Hive on Spark: Join Optimizations

Purpose:

The purpose of this document is to summarize the findings of all the research of different joins and describe a unified design to attack the problem in Spark. It will identify the optimization processors will be involved and their responsibilities.

It is not the purpose to go in depth for implementations of the various joins, such as the common-join (HIVE-7384), or the optimized join variants like mapjoin (HIVE-7613), skew-join (HIVE-8406) or SMB mapjoin (HIVE-8202). It will be helpful to refer to the design documents attached on JIRA for those details before reading this document.

MapReduce Summary:

This section summarizes plan-generation of different joins of Hive on MapReduce, which will serve as a model for Spark. We aim to support most of these join optimizations. Priority will be for the automatically-optimized joins, followed by those that need user input, such as hints and metadata.

Over the years, there have been lots of join optimization introduced to Hive beyond the common-join, via Processors (partial transformation of operator-tree or work tree). The following diagram (Figure 1) shows the relationships of different Processors, each of which does a small part in transforming an operator-tree from common-join to one of the optimized join work-trees (mapjoin, bucket mapjoin, SMB mapjoin, or skewjoin).

Processors are represented by boxes in the following diagram: They are split into three types:

- Logical optimization processor (Green): Deals with pure operator-tree (no physical works).
- Work-generation processor (Blue): Deals with operator-tree and takes part in forming a physical work-tree.
- Physical optimization processor (Red): Deals with fully-formed physical work-tree.

Each processor box shows the triggering condition, either a Hive configuration property, or the presence of a certain operator in the tree. So, you can see how to trigger a particular join by following its path through processors and making sure all configurations are triggered and the given operator has been created by previous processors. There are further conditions to do the transform listed on the top, (ie, size of table, etc), that are not be explained by this document, and can be referred from documents of individual joins.

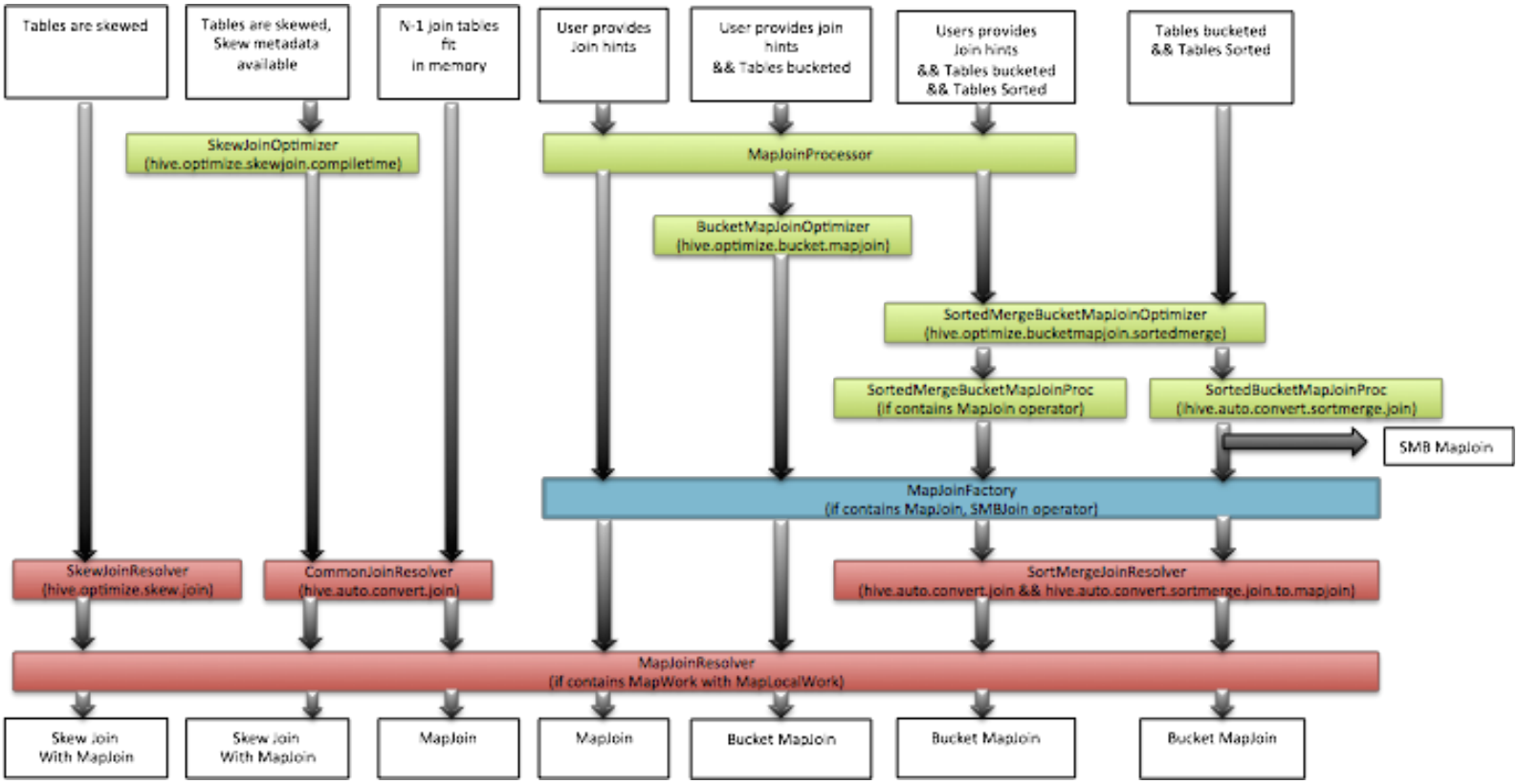
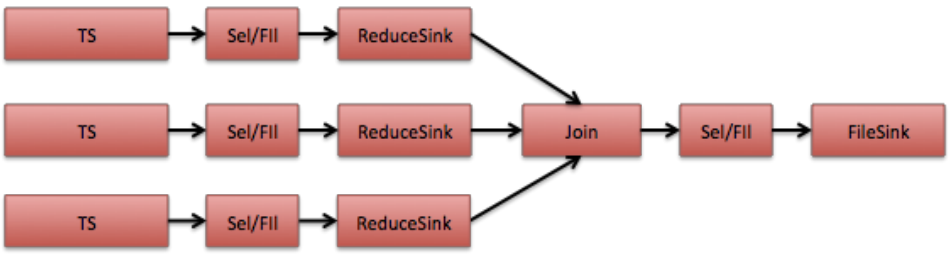


Figure 1. Join Processors for Hive on MapReduce

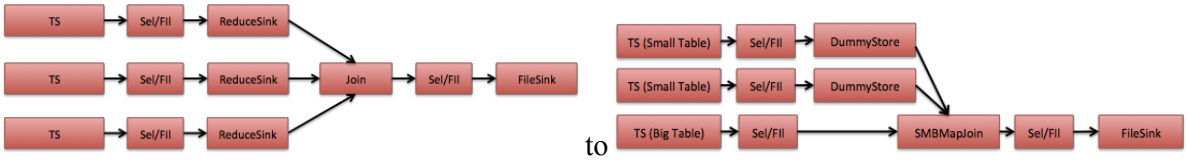
The input arrow at top before any Processor is always an operator-plan for common-join, which is shown as follows. In other words, this is the original join plan if none of the optimizer processors are activated.



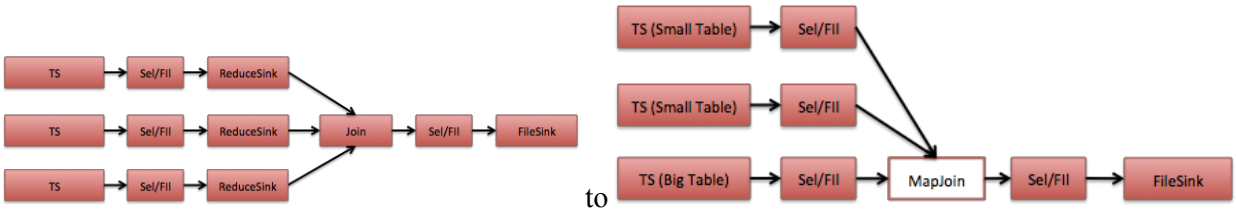
The 'exit' arrows of the join paths shown in Figure 1 are the various optimized join variants. There are of course other exit paths, in which the plan remains unchanged as a common-join upon falling out of any processor's pre-req checks, but they are not shown to simplify the diagram.

Description of the some important Processors follows. Again, there's no space to describe them all, so few key ones are chosen.

- SortedBucketMapJoinProc: This logical optimization processor handles auto-conversion of common-join operator-tree to SMB join operator-tree, for further processing.

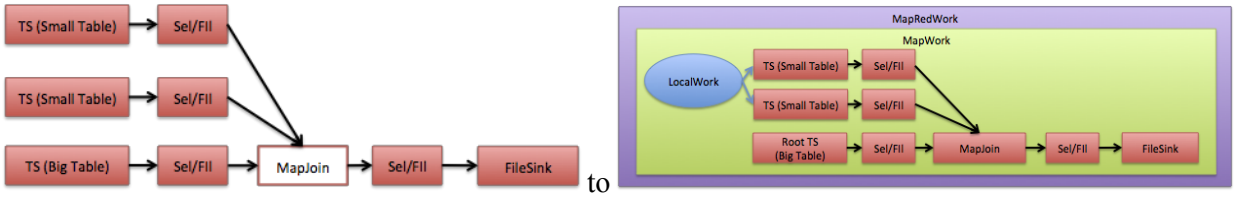


- MapJoinProcessor: This logical optimization processor handles initial conversion of common-join operator-tree to mapjoin operator-tree, for further-processing, when user has given hints to identify the small tables.

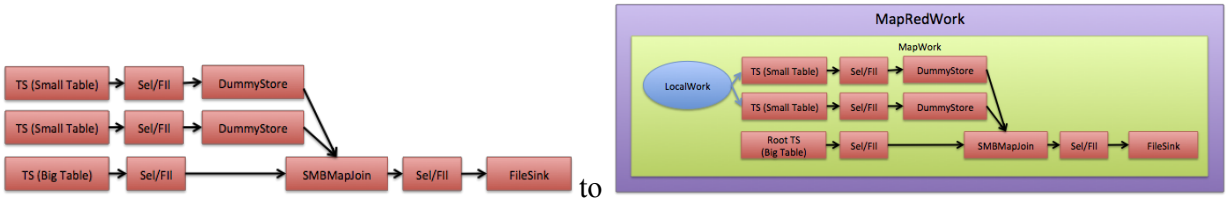


- MapJoinFactory: This follows MapJoinProcessor as part of work-generation phase. It handles the work-tree created from that, adding a local work to identify the small-tables. It also follows after SortedBucketMapJoinProc in the SMB MapJoin path as part of its work-generation phase in a similar way, also adding a local work to identify the small tables for SMB MapJoin.

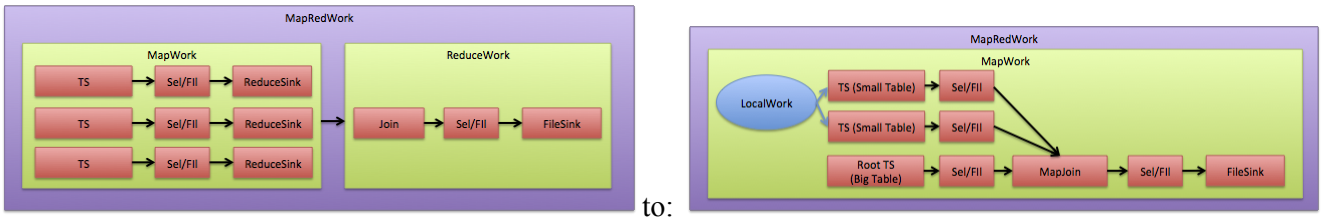
MapJoinFactory MapJoin processing:



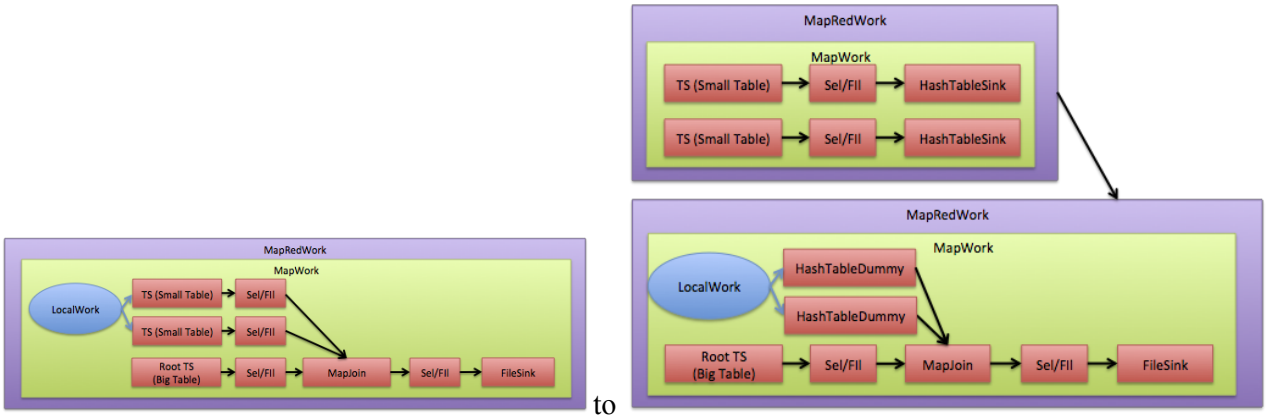
MapJoinFactory SMB MapJoin processing:



- CommonJoinResolver: This handles auto-conversions of joins to mapjoins and goes a separate path than hint-based mapjoin conversions. This takes a working common-join work tree already generated from the common-join operator-tree, and creates an alternative work-tree. The new work-tree consists of a mapwork rooted at the big table. Pointers to the small table are retained in the new work via the LocalWork data structure.



- MapJoinResolver: Here, the two mapjoin paths (hint and non-hint mapjoins) unite again. One last step for both results is to make it ready for physical execution on MR cluster, described in detail in below section "Spark MapJoin". The MapJoinResolver separates the single work into two works. First a local MapRedWork dealing with small tables, ending with HashTableSink writing the hashmap files. Then a MapRedWork dealing with big table, loading from small-table hashmap files via HashTableDummyOp.



A brief summary of all the processor-paths of possible join plans shown in Figure 1:

1. Skewjoin (compile-time)
    a. SkewJoinOptimizer: From a common-join operator tree, creates two join operator-trees connected by union operator. These will represent a join with skew key, and a join without it.
    b. One or both reduce-side join might be converted to mapjoin by CommonJoinResolver, see auto-mapjoin for more details.
2. Skewjoin (runtime)
    a. SkewJoinResolver: Create conditional work after the original common-join work, which is a list of mapjoin works. These will handle the skew keys.
    b. MapJoinResolver: Final preparation for mapjoin works as described.
3. Auto-mapjoin
    a. CommonJoinResolver: Convert common-join operator tree to mapjoin operator-tree, with big/small table(s) identified on the Mapjoin operator, as described.
    b. MapJoinResolver: Final preparation for mapjoin works as described.
4. Map join query with hints

    a. MapJoinProcessor: Convert common-join operator tree to mapjoin operator-tree, with big/small table(s) identified on the Mapjoin operator, as described.
    b. MapJoinFactory: Adds localWork pointing to small tables in mapjoin work, as described.
    c. MapJoinResolver: Final preparation for mapjoin works as described.
5. Bucket map join query with hints.
    a. MapJoinProcessor: Convert common-join operator tree to mapjoin operator-tree, with big/small table(s) identified on the Mapjoin operator, as described.
    b. BucketMapJoinProcessor: Add bucketing information to MapJoin op.
    c. MapJoinFactory: Adds localWork pointing to small tables in mapjoin work, as described.
    d. MapJoinResolver: Final preparation for mapjoin works as described.
6. SMB join query with hints
    a. MapJoinProcessor: Convert common-join operator tree to mapjoin operator-tree, with big/small table(s) identified on the Mapjoin operator, as described.
    b. SortedBucketMapJoinProc: Convert mapjoin operator-tree to SMBMapJoin operator-tree. Add DummyOp to small-tables.
    c. MapJoinFactory: Adds localWork pointing to small tables in SMBMapjoin work, as described.
    d. May be converted back to MapJoin (see #8 for details).
7. Auto-SMB join
    a. SortedMergeBucketMapJoinProc: Convert mapjoin operator-tree to SMBMapJoin operator-tree. Add DummyOp to small-tables.
    b. MapJoinFactory: Adds localWork pointing to small tables in SMBMapjoin work, as described.
    c. May be converted to MapJoin (see #8 for details).
8. SMB join that converts to mapjoin
    a. SMBJoin operator-tree constructed as mentioned in #6, #7 above.
    b. SortedMergeJoinResolver: For each possible big-table candidate, create a mapjoin work. These will have LocalWork data structures to keep track of small-tables. Create ConditionalWork with all of these mapjoin works (with the original SMBJoin work as the backup task of each one), and the original SMBJoin work as the last option.
    c. MapJoinResolver: For each mapjoin work created, final preparation as described.

Tez Comparison

Hive on Tez is still evolving. They currently disable all logical-optimizer processors, and use a processor called “ConvertJoinMapJoin” located in the work-generation phase. It utilitzes stats annotated on the operator-tree to make some decisions as to what join to take. It will directly create plans for the following joins:

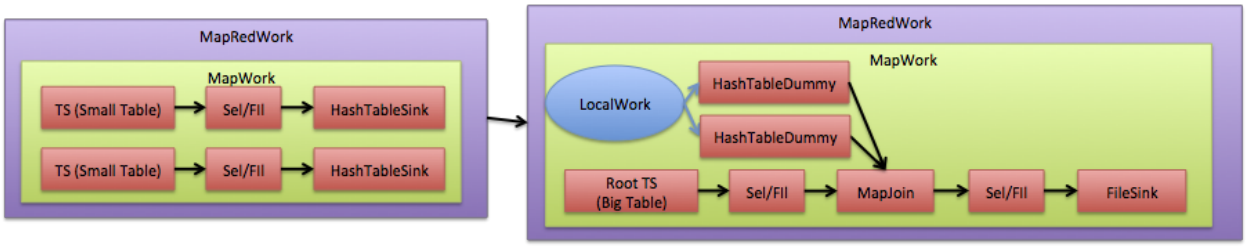
- MapJoin
- SMBJoin

These look different than MapReduce plans, and are based on the Tez physical feature "broadcast-edge". See JIRA's of those joins for more details.

Spark MapJoin

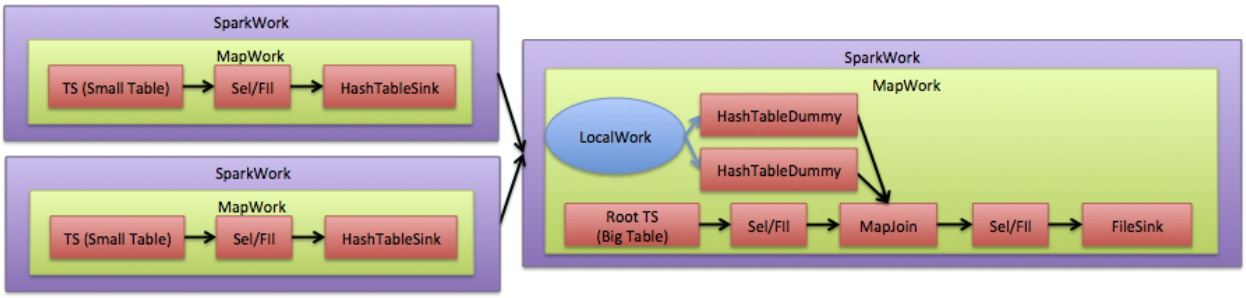
For most of the joins for Hive on Spark, the overall execution will be similar to MR for the first cut. Thus, a similar work-tree as in MR will be generated, though encapsulated in SparkWork(s) instead of MapRedWork(s).

One difference is implementation of mapjoin, which is worth spending some time discussing. Recall the mapjoin work-tree in MapReduce:



1. Run the MapredLocalWork containing small-table(s)' operator-tree, ending it with a HashTableSink op that dumps to file. This is made into a distributed cache.
2. Run the MapWork for the big table, which will populate small-table hashmap from the distributed cache file using HashTableDummy's loader.

Spark mapjoin has a choice to take advantage of faster Spark functionality like broadcast-variable, or use something similar to distributed-cache. A discussion for choosing MR-style distributed cache is given in "small-table broadcasting" document in HIVE-7613, though broadcast-variable support might be added in future. Here is the plan that we want.



1. Run the small-table SparkWorks on Spark cluster, which dump to hashmap file (this is main difference with MR, as the small-table work is distributed).
2. Run the SparkWork for the big table on Spark cluster. Mappers will lookup the small-table hashmap from the file using HashTableDummy's loader.

For bucket map-join, each bucket of each small table goes to a separate file, and each mapper of big-table loads the specific bucket-file(s) of corresponding buckets for each small table.

Spark Join Design

Let's redraw the processor diagram for Hive on Spark. There are several other points to note in this section:

- Logical optimizers are mostly re-used from Hive on MapReduce, directly or slightly modified.
- GenSparkWork is the first of the work-generator processors, which creates SparkWorks.

There are also some minor differences (improvements) over original MapReduce, beyond the one mentioned in Spark MapJoin section.

- Hive on Spark supports automatic bucket mapjoin, which is not supported in MapReduce. This is done in extra logic via SparkMapJoinOptimizer and SparkMapJoinResolver.
- Hive on Spark's SMB to MapJoin conversion path is simplified, by directly converting to MapJoin if eligible.

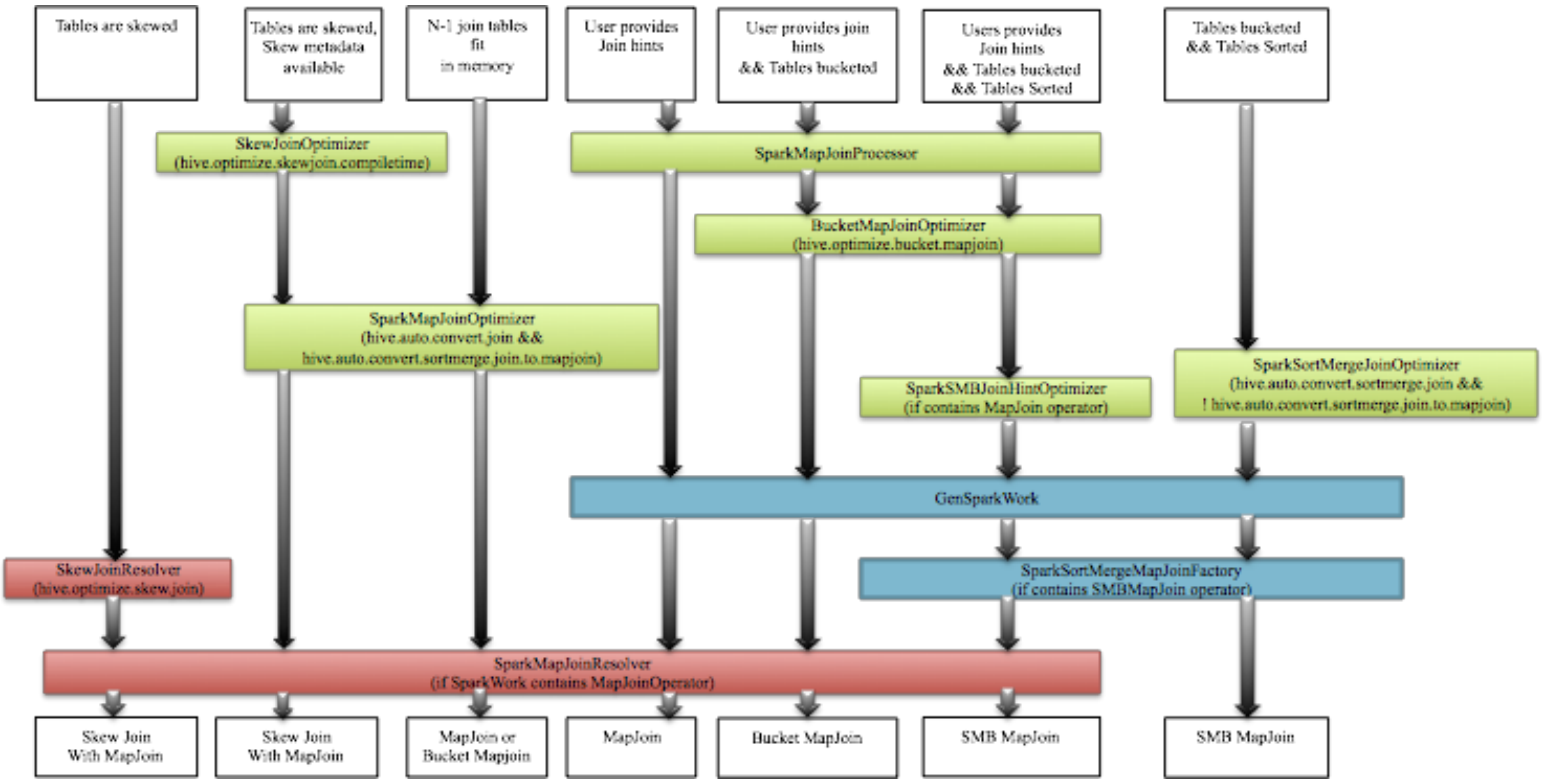
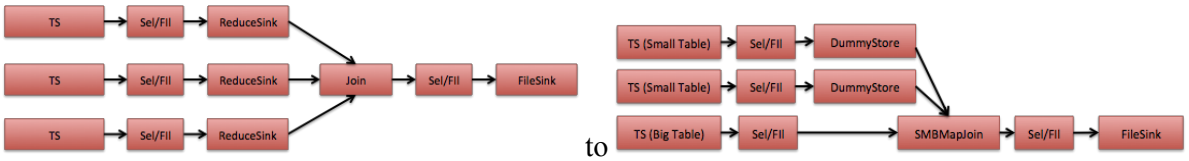


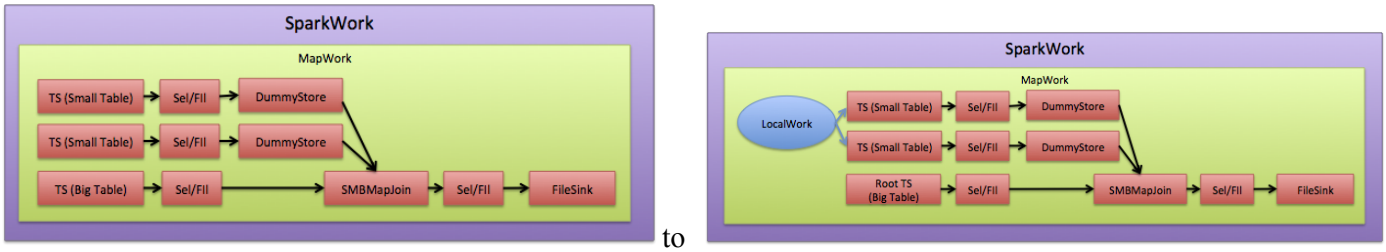
Figure 2: Join Processors for Hive on Spark

Again, we first explore some of the interesting processors:

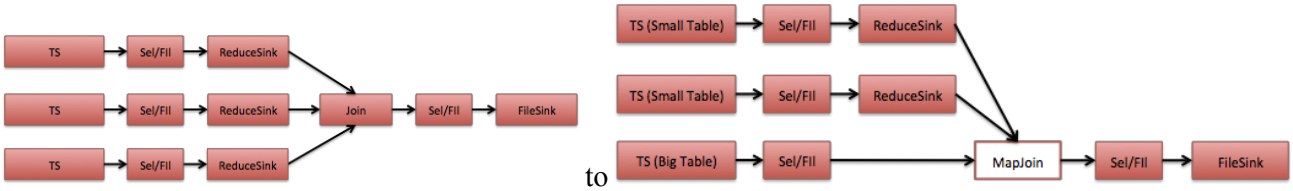
- SparkSortMergeJoinOptimizer: Like SortrtedBucketMapJoinProc, this logical optimization processor handles auto-conversion of common-join to SMB join operator-tree, for further processing.



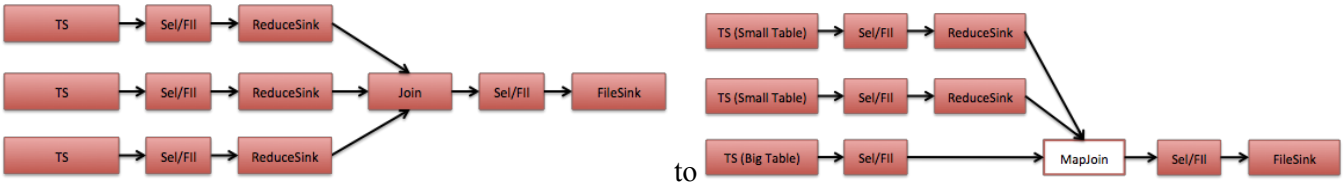
- SparkSortMergeMapJoinFactory: This takes a MapWork with operator-tree already with a SMBMapJoin operator and big/small table(s) identified, and creates a LocalWork pointing at small tables.



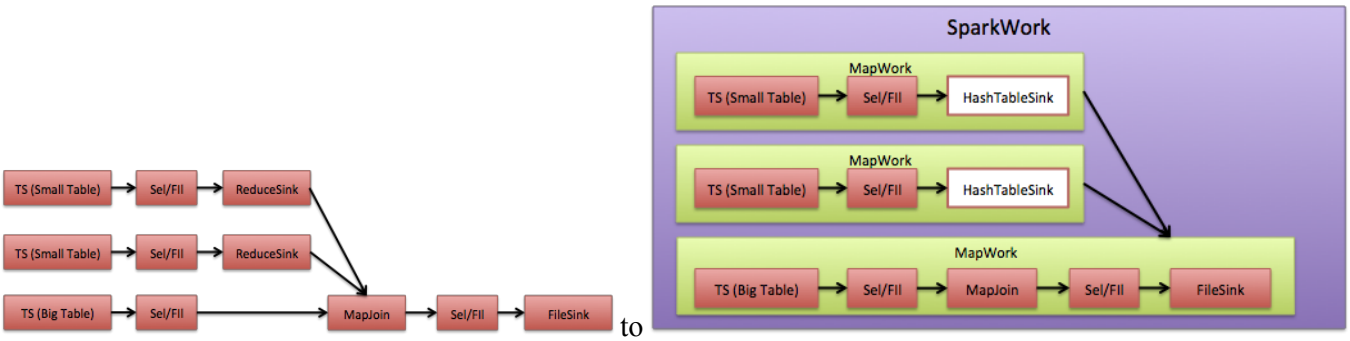
- SparkMapJoinProcessor: Like MapJoinProcessor, this logical optimization processor handles initial conversion of common-join to mapjoin, for further-processing, when user has given hints to identify the small tables. Final operator-tree is just a bit different than with MapReduce, with ReduceSinks of small table branch(es) still attached.



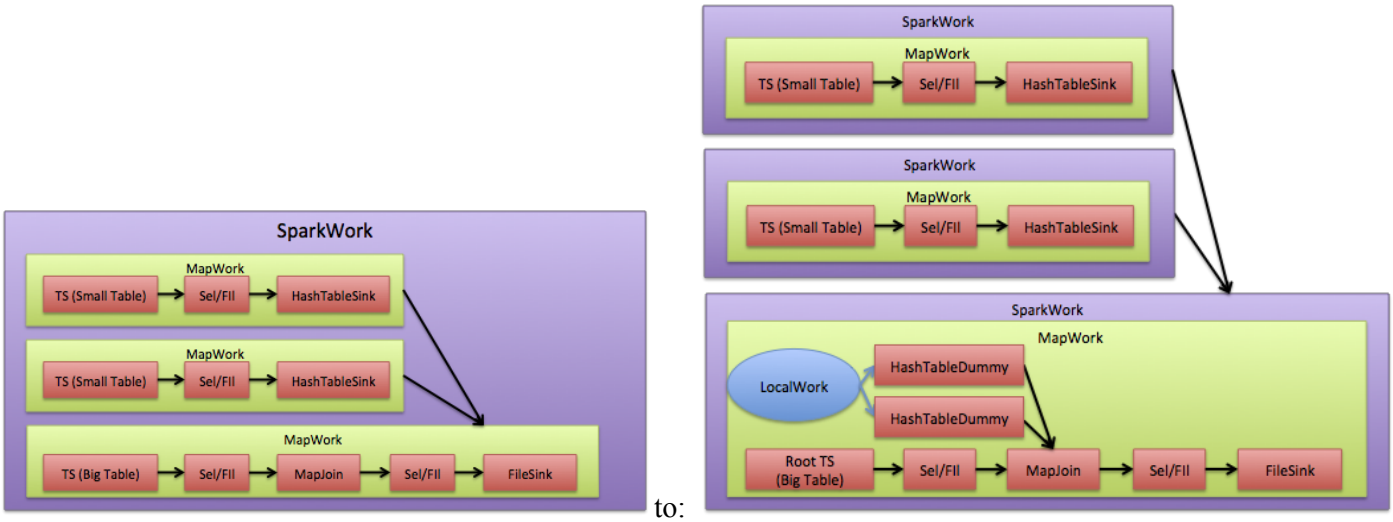
- SparkMapJoinOptimizer: Similar to MapReduce's MapJoinProcessor and Tez's ConvertJoinMapJoin, this will transform a common join operator-tree to a mapjoin operator-tree, by identifying the big and small tables via stats. Like Tez's processor, it removes the reduce-sinks from the big-table's branch but keeps them for the small-tables.



- GenSparkWork/SparkReduceSinkMapJoinProc: During the work-generation phase, a combination of these processors will draw the appropriate work boundaries around the mapjoin operator tree. It also transforms ReduceSinks into HashTableSinks.



- SparkMapJoinResolver: Again, the various mapjoin paths (hint and automatic) converge at this final processor before execution on Spark cluster. This takes a single SparkWork with MapJoin operator and big/small table(s) identified, and splits it into dependent SparkWorks, with the small-table SparkWork(s) being parents of the big-table SparkWork. This will be sent to Spark cluster to run mapjoin as described in the section "Spark MapJoin". The LocalWork data structure is created within the dependent (big-table) SparkWork to contain HashTableDummy's which load small-table hashmap files.



And the summary of each join plan's processor path of Figure 2.

1. Compile-time skewjoin without mapjoin: Logical optimizer completely re-used from MapReduce.
    a. SkewJoinOptimizer: This logical-optimizer processor is reused, to create two join plans out of one connected by union.
    b. Follows auto-conversion to MapJoin path.
2. SMB MapJoin (with hints): again, logical optimizers are mostly similar to those in MapReduce.
    a. SparkMapJoinProcessor/BucketMapJoinOptimizer/SparkSMBJoinHintOptimizer: Almost identical to MapReduce versions, these transform the operator tree to include SMBMapJoinOp with big/small table(s) identified.

    b. GenSparkWork: Generate the SparkWork, which is an SMBMapJoin operator-tree rooted at big-table TS.
    c. SparkSortMergeJoinFactory: Attach Localwork data structure pointing to small tables in the SMBMapJoin work as described.
3. SMB MapJoin (without hints): again, logical optimizers are mostly similar to those in MapReduce
    a. SparkSortMergeJoinOptimizer: Almost identical to MapReduce version, this transforms the common-join operator tree to SMB mapjoin operator-tree, with big/small table(s) identified on SMBMapJoin operator, as described.
    b. GenSparkWork: Generate the SparkWork, which is an SMBMapJoin operator-tree rooted at big-table TS.
    c. SparkSortMergeJoinFactory: Attach Localwork data structure pointing to small tables in the SMBMapJoin work as described.
4. Auto-mapjoin: Mostly a rewrite, unable to reuse the MapReduce processors.
    a. SparkMapJoinOptimizer: Based on stats, converts a common-join operator tree to mapjoin operator-tree, with big/small table(s) identified in MapJoinOp, as described.
    b. GenSparkWork: Generate the SparkWork, which has MapJoin operator-trees rooted at various table TS's.
    c. SparkMapJoinResolver: Create two SparkWorks to achieve the mapjoin, as described.
5. Mapjoin via hints: again, logical optimizers are mostly similar to those in MapReduce
    a. SparkMapJoinProcessor: Almost identical to MapReduce version, this transforms the common-join operator tree to mapjoin operator-tree, with big/small table(s) identified in MapJoinOp, as described.
    b. GenSparkWork: Generate the SparkWork, which has MapJoin operator-trees rooted at various table TS's.
    c. SparkMapJoinResolver: Create two SparkWorks to achieve the mapjoin, as described.
6. SMB joins converted to mapjoin:
    a. This route is avoided unlike in MapReduce. If conditions are met, join is directly sent to SparkMapJoinOptimizer and SparkMapJoinResolver, just like a normal auto-mapjoin.
7. Skew join (runtime):
    a. SparkSkewJoinResolver: Takes a SparkWork with common join, and turn it in a conditional work. Then add additional SparkWork with mapjoin operator-tree as backups in the conditional work. These will handle the skew keys.
    b. SparkMapJoinResolver: For each backup SparkWork with mapjoin, create two SparkWorks to achieve the mapjoin, as described.
8. Auto-bucket join
    a. SparkMapJoinOptimizer: Extra logic here beyond auto-mapjoin conversion to support auto-bucket mapjoin conversion.
    b. SparkMapJoinResolver: Extra logic here beyond auto-mapjoin conversion to support auto-bucket mapjoin conversion.