Demystifying Joins in Apache Spark

This story is exclusively dedicated to the Join operation in Apache Spark, giving you an overall perspective of the foundation on which Spark Join technology is built upon.

Join operations are often used in a typical data analytics flow in order to correlate two data sets. Apache Spark, being a unified analytics engine, has also provided a solid foundation to execute a wide variety of Join scenarios.

At a very high level, Join operates on two input data sets and the operation works by matching each of the data records belonging to one of the input data sets with every other data record belonging to another input data set. On finding a match or a non-match (as per a given condition), the Join operation could either output an individual record, being matched, from either of the two data sets or a Joined record. The joined record basically represents the combination of individual records, being matched, from both the data sets.

Important Aspects of Join Operation:

Let us now understand the three important aspects that affect the execution of Join operation in Apache Spark. These are:

1) Size of the Input Data sets: The size of the input data sets directly affects the execution efficiency and reliability of the Join operation. Also, the comparative sizing of the input data sets affects the selection of the Join mechanism which could further affect the efficiency and reliability of the Join mechanism.

2) The Join Condition: Condition or the clause on the basis of which the input data sets are being joined is termed as Join Condition. The condition typically involves logical comparison(s) between attributes belonging to the input data sets. Based on the Join condition, Joins are classified into two broad categories, Equi Join and Non-Equi Joins.

Equi Joins involves either one equality condition or multiple equality conditions that need to be satisfied simultaneously. Each equality condition being applied between the attributes from the two input data sets. For example, (A.x == B.x) or ((A.x == B.x) and (A.y == B.y)) are the two examples of Equi Join conditions on the x, y attributes of the two input data sets, A and B, participating in a Join operation.

Non-Equi Joins do not involve equality conditions. However, they may allow for multiple equality conditions that must not be satisfied simultaneously. For example, (A.x < B.x) or ((A.x == B.x) or (A.y == B.y)) are the two examples of Non-Equi Join conditions on the x, y attributes of the two input data sets, A and B, participating in a Join operation.

3) The Join type: The Join type affects the outcome of the Join operation after the Join condition is applied between the records of the input data sets. Here is the broad classification of the various Join types:

Inner Join: Inner Join outputs only the matched Joined records (on the Join condition) from the input data sets.

Outer Join: Outer Join outputs, in addition to matched Joined records, also outputs the non-matched records. Outer Join is further classified into the left, right, and full outer Joins based on the choice of the input data set(s) for outputting the non-matched records.

Semi Join: Semi Join outputs the individual record belonging to only one of the two input datasets, either on a matched or non-matched instance. If the record, belonging to one of the input datasets, is outputted on a non-matched instance, Semi Join is also called as Anti Join.

Cross Join: Cross Join outputs all Joined records that are possible by combining each record from one input data set with every record of the other input data set.

Based on the above three important aspects of the Join execution, Apache Spark chooses the right mechanism to execute the Join.

Various Mechanisms of Join execution

After understanding the various aspects of executing a Join operation, Let us now understand the various mechanisms to execute the Join operation.

Apache Spark provides five mechanisms in total to execute Join operations. These are:

- Shuffle Hash Join
- Broadcast Hash Join
- Sort Merge Join
- Cartesian Join
- Broadcast Nested Loop Join

Broadcast Hash Join: In the 'Broadcast Hash Join' mechanism, one of the two input Datasets (participating in the Join) is broadcasted to all the executors. A Hash Table is being built on all the executors from the broadcasted Dataset, after which, each partition of the non-broadcasted input Dataset is joined independently to the other Dataset being available as a local hash table.

> *'Broadcast Hash Join' does not involve any shuffling stage and is most efficient. The only requirement for the reliability of the same is that executors should have enough memory to house the broadcasted data set. Therefore, Spark avoids this mechanism when both the input datasets are fairly large then a configurable threshold.*

Shuffle Hash Join: In the 'Shuffle Hash Join' mechanism, firstly, two input data sets are aligned to a chosen output partitioning scheme (To know more about the chosen output partitioning scheme, you can refer to my recent book titled, "Guide to Spark Partitioning"). In case, if one or both the input data sets don't conform to the chosen partitioning scheme, a shuffle operation is executed before the actual Join to achieve the conformance.

After the conformance to the selected output partitioning is ensured for both the input Datasets, Shuffle Hash executes the Join operation, per output partition, using the standard Hash Join approach. Meaning, per output partition, a hash table is first constructed from the corresponding partition of the smaller input Dataset, and then the corresponding partition of the larger input data set is joined with the constructed hash table.

> *Memory requirement on executors is relatively less in the case of 'Shuffle Hash Join' as compared to 'Broadcast Hash Join'. This is due to the fact that the hash table is being built only on a certain partition of smaller input data set. So, if you provision a large number of output partitions and you have a large number of executors with decent memory configuration, you can achieve higher efficiency for your Join operation with 'Shuffle Hash Join'. However, the efficiency would be less than the 'Broadcast Hash Join' if Spark needs to execute an additional shuffle operation on one or both input data sets for conformance to output partitioning.*

Sort Merge Join: The initial part of 'Sort Merge Join' is similar to 'Shuffle Hash Join'. Here also, firstly, two input data sets are aligned to a chosen output partitioning scheme. In case, if one or both the input data sets don't conform to the chosen partitioning scheme, a shuffle operation is executed before the actual Join to achieve the conformance.

After the conformance to the selected output partitioning is ensured for both the input Datasets, Sort Merge executes the Join operation, per output partition, using the standard Sort Merge Join approach.

> *'Sort Merge Join' is computationally less efficient when compared to 'Shuffle Hash Join' and 'Broadcast Hash Join', however, the memory requirements on executors for executing 'Sort Merge Join' are significantly lower than the 'Shuffle Hash' and 'Broadcast Hash'. Also, similar to 'Shuffle Hash Join', if input data sets don't conform to desired output partitioning, then shuffle operation of one or both the input data sets, as the case may be, adds to the overhead of the 'Sort Merge Join' execution.*

Cartesian Join: Cartesian Join is used exclusively to perform cross join between two input data sets. The number of output partitions is always equal to the product of the number of partitions of the input data set. Each output partition is mapped to a unique pair of partitions, each pair comprising of one partition from one input dataset and other partition from the other input dataset. For each of the output partition of the output data set, the data is computed by doing a cartesian product on data from two input partitions mapped to the output partition.

> *Drawback of Cartesian Join explodes the number of output partitions. But if you require cross Join, Cartesian is the only mechanism.*

Broadcast Nested Loop Join: In 'Broadcast Nested Loop Join', one of the input data set is broadcasted to all the executors. After this, each partition of the non-broadcasted input data set is joined to the broadcasted data set using the standard Nested Loop Join procedure to produce the output joined data.

> 'Broadcast Nested Loop Join' *is computationally least efficient since a nested loop is executed to compare the two data sets. Also, it is memory intensive since one of the input data set needs to be broadcasted to all the executors.*

How Spark chooses a Join Mechanism?

After looking at the important aspects of Join operation and various mechanisms of Join execution, let us now see how Spark chooses a particular mechanism:

Spark chooses a particular mechanism for executing a Join operation based on the following factors:

- Configuration parameters
- Join hints
- Size of input data sets
- Join Type
- Equi or Non-Equi Join

Spark has provided flexibility in Join APIs to specify optional Join hints to finalize a Join mechanism. Join hints, such as 'broadcast', 'merge', 'shuffle_hash' and 'shuffle_replicate_nl' can be provided with the datasets participating in Joins.

Here is a comprehensive description of how Spark chooses various Join mechanisms with respect to the above factors:

'Broadcast Hash Join'

Mandatory Conditions

- Applicable to only Equi Join condition
- Not applicable to 'Full Outer' Join type

Apart from the Mandatory Condition, one of the following conditions should hold true:

- 'Broadcast' hint provided on the left input data set and the Join type is 'Right Outer', 'Right Semi', or 'Inner'.
- No hint is provided, but the left input data set is broadcastable as per the configuration '*spark.sql.autoBroadcastJoinThreshold (default 10 MB)*' and the Join type is 'Right Outer', 'Right Semi', or 'Inner'.
- 'Broadcast' hint provided on the right input data set and the Join type is 'Left Outer', 'Left Semi', or 'Inner'.
- No hint is provided, but the right input data set is broadcastable as per the configuration '*spark.sql.autoBroadcastJoinThreshold (default 10 MB)*' and the Join type is 'Left Outer', 'Left Semi', or 'Inner'.
- 'Broadcast' hint provided on both the input data sets and the Join type is 'Left Outer', 'Left Semi', 'Right Outer', 'Right Semi', or 'Inner'.
- No hint is provided, but both the input data sets are broadcastable as per the configuration '*spark.sql.autoBroadcastJoinThreshold (default 10 MB)*' and the Join type is 'Left Outer', 'Left Semi', 'Right Outer', 'Right Semi' or 'Inner'.

'Shuffle Hash Join'

Mandatory Conditions

- Applicable to only Equi Join condition
- Not applicable to 'Full Outer' Join type
- The configuration '*spark.sql.join.prefersortmergeJoin (default true)*' is set to false

Apart from the Mandatory Condition, one of the following conditions should hold true:

- 'shuffle_hash' hint provided on the left input data set and the Join type is 'Right Outer', 'Right Semi', or 'Inner'.
- No hint is provided, but the left input data set is considerably smaller than the right input data set and the Join type is 'Right Outer', 'Right Semi', or 'Inner'.
- 'shuffle_hash' hint provided on the right input data set and the Join type is 'Left Outer', 'Left Semi', or 'Inner'.
- No hint is provided, but the right input data set is considerably smaller than the left data set and the Join type is 'Left Outer', 'Left Semi', or 'Inner'.
- 'shuffle_hash' hint provided on both the input data sets and the Join type is 'Left Outer', 'Left Semi', 'Right Outer', 'Right Semi', or 'Inner'.
- No hint is provided, but both the data sets are considerably smaller and the Join type is 'Left Outer', 'Left Semi', 'Right Outer', 'Right Semi', or 'Inner'.

'Sort Merge Join'

Mandatory Conditions

- Applicable to only Equi Join condition
- Join Keys, identified from the Equi Join condition, are sortable
- The configuration '*spark.sql.join.prefersortmergeJoin (default true)*' is set to true

Apart from the Mandatory Conditions, one of the following conditions should hold true:

- 'merge' hint is provided on any of the input data set, and the Join type could be any.
- No hint is provided, and the Join type could be any.

'Cartesian Join'

Mandatory Conditions

- Join type 'Inner'

Apart from the Mandatory Condition, one of the following conditions should hold true:

- The 'shuffle_replicate_nl' hint is provided on any of the input data sets, the Join condition could be Equi or Non-Equi.
- No hint is provided, the Join condition could be Equi or Non-Equi.

'Broadcast Nested Loop Join'

'Broadcast Nested Loop Join' is the default Join mechanism, when no other mechanisms can be chosen, then 'Broadcast Nested Loop Join' is chosen as the ultimate mechanism to execute any Join type for any Join condition.

> *In case more than one Join mechanism becomes eligible for execution, then the preferred one is chosen in order of 'Broadcast Hash Join' over 'Sort Merge Join' over 'Shuffle Hash Join' over 'Cartesian Join '.*
>
> *Among, Cartesian and Broadcast Nested Loop Join, Broadcast Nested Loop is preferred for Inner, Non-Equi Joins over Cartesian Join when one of the input data set can be broadcasted.*

Last but not the least, partitioning also plays a very important role in the execution efficiency of a given Join mechanism. To know more about partitioning, you can refer to the earlier link.

Hopefully, the story would clear all your confusion and doubts about Join's execution in Apache Spark. In case, any of the doubts still remain, please write in the comments section or send me a message.