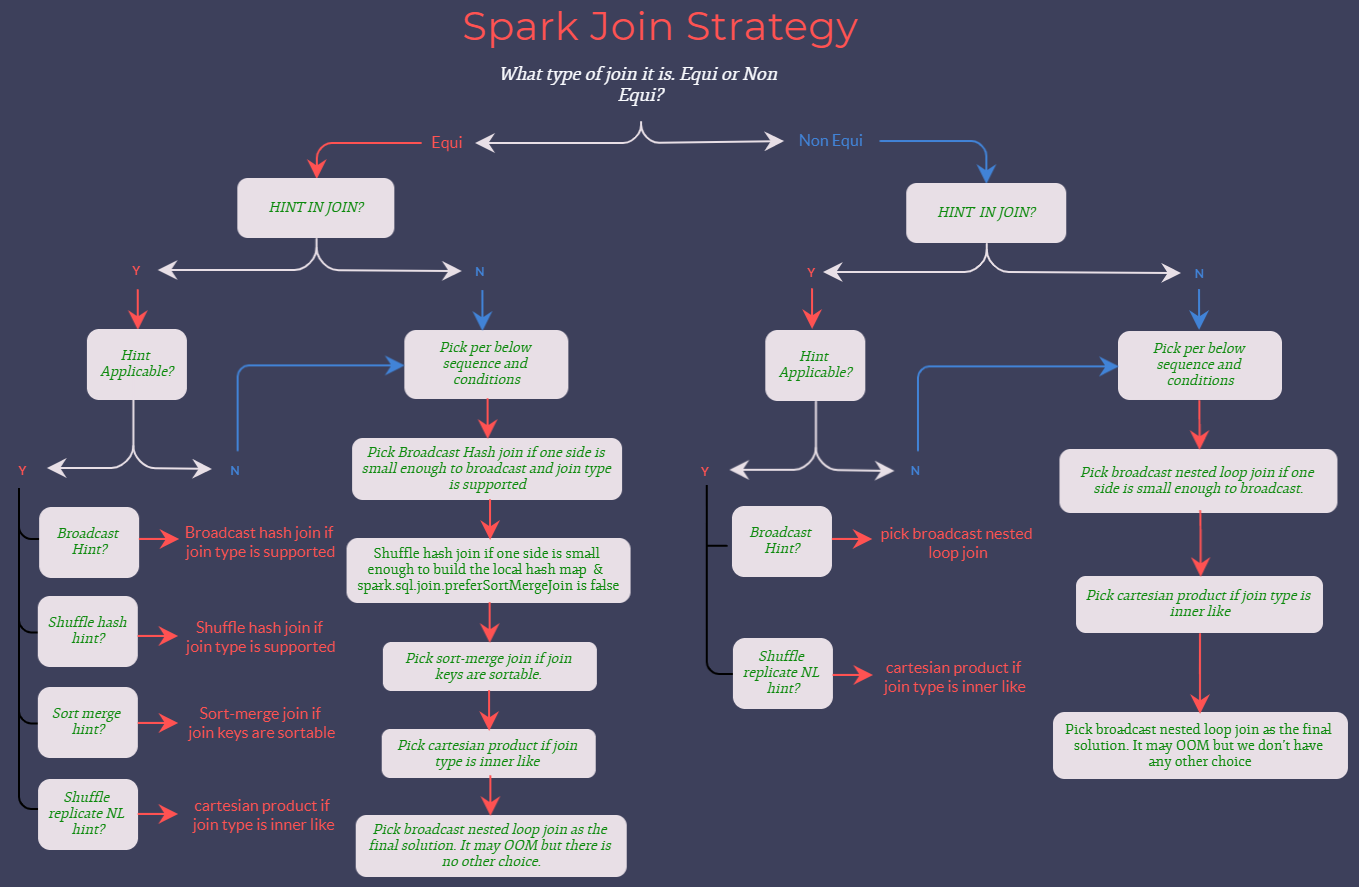
***How does Spark Select Join Strategy?***

*Spark selects Join strategy by considering the below:*

1. Type of Join
2. Hint in Join



Spark Join Strategy Flowchart

As shown in the above Flowchart, Spark selects the Join strategy based on Join type and Hints in Join. Spark 2.x supports Broadcast Hint alone whereas Spark 3.x supports all Join hints mentioned in the Flowchart.

When the hints are specified on both sides of the Join, Spark selects the hint in the below order:  
1. BROADCAST hint  
2. MERGE hint  
3. SHUFFLE\_HASH hint  
4. SHUFFLE\_REPLICATE\_NL hint  
5. When BROADCAST hint or SHUFFLE\_HASH hint are specified on both sides, Spark will pick up the build side based on the join type and the data size

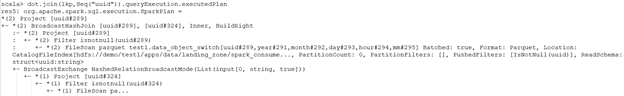
The specified hint will not always be selected since a specific strategy may not support all the Join types.

***Let’s understand Spark Join Strategies in detail.***

***Join Strategy Types***

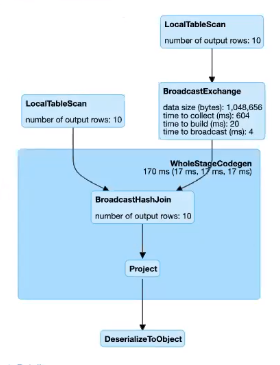
**1. Broadcast Hash Join**

When one of the data frames is small and fits in the memory, it will be broadcasted to all the executors, and a Hash Join will be performed.



Broadcast Hash Join- Without Hint

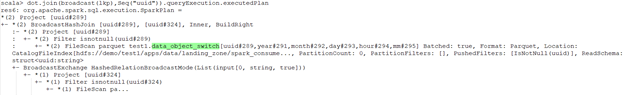
The property spark.sql.autoBroadcastJoinThreshold can be configured to set the Maximum size in bytes for a dataframe to be broadcasted.  
Here, spark.sql.autoBroadcastJoinThreshold=-1 will disable the broadcast Join whereas default spark.sql.autoBroadcastJoinThreshold=10485760, i.e 10MB.



Broadcast Hash Join

Which table will be broadcasted in the below conditions?

1. The Join side with the hint will be broadcasted irrespective of autoBroadcastJoinThreshold, if a broadcast hint is specified on either side of the join.
2. The side with a smaller physical data size will be broadcasted, if broadcast hints are specified on both sides of the Join.
3. The table will be broadcasted to all the executor nodes if there is no hint and the physical size of the table < autoBroadcastJoinThreshold.



Broadcast Hash Join

If the broadcast side is small, BHJ can perform faster than other Join algorithms as there is no shuffling involved.

***Is broadcasting always good for performance?***Not at all!

The broadcasting table is a network-intensive operation. When the broadcasted table is big, it may lead to OOM or performs worse than other algorithms.

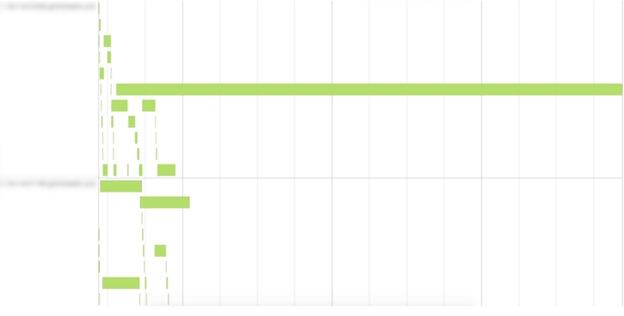
In the above snippets, if you give more resources to the cluster, the non-broadcasted version will run faster than the broadcasted one as the broadcasting operation is expensive in itself. If we are increasing the number of executors, those executors need to receive the table. By increasing the number of executors, we are increasing the broadcasting cost too.

Now, imagine you are broadcasting a medium-sized table. When you run the code, everything is fine and super-fast. But in the future when a medium-sized table is no more “medium”, then your code will break with OOM.

**Skewness**

When you want to join the two tables, ‘Skewness’ is the most common issue developers face. When the Join key is not uniformly distributed in the dataset, the Join will be skewed. Spark cannot perform operations in parallel when the Join is skewed, as the Join’s load will be distributed unevenly across the Executors.

If one table is very small, we can decide to broadcast it straightaway! Observe what happened to the tasks during the execution: one of the tasks took much more time.



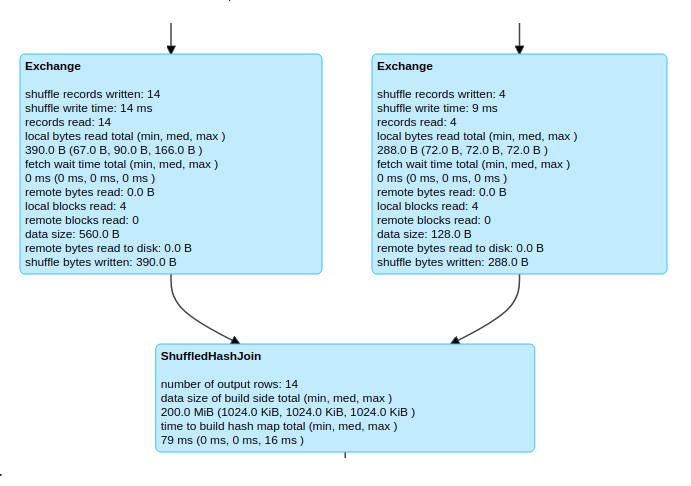
Join Skewness

**2. Shuffle Hash Joins**

When the table is relatively large, the use of broadcast may cause driver- and executor-side memory issues. In this case, the Shuffle Hash Join will be used. It is an expensive join as it involves both shuffling and hashing. Also, it requires memory and computation for maintaining a hash table.

Shuffle Hash Join is performed in two steps:

1. Step 1- Shuffling: The data from the Join tables are partitioned based on the Join key. It does shuffle the data across partitions to have the same Join keys of the record assigned to the corresponding partitions.
2. Step 2- Hash Join: A classic single node Hash Join algorithm is performed for the data on each partition.



Shuffle Hash Join

If you want to use the Shuffle Hash Join, spark.sql.join.preferSortMergeJoin needs to be set to false, and the cost to build a hash map is less than sorting the data. The Sort-merge Join is the default Join and is preferred over Shuffle Hash Join.

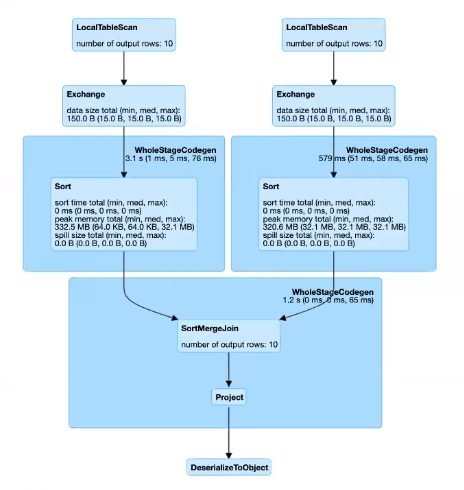
Shuffle Hash Join’s performance is the best when the data is distributed evenly with the key you are joining and you have an adequate number of keys for parallelism.

**3. Shuffle sort-merge Join**

Shuffle Sort-merge Join (SMJ) involves shuffling of data to get the same Join key with the same worker, and then performing Sort-merge Join operation at the partition level in the worker nodes. Partitions are sorted on the Join key before the Join operation.

It has 3 phases:

1. Shuffle Phase: Both large tables will be repartitioned as per the Join keys across the partitions in the cluster.
2. Sort Phase: Sort the data within each partition parallelly.
3. Merge Phase: Join the sorted and partitioned data. It is merging of the dataset by iterating over the elements and joining the rows having the same value for the Join keys.



Sort-Merge Join

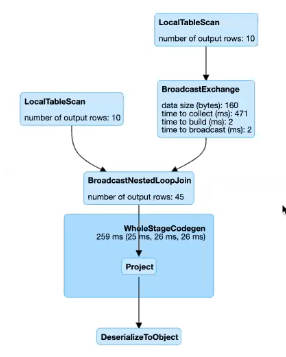
SMJ performs better than other joins most of the time and has a very scalable approach as it does away with the overhead of hashing and does not require the entire data to fit inside the memory.

**4. Broadcast Nested Loop Join**

Broadcast Nested Loop Join opts when it does not cross the threshold for broadcasting. It supports both Equi-Joins and Non-Equi-Joins. It also supports all the other Join types, but the implementation is optimized when :

1. The Left side is broadcasted in the right outer Join.
2. The Right side is broadcasted in a left outer, left semi, and left anti Join.
3. In an inner-like Join.

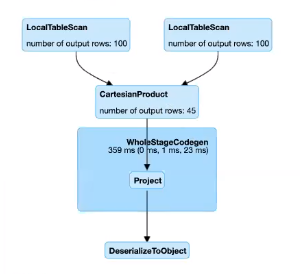
In other cases, we need to scan the data multiple times, which can be rather slow.



Broadcast Nested Loop Join

**5. Cartesian Join**

When the Join type is inner like and there are no Join keys present, the Cartesian Join will be selected. Cross Join computes a cartesian product of 2 tables. If we want to use Cartesian Join, we have to either set the spark.sql.crossJoin.enabled=true in our Spark session builder object or set it for Spark-shell : spark-shell — conf spark.sql.crossJoin.enabled=true, otherwise Spark will throw an AnalysisException.



Cartesian Join

**Check the below table for the Join strategies Supported by the Join types**



Join type to Join Strategy mapping

***Conclusion***

Even if Joins in Apache Spark internally choose the best Join algorithm, developers can change this decision using hints. Providing hints in the Join without understanding the nature of the data may lead to OOM errors. If the developer is familiar with the underlying data and is not providing hint in the Join, he/she may lose an occasion to optimize the Join operation.