**Spark SQL - 3 common joins (Broadcast hash join, Shuffle Hash join, Sort merge join) explained**

* Published on April 4, 2019

**[Ram Ghadiyaram](https://www.linkedin.com/in/ram-ghadiyaram/)**

**[Ram Ghadiyaram](https://www.linkedin.com/in/ram-ghadiyaram/)**

[**7 articles**](https://www.linkedin.com/in/ram-ghadiyaram/detail/recent-activity/posts/)Follow

1.Introduction

2. Spark SQL in the commonly used implementation

2.1 Broadcast HashJoin Aka BHJ

2.2 Shuffle Hash Join Aka SHJ

2.3 Sort Merge Join Aka [SMJ](https://en.wikipedia.org/wiki/Sort-merge_join)

3 Conclusion

Introduction

Join is a common operation in SQL statements. A good table structure can spread data in different tables, make it conform to a paradigm, reduce table redundancy, update fault tolerance, and so on. The best way to establish a relationship between tables and tables is the Join operation.

Spark SQL as a large data area of the SQL implementation, naturally also on the Join operation to do a lot of optimization, today mainly look at the Spark SQL for Join, the common 3 kinds of implementation.

**Spark SQL in the join commonly used to achieve**

**Broadcast HashJoin Aka BHJ**

As we all know, in the database common model (such as star model or snowflake model), the table is generally divided into two types: ***fact***table and ***dimension***table. Dimension tables (small tables) generally refer to fixed, less variable tables, such as contacts, items, etc., the general data is limited. The fact table generally records water, such as sales lists, etc., usually with the growth of time constantly expanding.... means large tables

Because the Join operation is the two tables in the same key value of the record to connect, in SparkSQL, the two tables to do Join the most direct way is based on the key partition, and then in each partition the key value of the same record Come out to do the connection operation. But this will inevitably involve shuffle, and shuffle in Spark is a more time-consuming operation, we should try to design Spark application to avoid a lot of shuffle.

When the dimension table and the fact table for the Join operation, in order to avoid shuffle, we can be limited size of the dimension table of all the data distributed to each node for the fact table to use. executor all the data stored in the dimension table, to a certain extent, sacrifice the space, in exchange for shuffle operation a lot of time-consuming, which in SparkSQL called Broadcast Join, as shown below:

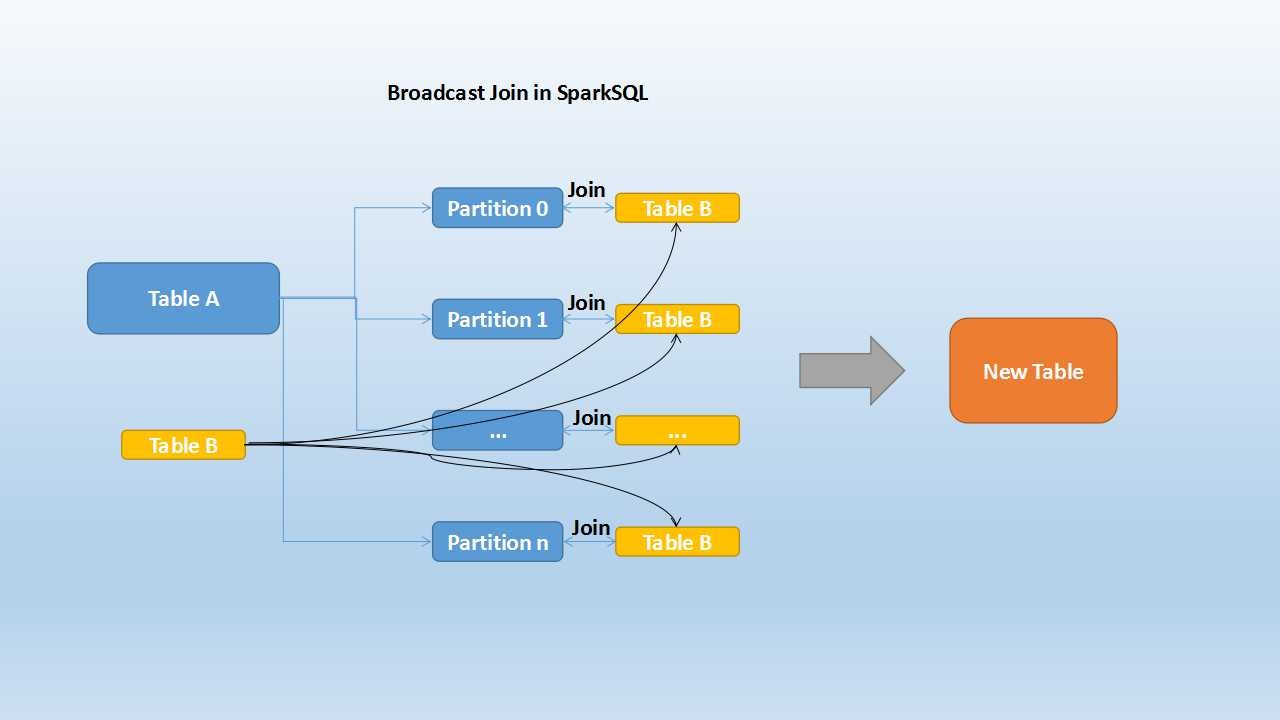


Table B is a smaller table, black means that it will be broadcast to each executor node, Table A of each partition will be through the block manager to get Table A data. According to each record of the Join Key to take the corresponding record in Table B, according to the Join Type to operate. This process is relatively simple, do not repeat them.

Broadcast Join conditions are the following:

*·       Table needs to be broadcast less than***spark.sql.autoBroadcastJoinThreshold***the configured value, default 10M (or add a broadcast join the hint)*

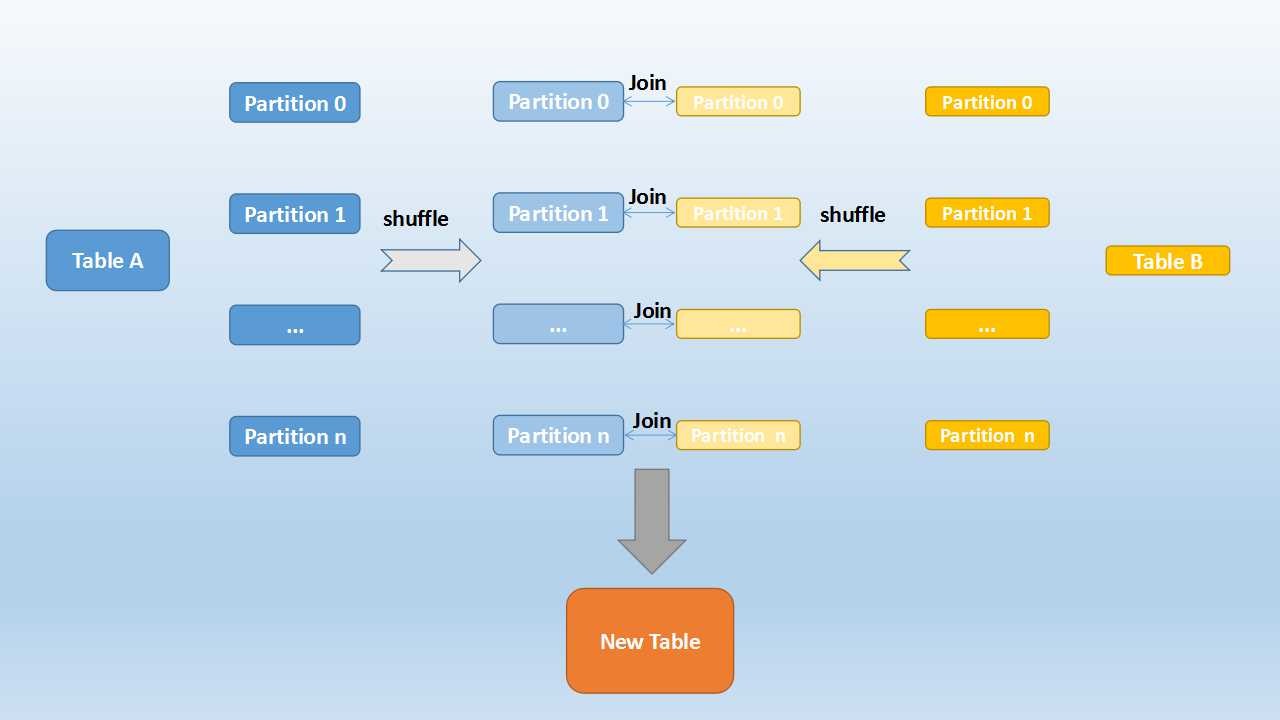
*·       Base table can not be broadcast, such as the left outer join, only broadcast the right table*

Looks like a broadcast is an ideal solution, but does it have any drawbacks? It is also obvious. *This algorithm can only be used to broadcast smaller tables, otherwise the redundant transmission of data is much greater than the cost of shuffle*; In addition, the broadcast needs to be broadcast-ed to the driver of the collector, when there are frequent broadcast, the driver's Memory is also a test.

**Shuffle Hash Join (SHJ)**

When the side of the table is relatively small, we choose to broadcast it out to avoid shuffle, improve performance. But because the broadcast table is first to collect to the driver segment, and then distributed to each executor redundant, so when the table is relatively large, the use of broadcast progress will be the driver and executor side caused greater pressure.

But because Spark is a distributed computing engine, you can partition the large number of data can be divided into n smaller data sets for parallel computing. This idea is applied to the Join is Shuffle Hash Join. Spark SQL will be larger table join and rule, the first table is divided into n partitions, and then the corresponding data in the two tables were Hash Join, so that is to a certain extent, the same time, Reducing the pressure on the side of the driver broadcast side, but also reduce the executor to take the entire broadcast by the memory of the table. The principle is as follows:



Shuffle Hash Join is divided into two steps:

1.      On the two tables were in accordance with the join keys re-zoning, that shuffle, the purpose is to have the same join keys value of the record assigned to the corresponding partition

2.       The corresponding partition in the data for the join, here first small table partition is constructed as a hash table, and then according to the large table recorded in the join keys value out to match

Shuffle Hash Join conditions are the following:

*·       The average size of the partition does not exceed the value configured by***spark.sql.autoBroadcastJoinThreshold***, the default is 10M*[*see here*](https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/execution/SparkStrategies.scala#L164)

*·       Base table can not be broadcast, such as the left outer join, only broadcast the right table*

·       The side of the table should be significantly smaller than the other side, the small side will be broadcast (obviously less than the definition of 3 times the small, here for the empirical value)

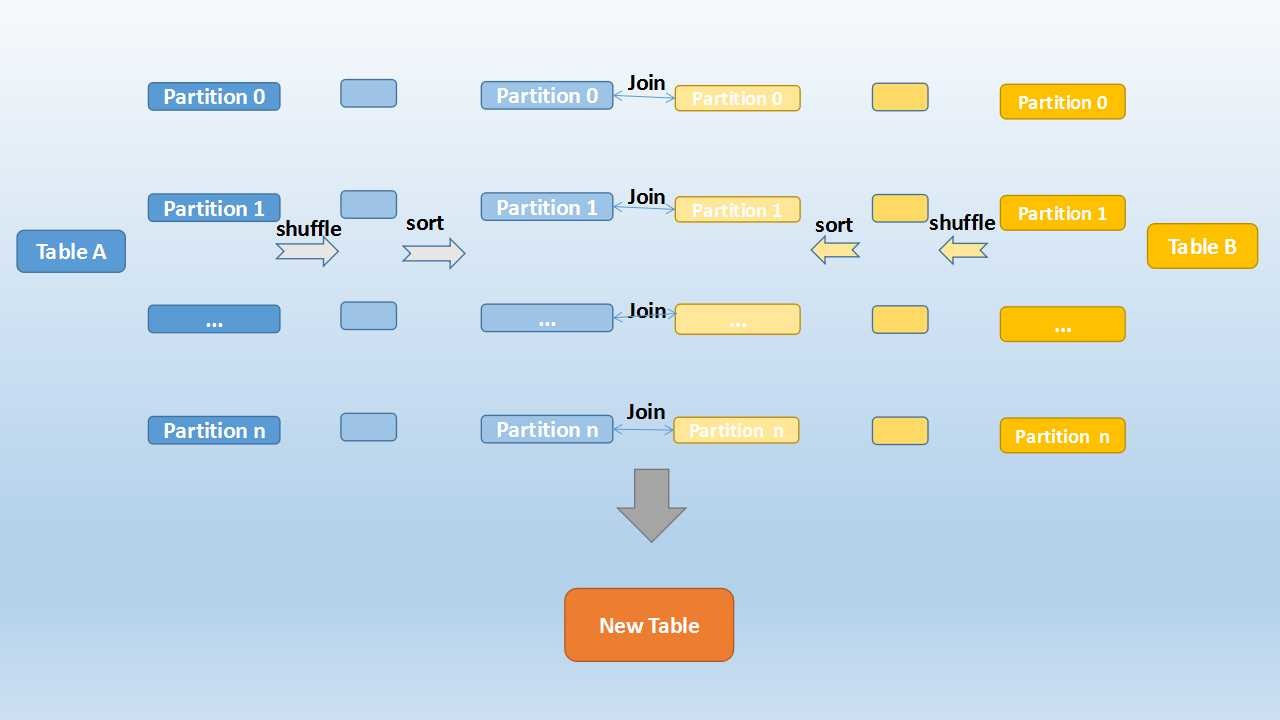
We can see that in a certain size of the table, SparkSQL from the perspective of time and space, the two tables will be re-zoning, and the small table in the partition hash, to complete the join. In maintaining a certain degree of complexity on the basis of minimizing the driver and executor memory pressure, to enhance the stability of the calculation.

**Sort Merge Join -**[**SMJ**](https://en.wikipedia.org/wiki/Sort-merge_join)

The two implementations described above are more applicable to tables of a certain size, but when both tables are very large, it is clear that whatever of them will apply a lot of pressure on the memory. This is because the join is taken when the two are hash join, is the side of the data completely loaded into memory, the use of hash code to take bond values equal to the record to connect.

When the two tables are very large, Spark SQL uses a new algorithm to join the table, that is, Sort Merge Join. This method does not have to load all the data and then into the start hash join, but need to ***sort the data before the join***, as shown below:

You can see that the first two tables in accordance with the join keys were re-shuffle, to ensure that the same value of the join keys will be divided in the corresponding partition. After partitioning the data in each partition, sorting and then the corresponding partition within the record to connect, as shown below:



Note :

[***Sort merge: if the matching join keys are sortable***](https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/execution/SparkStrategies.scala#L123)***then this join is possible*** ...

The property ***spark.sql.join.preferSortMergeJoin*** which controls the behavior of the algorithm

Look very familiar, right? Is also very simple, because the two sequences are orderly, from scratch traverse, hit the same key on the output; if different, left to continue to take the left, and vice versa.

It can be seen, no matter how large the partition, Sort Merge Join do not have a side of the data all loaded into memory, but that is ready to take away, which greatly enhance the large amount of data under the stability of sql join.

***Conclusion :***This article describes the Spark SQL in the 3 ways to achieve, in fact, this is not new. Traditional DB also has this play, Spark SQL only made it a distributed implementation.

This article only from the large theoretical aspects of the introduction of these types of implementation.. if you want how it was implemented then see [spark source code](https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/execution/SparkStrategies.scala)