🔥 Most Asked PySpark Interview Question: Sort-Merge vs Shuffle Hash Join Explained! 🔥

📝 **Introduction:**

In my earlier days, I often struggled with choosing **which join to use** in PySpark — *Sort-Merge Join* vs *Shuffle Hash Join*. But once I understood their **internals**, everything became clearer, and I was able to optimize code based on the **size and characteristics** of my data.

If you've ever been stuck choosing between these joins in PySpark, this post is for you! 💡

Here's a complete breakdown of the differences between these two joins, including how they work internally and when to use each one.

🌟 Step 1: Common First Step: Data Shuffling 🌟

Both join types start with a common step: shuffling data across the cluster.

- ➡️**Purpose**:
  1. Ensure that all rows with the same join key end up on the same executor (node).
  2. Shuffling is the process where Spark redistributes data across different nodes, ensuring that rows with the same join key from both datasets end up on the same node

- ➡️ **Process**:
  1. Each row's join key is hashed.
  2. Based on this hash, the row is sent to a specific executor.

- ➡️ **Result**: All data with the same join key is co-located, enabling efficient joining.

Why is this important? Without shuffling, matching rows from different nodes can't be compared and joined. The network overhead in this step is significant for large datasets.

🔥 Sort-Merge Join: Sorting and Merging🔥

Once the data is shuffled, the Sort-Merge Join starts by sorting both datasets on the join key. After sorting, Spark performs a merge operation to match rows with the same key.

➡️**Step 1:** Sorting: Spark sorts both datasets locally in each partition.

➡️**Step 2:** After sorting, Spark can then efficiently merge the two datasets. It starts by looking at the first row from both datasets and compares their join keys. If the keys match, the rows are joined. If one key is smaller, Spark moves to the next row in that dataset until the keys match.

Think of it like merging two sorted arrays:

- If the value in one array is smaller, you move the pointer in that array until you find a match.

💡Example💡

Let's say we're joining **'Customers'** and **'Orders'** on **'customer_id'**:

1. ✨ Shuffle ✨ : Both datasets are distributed so that all data for each 'customer_id' is on the same executor.
2. ✨ Sort ✨: Each executor sorts its portion of 'Customers' and 'Orders' by 'customer_id'.
3. ✨ Merge ✨: Sorted data is scanned, matching 'Customers' and 'Orders' with the same 'customer_id'.

🚀When to Use🚀

- Both datasets are large.
- Join key has high cardinality (many unique values).
- Available memory is limited.

➡️🔥 Shuffle Hash Join: Building and Probing the Hash Table🔥

Step-by-Step Process

1. ➡️**Shuffle**: Data is distributed based on join keys (same as Sort-Merge Join).

2. ➡️**Build Phase**:
    - Spark creates a hash table using the join key as the **hash key** and the entire row of the smaller dataset as the **value**.
    - **Hash Key**: The join key.
    - **Value**: The rest of the columns in the smaller dataset. For example, if you're joining on **`CustomerID`**, the hash key is **`CustomerID`**, and the value would be the entire row containing customer details (`Name`, `Country`, etc.).

3. ➡️ **Probe Phase**:

    For each partition of the **larger dataset**, Spark computes the hash of the join key and checks if a matching key exists in the hash table. If a match is found, the rows are joined.

    - The larger dataset is processed in partitions.
    - Each row's join key is used to probe the hash table for matches.

Detailed Look at the Probe Phase

- **Partition Loading**: A partition of the larger dataset is loaded into memory.
    - Partition size typically ranges from 32MB to 128MB.
- **Row-by-Row Processing**: Within each partition, rows are processed sequentially.
    - For each row:
        1. Extract the join key.
        2. Use this key to look up matches in the hash table.
        3. If a match is found, produce a joined row.
- **Partition Cycling**: After processing one partition, move to the next until all are processed.

💡Example💡

Joining 'Customers' (smaller, 100,000 records) with 'Orders' (larger, 10 million records):

1. ✨ Shuffle ✨: Distribute both datasets across executors based on 'customer_id'.
2. ✨ Build ✨: Create hash tables for 'Customers' on each executor.
3. ✨ Probe ✨:
    - **'Orders'** are divided into partitions (e.g., 100 partitions of 100,000 records each).
    - For each partition:
        - Load into memory.
        - Process each order, looking up the customer in the hash table.
        - Output joined results.
        - Move to the next partition.

5. Key Differences and When to Use Each Join

Aspect	Sort-Merge Join (SMJ)	Shuffle Hash Join (SHJ)
Shuffling	Data is shuffled based on the join key.	Data is shuffled based on the join key.
Sorting	Data is sorted after shuffling.	No sorting is required.
Hash Table	No hash table is built.	A hash table is built for the smaller dataset.
Memory Usage	No significant memory requirement for sorting.	Requires the smaller dataset to fit into memory.
Handling Large Datasets	Best for large datasets on both sides.	Best when one dataset is significantly smaller.
Handling Skewed Data	Can handle skewed join keys relatively well.	May face memory issues if the hash table becomes too large.
Performance	Can be slower due to sorting overhead.	Fast but limited by memory availability.

🧠When to Use Sort-Merge Join (SMJ)🧠:

- **Large datasets on both sides**: If both datasets are large and comparable in size, SMJ is more efficient.
- **Skewed data**: SMJ can handle data skew (i.e., when some join keys appear much more frequently than others) more gracefully because it doesn't require loading large parts of the data into memory.
- **Resource constraints**: When memory is limited, SMJ might be preferable because it doesn't require building large in-memory structures like hash tables.

🧠When to Use Shuffle Hash Join (SHJ)🧠:

- **One small dataset and one large dataset**: SHJ is more efficient when one of the datasets is much smaller and can easily fit into memory.
- **Faster join operation**: If sorting is too expensive or unnecessary, SHJ can be faster by avoiding the sorting step altogether.

---

6. Conclusion: Understanding Joins in PySpark

Both **Sort-Merge Join** and **Shuffle Hash Join** are powerful techniques for joining datasets in PySpark, but they have different use cases and trade-offs.

- **Sort-Merge Join** is better for large datasets on both sides and is more robust when dealing with skewed data or when memory is a constraint.
- **Shuffle Hash Join** shines when one dataset is much smaller and can be held in memory, allowing for faster joins without sorting.