Partition and Join Strategy

# Partitioning Hints

* **COALESCE**The COALESCE hint can be used to reduce the number of partitions to the specified number of partitions. It takes a partition number as a parameter.
* **REPARTITION**The REPARTITION hint can be used to repartition to the specified number of partitions using the specified partitioning expressions. It takes a partition number, column names, or both as parameters.
* **REPARTITION\_BY\_RANGE**The REPARTITION\_BY\_RANGE hint can be used to repartition to the specified number of partitions using the specified partitioning expressions. It takes column names and an optional partition number as parameters.

| SELECT /\*+ COALESCE(3) \*/ \* FROM t;  SELECT /\*+ REPARTITION(3) \*/ \* FROM t;  SELECT /\*+ REPARTITION(c) \*/ \* FROM t;  SELECT /\*+ REPARTITION(3, c) \*/ \* FROM t;  SELECT /\*+ REPARTITION\_BY\_RANGE(c) \*/ \* FROM t;  SELECT /\*+ REPARTITION\_BY\_RANGE(3, c) \*/ \* FROM t; |
| --- |

## 

# Join Hints

Join hints allow users to suggest the join strategy that Spark should use. Prior to Spark 3.0, only the BROADCAST Join Hint was supported. MERGE, SHUFFLE\_HASH and SHUFFLE\_REPLICATE\_NL Joint Hints support was added in 3.0. When different join strategy hints are specified on both sides of a join, Spark prioritizes hints in the following order: BROADCAST over MERGE over SHUFFLE\_HASH over SHUFFLE\_REPLICATE\_NL. When both sides are specified with the BROADCAST hint or the SHUFFLE\_HASH hint, Spark will pick the build side based on the join type and the sizes of the relations. Since a given strategy may not support all join types, Spark is not guaranteed to use the join strategy suggested by the hint.

Join Hints Types

* **BROADCAST**Suggests that Spark use broadcast join. The join side with the hint will be broadcast regardless of autoBroadcastJoinThreshold. If both sides of the join have the broadcast hints, the one with the smaller size (based on stats) will be broadcast. The aliases for BROADCAST are BROADCASTJOIN and MAPJOIN. The small table is not written to disk, it is rather kept in executor memory.
* **MERGE**Suggests that Spark use shuffle sort merge join. The aliases for MERGE are SHUFFLE\_MERGE and MERGEJOIN. This is the reducer side join strategy and the most common join strategy (default) and it is scalable (e.g. join two large tables), but it is slower compared to broadcast join and hash join. One drawback is that it creates high disk I/O due to spill. To force sort merge hash join, you might need to disable auto broadcast join. If there is a skew in the partition, this join is affected by the large partition.
* **SHUFFLE\_HASH**Suggests that Spark use shuffle hash join. If both sides have the shuffle hash hints, Spark chooses the smaller side (based on stats) as the build side. To opt for this join spark.sql.join.preferSortMergeJoin=false. Does not work for non-equi joins. Data skew affects the join performance.
* **SHUFFLE\_REPLICATE\_NL**Suggests that Spark use shuffle-and-replicate nested loop join.

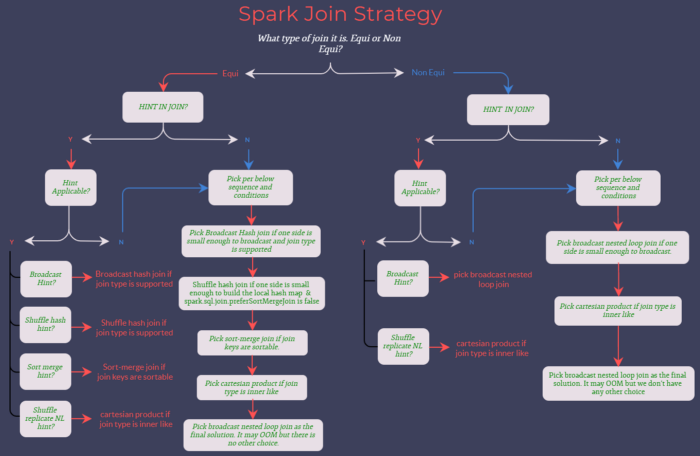
| -- Join Hints **for** broadcast join SELECT /\*+ BROADCAST(t1) \*/ \* FROM t1 INNER JOIN t2 ON t1.key = t2.key; SELECT /\*+ BROADCASTJOIN (t1) \*/ \* FROM t1 left JOIN t2 ON t1.key = t2.key; SELECT /\*+ MAPJOIN(t2) \*/ \* FROM t1 right JOIN t2 ON t1.key = t2.key;  -- Join Hints for shuffle sort merge join SELECT /\*+ SHUFFLE\_MERGE(t1) \*/ \* FROM t1 INNER JOIN t2 ON t1.key = t2.key; SELECT /\*+ MERGEJOIN(t2) \*/ \* FROM t1 INNER JOIN t2 ON t1.key = t2.key; SELECT /\*+ MERGE(t1) \*/ \* FROM t1 INNER JOIN t2 ON t1.key = t2.key;  -- Join Hints for shuffle hash join, set spark.sql.join.preferSortMergeJoin=false SELECT /\*+ SHUFFLE\_HASH(t1) \*/ \* FROM t1 INNER JOIN t2 ON t1.key = t2.key;  -- Join Hints for shuffle-and-replicate nested loop join SELECT /\*+ SHUFFLE\_REPLICATE\_NL(t1) \*/ \* FROM t1 INNER JOIN t2 ON t1.key = t2.key;  -- When different join strategy hints are specified on both sides of a join, Spark -- prioritizes the BROADCAST hint over the MERGE hint over the SHUFFLE\_HASH hint -- over the SHUFFLE\_REPLICATE\_NL hint. -- Spark will issue Warning in the following example -- org.apache.spark.sql.catalyst.analysis.HintErrorLogger: Hint (strategy=merge) -- is overridden by another hint and will not take effect. SELECT /\*+ BROADCAST(t1), MERGE(t1, t2) \*/ \* FROM t1 INNER JOIN t2 ON t1.key = t2.key; |
| --- |

The Join operation is the most frequently used transformation in Apache Spark. The syntax for writing the Join operation is simple but what goes on behind is lost. Apache Spark checks a couple of Algorithms and then chooses the best out of them. If we don't know these internal algorithms and are not aware of what Spark chooses, it might make a simple Join operation expensive.

***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

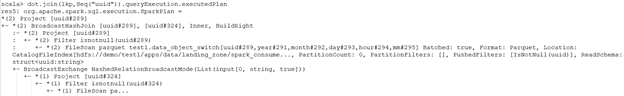
The specified hint will not always be selected since a specific strategy may not support all the Join types.

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

# **Join Strategy Types**

## **1. Broadcast Hash Join**

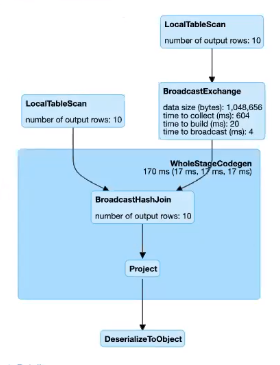
When one of the data frames is small and fits in the memory, it will be broadcasted to all the executors, and a Hash Join will be performed.



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.

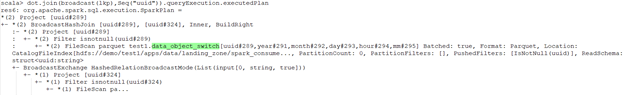
Here, spark.sql.autoBroadcastJoinThreshold=-1 will disable the broadcast Join whereas default spark.sql.autoBroadcastJoinThreshold=10485760, i.e 10MB.



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 perform worse than other algorithms.

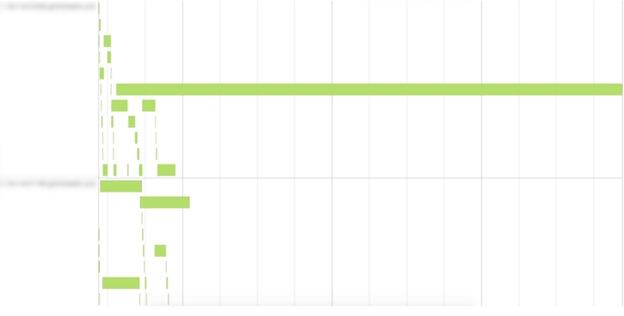
In the above snippets, if you give more resources to the cluster, the non-broadcasted version will run faster than the broadcasted one as the broadcasting operation is expensive in itself. If we are increasing the number of executors, those executors need to receive the table. By increasing the number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



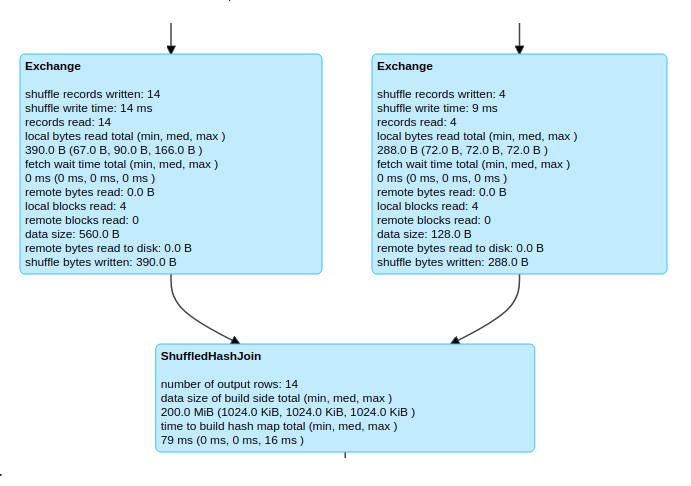
Join Skewness

## **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.

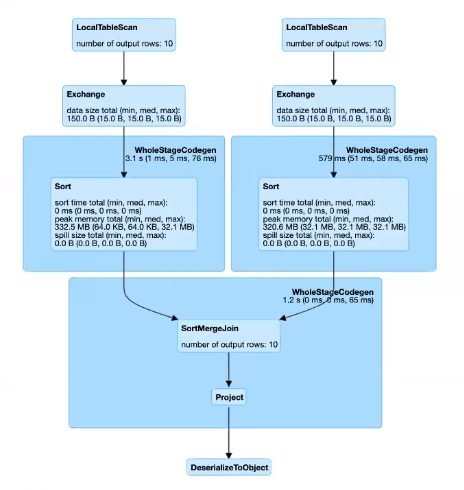
Shuffle Hash Join’s performance is the best when the data is distributed evenly with the key you are joining and you have an adequate number of keys for parallelism.

## **3. Shuffle sort-merge Join**

Shuffle Sort-merge Join (SMJ) involves shuffling 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 a merging of the dataset by iterating over the elements and joining the rows having the same value for the Join keys.



Sort-Merge Join

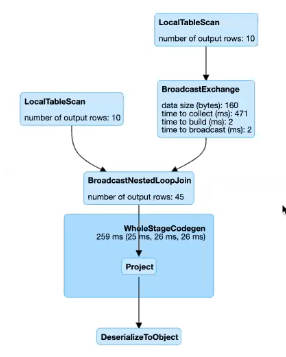
SMJ performs better than other joins most of the time and has a very scalable approach as it does away with the overhead of hashing and does not require the entire data to fit inside the memory.

## **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.

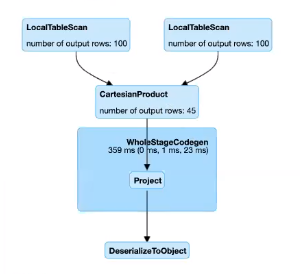
In other cases, we need to scan the data multiple times, which can be rather slow.



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.

# The Adaptive Query Execution (AQE) framework

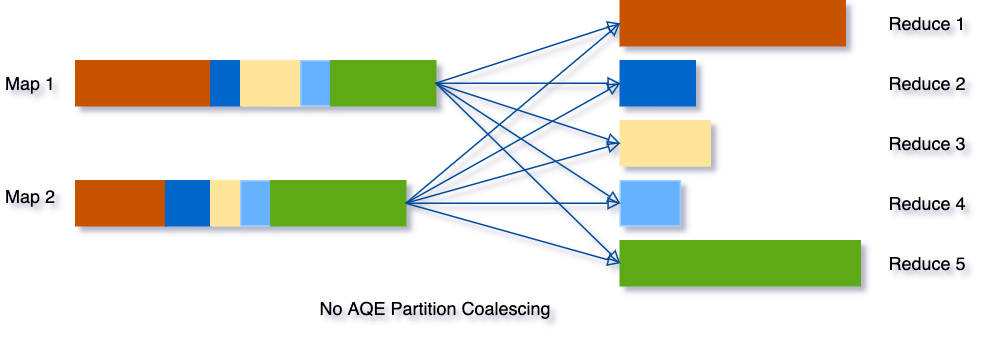
<https://spark.apache.org/docs/latest/sql-performance-tuning.html#adaptive-query-execution>

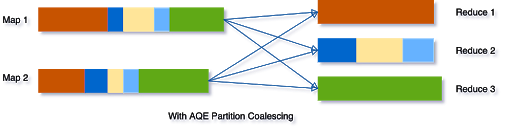
* When the query starts, the Adaptive Query Execution framework first kicks off all the leaf stages — the stages that do not depend on any other stages.
* As soon as one or more of these stages finish materialization, the framework marks them complete in the physical query plan and updates the logical query plan accordingly, with the runtime statistics retrieved from completed stages.
* Based on these new statistics, the framework then runs the optimizer (with a selected list of logical optimization rules), the physical planner, as well as the physical optimization rules, which include the regular physical rules and the adaptive-execution-specific rules, such as coalescing partitions, skew join handling, etc.
* Now that we’ve got a newly optimized query plan with some completed stages, the adaptive execution framework will search for and execute new query stages whose child stages have all been materialized, and repeat the above execute-reoptimize-execute process until the entire query is done.

In Spark 3.0, the AQE framework is shipped with three features:

* Dynamically coalescing shuffle partitions
* Dynamically switching join strategies
* Dynamically optimizing skew joins

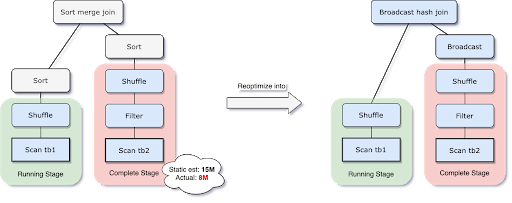
## Dynamically coalescing shuffle partitions





## Dynamically switching join strategies

* Spark supports a number of join strategies, among which broadcast hash join is usually the most performant if one side of the join can fit well in memory. And for this reason, Spark plans a broadcast hash join if the estimated size of a join relation is lower than the broadcast-size threshold.
* But a number of things can make this size estimation go wrong — such as the presence of a very selective filter — or the join relation being a series of complex operators other than just a scan.
* To solve this problem, AQE now replans the join strategy at runtime based on the most accurate join relation size. As can be seen in the following example, the right side of the join is found to be way smaller than the estimate and also small enough to be broadcast, so after the AQE reoptimization the statically planned sort merge join is now converted to a broadcast hash join.

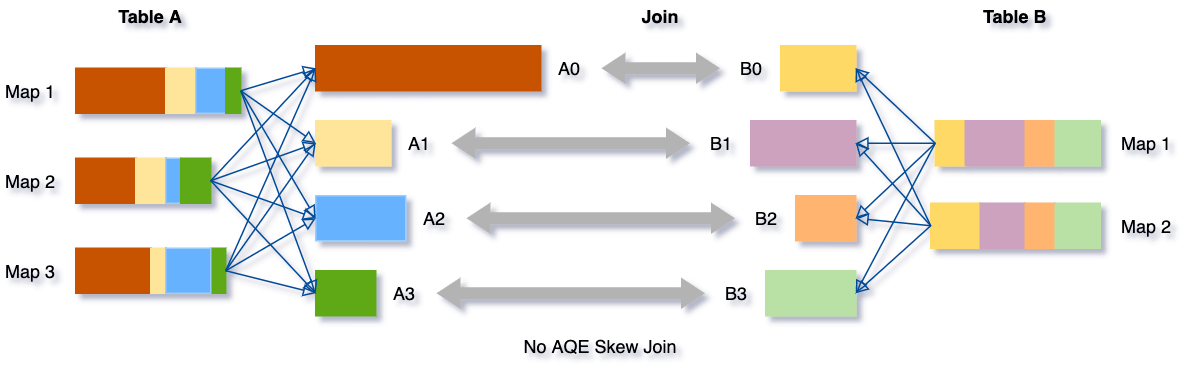


## 

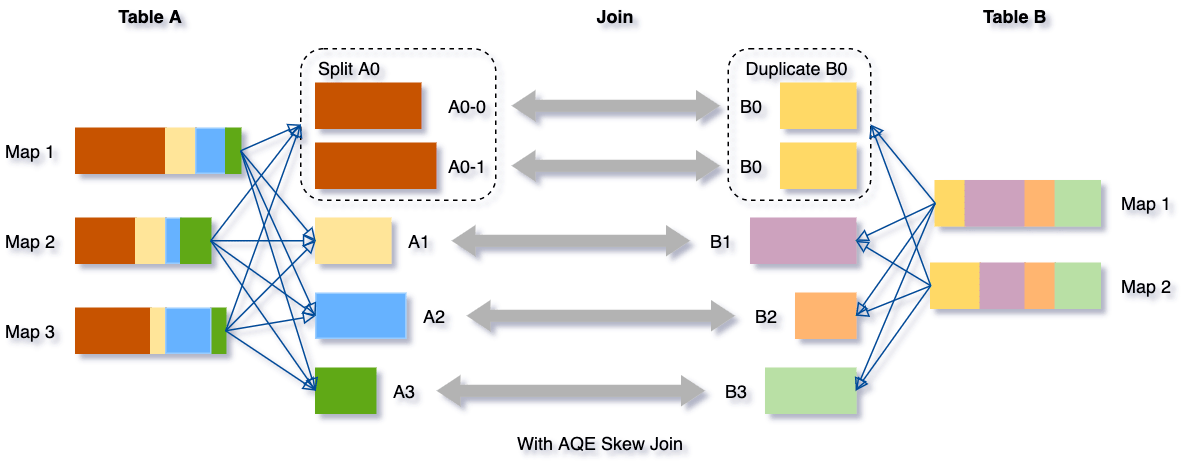
## Dynamically optimizing skew joins

* Data skew occurs when data is unevenly distributed among partitions in the cluster.
* Severe skew can significantly downgrade query performance, especially with joins.
* AQE skew join optimization detects such skew automatically from shuffle file statistics.
* It then splits the skewed partitions into smaller subpartitions, which will be joined to the corresponding partition from the other side respectively.

Let’s take this example of table A join table B, in which table A has a partition A0 significantly bigger than its other partitions.



The skew join optimization will thus split partition A0 into two subpartitions and join each of them to the corresponding partition B0 of table B.



Without this optimization, there would be four tasks running the sort merge join with one task taking a much longer time. After this optimization, there will be five tasks running the join, but each task will take roughly the same amount of time, resulting in an overall better performance.

**References**

* <https://blog.clairvoyantsoft.com/apache-spark-join-strategies-e4ebc7624b06>
* <https://nivedita-mondal.medium.com/spark-join-strategies-fb984b50441d>
* <https://spark.apache.org/docs/3.0.0/sql-ref-syntax-qry-select-hints.html#description>
* <https://towardsdatascience.com/best-practices-for-bucketing-in-spark-sql-ea9f23f7dd53>
* <https://www.databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html>