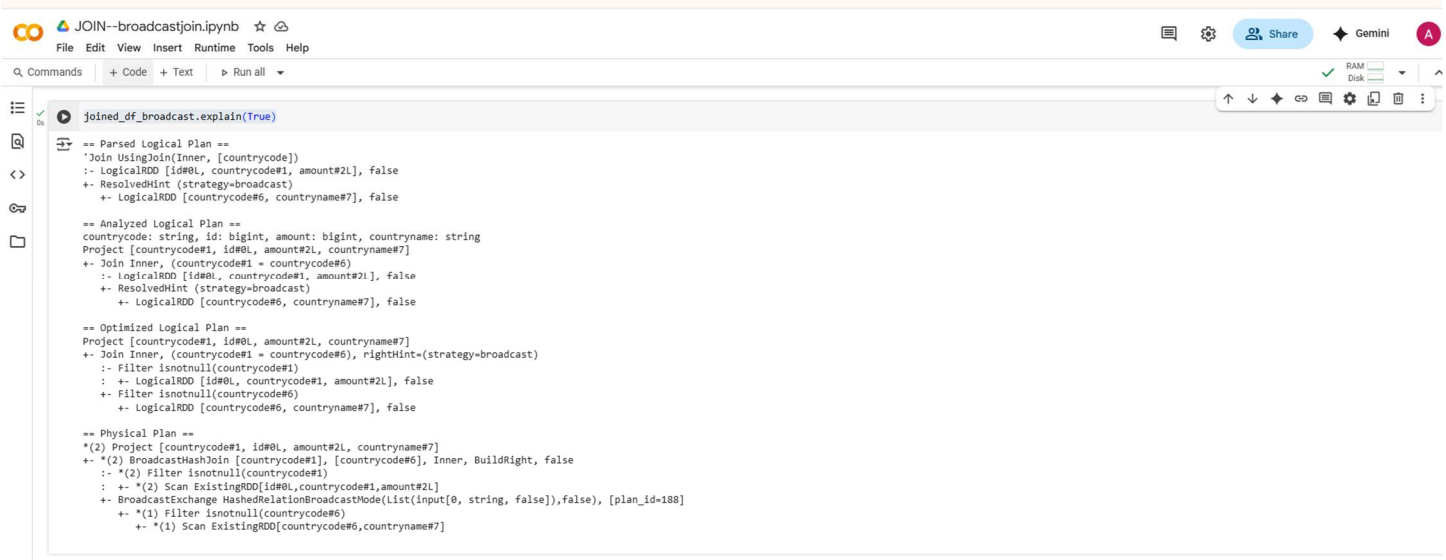
Note hash partitions is as 200



Then go for sortmerge join



Full Breakdown: Broadcast Join Execution Plan



Step-by-Step Execution (Bottom-Up)

```
== Physical Plan ==
*(2) Project [countrycode#1, id#0L, amount#2L, countryname#7]                    <-- Step 7
+- *(2) BroadcastHashJoin [countrycode#1], [countrycode#6], Inner, BuildRight, false    <-- Step 6
   :- *(2) Filter isnotnull(countrycode#1)                                       <-- Step 5
   :  +- *(2) Scan ExistingRDD[id#0L,countrycode#1,amount#2L]                    <-- Step 4
   +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, false]),false) <-- Step 3
      +- *(1) Filter isnotnull(countrycode#6)                                    <-- Step 2
         +- *(1) Scan ExistingRDD[countrycode#6,countryname#7]                   <-- Step 1
```

Step 1: Scan Right Table (Small Table countries_df)

What happens: Spark reads the countries_df, which contains a few rows and is small enough to be broadcast.

Why: Since it's a small lookup table, it's efficient to load it into memory on each executor.

*(1) Scan ExistingRDD[countrycode#6,countryname#7]

Step 2: Filter Nulls on Right Table

What happens: Spark applies a filter to remove any rows where countrycode is null.

Why: Join keys must be non-null for hash joins to work properly.

*(1) Filter isnotnull(countrycode#6)

Step 3: BroadcastExchange (Right Table)

What happens: Spark broadcasts the small table (countries_df) across all executors.

Why: This avoids shuffling the large table and enables each executor to perform a local hash join.

BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, false]),false)

Details:

HashedRelationBroadcastMode builds an in-memory hash map of countrycode → countryname.

This map is distributed to all worker nodes.

Step 4: Scan Left Table (Large Table transactions_df)

What happens: Spark scans the main dataset (transactions_df) which has many rows.

Why: This is the base dataset we want to enrich with country names.

*(2) Scan ExistingRDD[id#0L,countrycode#1,amount#2L]

Step 5: Filter Nulls on Left Table

What happens: Spark filters out rows where countrycode is null on the left side.

Why: Hash join requires valid keys to probe the hash table built from the broadcasted right table.

*(2) Filter isnotnull(countrycode#1)

Step 6: Broadcast Hash Join

What happens: Spark performs an **in-memory hash join**.

How: For each row in the large table, it:

Takes countrycode#1

Looks it up in the broadcasted hash table (from right side)

Why: No shuffle needed; this join is fast and efficient.

*(2) BroadcastHashJoin [countrycode#1], [countrycode#6], Inner, BuildRight

Notes:

BuildRight = the right table is the one broadcast and hashed.

Only countrycode#1 is probed against the pre-built hash map.

Step 7: Project Final Columns

What happens: Spark selects and arranges final output columns: countrycode, id, amount, and countryname.

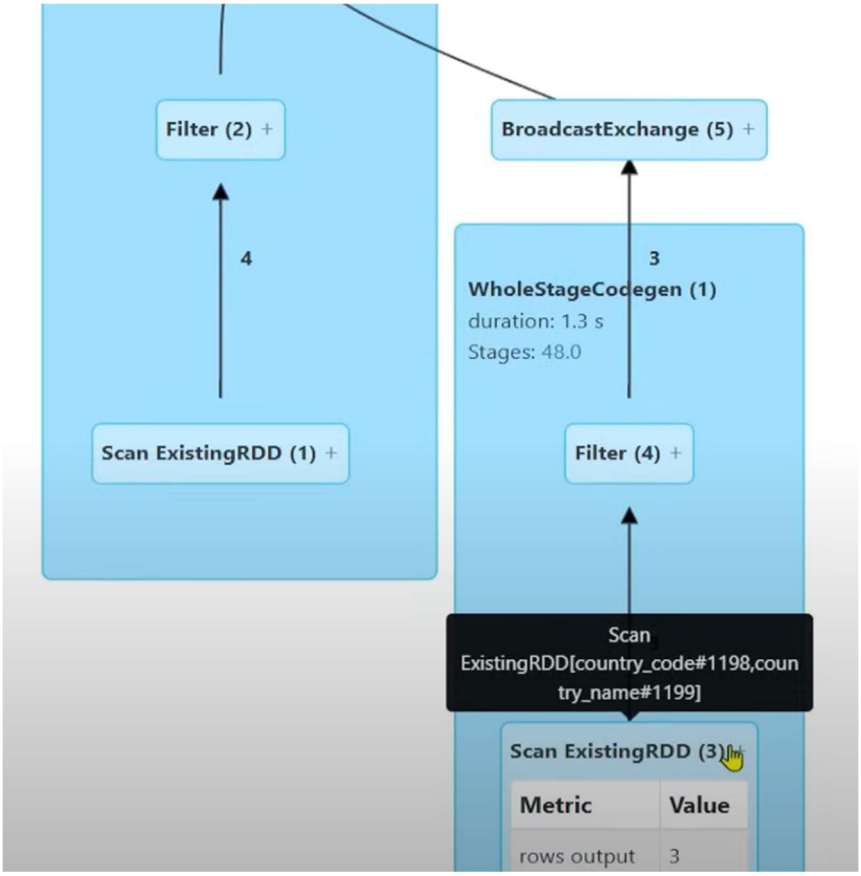
Why: Only required columns are returned in the final result.

*(2) Project [countrycode#1, id#0L, amount#2L, countryname#7]

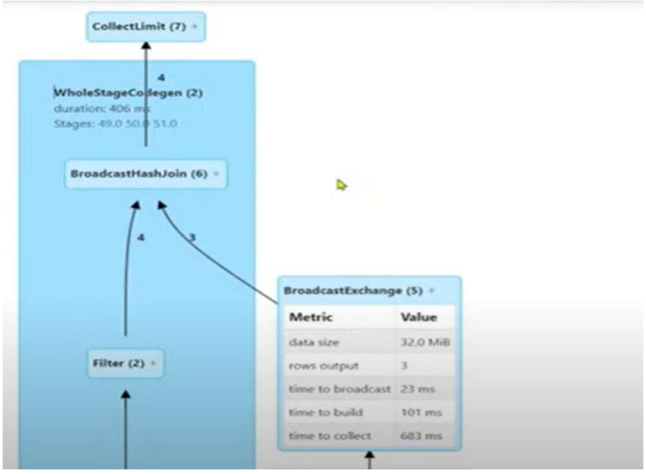
Visual Summary

[countries_df] --Scan--> --Filter--> --Broadcast-->
\
-> Hash Join <- --Filter<- --Scan-- [transactions_df]
|
Final Projected Output

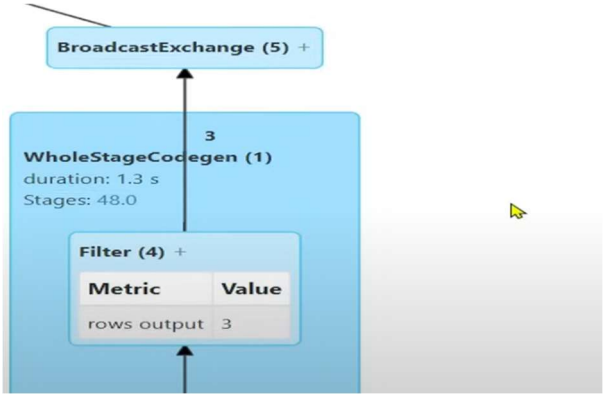
Here broadcast join it read 3 records :



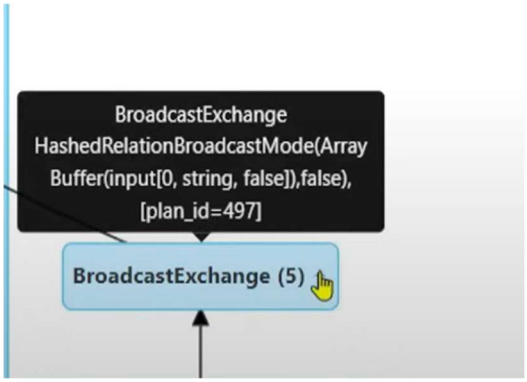
No shuffling was done



Filtered values for notnull

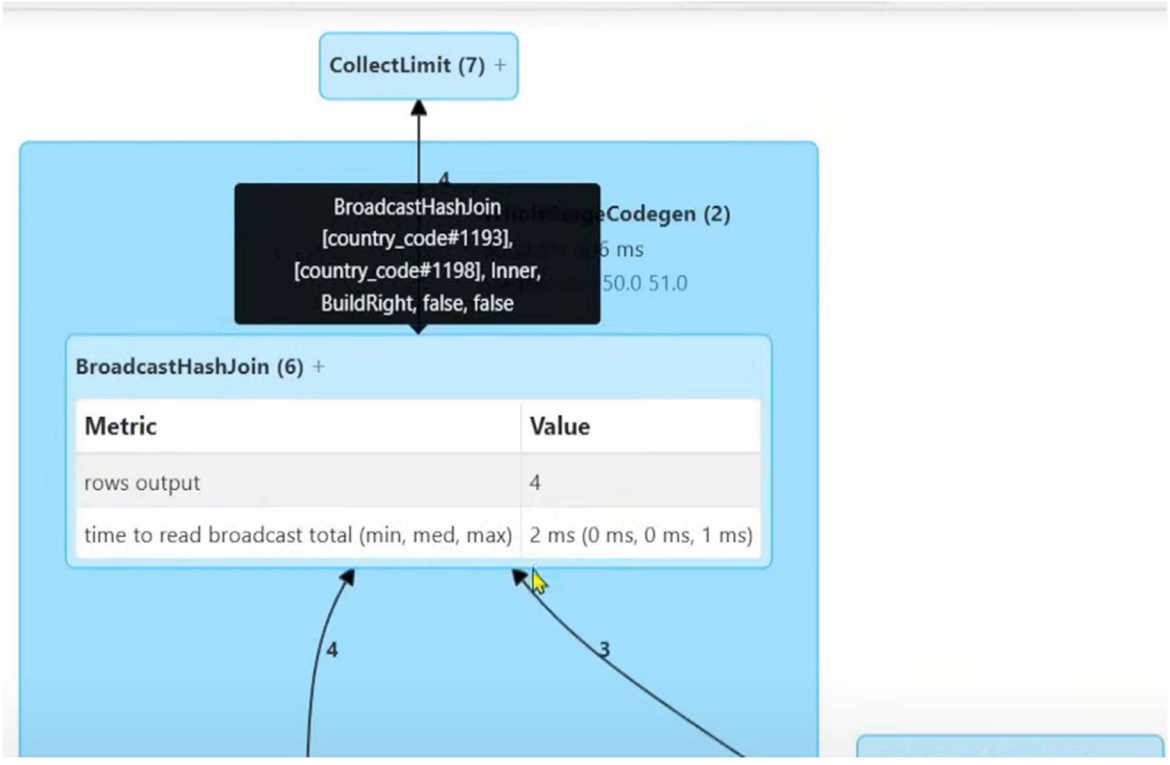


iT performed broadcast exchange



there was no shuffling on either side.

That table is sent to broadcast join



Key Differences

Feature	Broadcast Hash Join	Sort Merge Join
Shuffling	❌ No	✅ Yes (both sides)
Sorting	❌ No	✅ Required
Memory Usage	✅ Low (if right table is small)	❌ High (especially for large data)
Suitable When	One table is **small**	Both tables are **large**
Parallelism	Very good	Good but costly
Performance	Very fast for small lookup joins	Slower due to shuffle + sort
Broadcast Limit	Must fit in driver's broadcast threshold	No such limit