Skew join background

Optimization for skew join can happen both during compilation and at runtime. Two configure properties are used to control it respectively: ***hive.optimize.skewjoin.compiletime*** and ***hive.optimize.skewjoin***.

The basic idea of skew join optimization is to perform a common join for non-skew keys, and perform map join for skewed keys. Suppose we have:

A join B on A.id=B.id

And A skews for id=1. Then we perform the following two joins:

1. A join B on A.id=B.id and A.id!=1
2. A join B on A.id=B.id and A.id=1

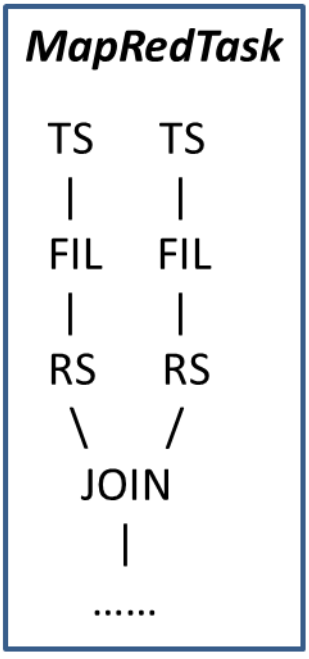
If B doesn't skew on id=1, then #2 will be a map join. Finally we union the result from the two joins to get the final results.

Runtime optimization

So far as I've found, Tez doesn't seem to support runtime skew join optimization. Therefore in this part we'll focus on the MR mode.

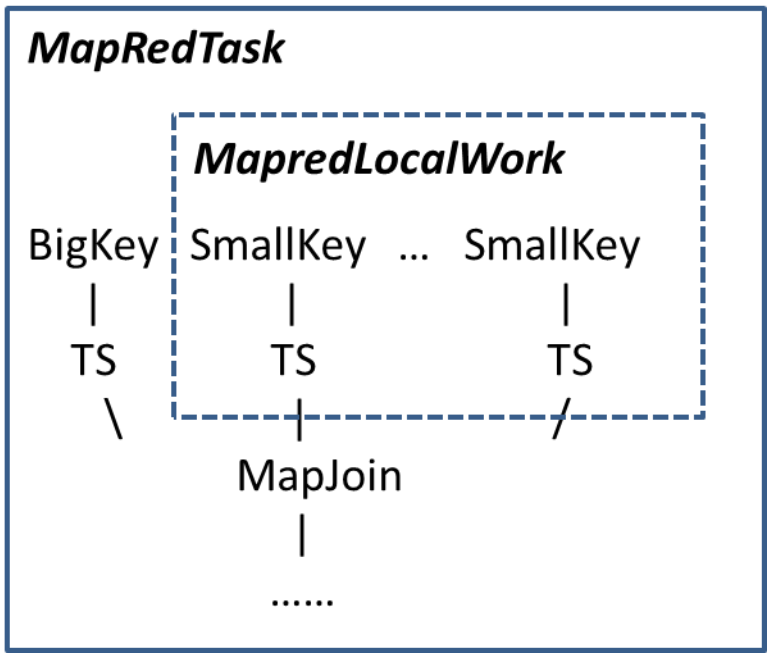
SkewJoinResolver is used to create the conditional tasks containing the map join tasks. It creates directories for skew keys in each table. Using our example above, we'll have A_bigKey and B_bigKey to store records on skewed keys in A and B respectively. For a table's big key, we also have directories to store records from other tables on the same key. Therefore we have B_forA_smallKey and A_forB_smallKey. If there're N joining tables, we end up have N*N such directories.

Once we have all the directories set up, we can create the conditional task. Suppose the original join task is quite simple:



OriginalJoin

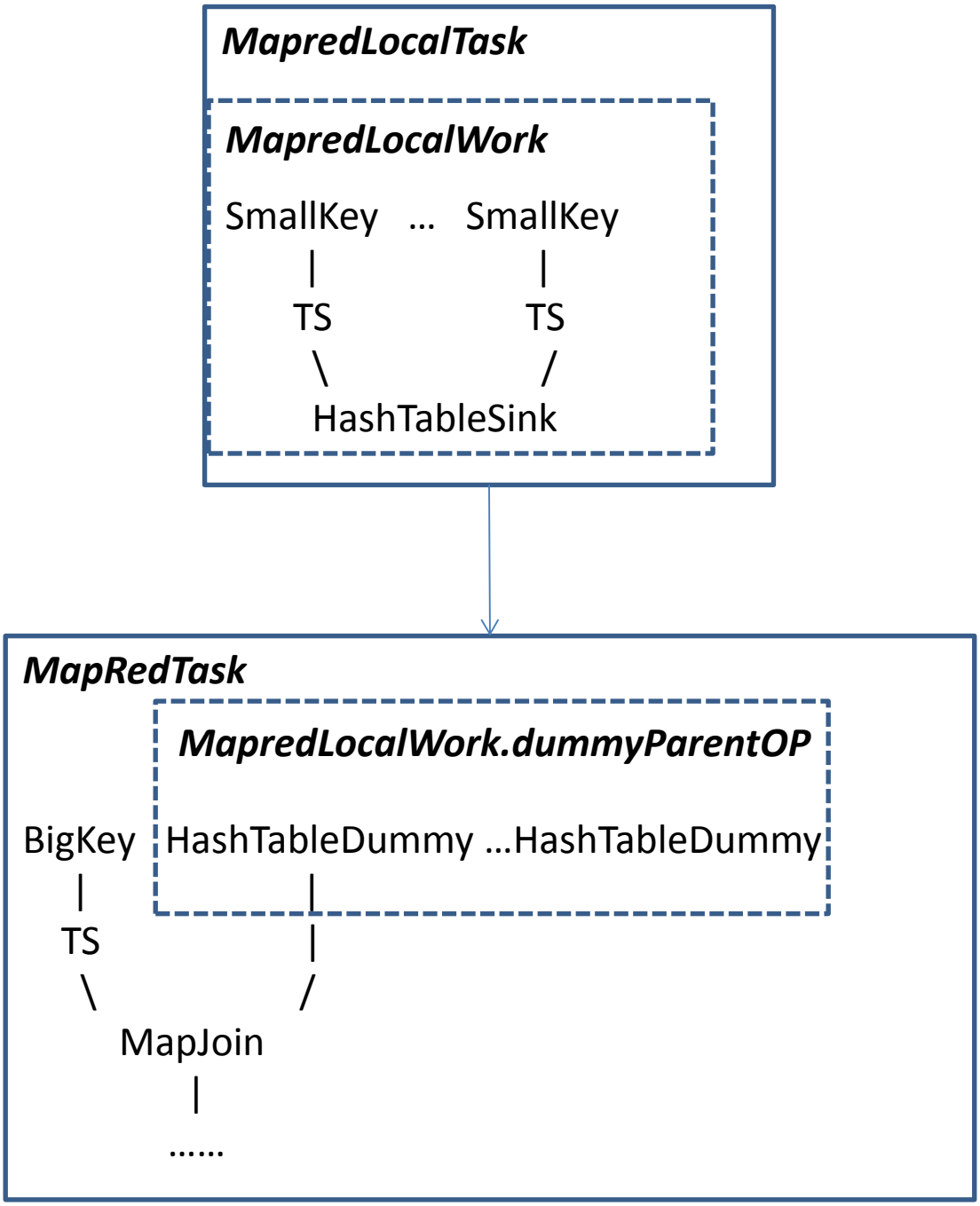
Then a conditional task is created with a list of map join tasks like this:



MapJoinTask

There's such a task for each joining table, except the last one. Child operator of the MapJoin is cloned from the original join's child. And the conditional task is marked as child of the OriginalJoin.

Then the map join task is modified by MapJoinResolver. MapJoinResolver basically separates the MapredLocalWork from the map join task and wraps it in a MapredLocalTask. The child operator for small keys' TSes will also be converted to a hash table sink operator. So the map join task is transformed to something like this:



TransformedMapJoinTask

At runtime, join is done on a key-group basis. RowContainer is used to cache all the data within a key group for each joining table. SkewJoinHandler is responsible for tracking the number of rows in each table. If the #row >= the threshold (***hive.skewjoin.key***), SkewJoinHandler decides that it finds skew in this key group. The current table is marked as the big table and all other tables will be the small tables. At the end of the key group, SkewJoinHandler dumps all the RowContainers to corresponding big/small key dirs. So the join operator will skip processing this key group and leave it to following map join tasks.

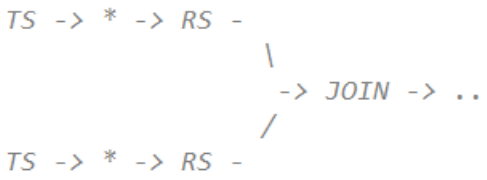
When the original join finishes, the conditional task is resolved by checking whether there's data in the big key directories. If yes, the corresponding map join task gets executed (*For N tables there're N-1 map joins. If skew only exists in the last table, the optimization won't have any*

effects?). If all the directories are empty, no map join is needed because all the records get processed in the original join.
Since the map join has same FS as the original join (same output path), the fetch task will eventually get output from all the joins and produce the final result.

Compile time optimization

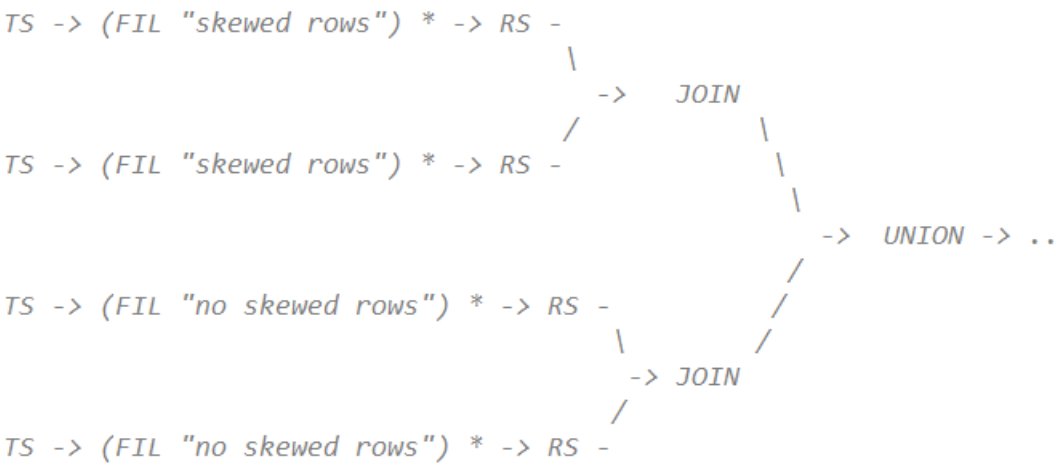
SkewJoinOptimizer is the logical optimizer used for compile time optimization. The fundamental idea is quite similar to runtime optimization, except that everything is determined during compilation. The following outlines the main functionality of SkewJoinOptimizer:

- Match the pattern: TS -> … -> RS -> JOIN.
- Clone the whole operator tree from all preceding TSes to the current JOIN. If JOIN is followed by a SEL, the SEL is cloned as well. It also traverses the tree from JOIN to TS and checks if all the operators support skew join optimization. The optimization terminates if there's any ancestor not supporting the optimization. Currently, only TS, FIL and SEL support the optimization.
- Retrieve skew information from TSs' metadata and keep it in ***skewedValues:Map<List<ExprNodeDesc>, List<List<String>>>***. For example, if T1 and T2 joins on C1, C2 and C3. And T1 skews on {(C1=1 and C4=2), (C1=3 and C4=4)}, T2 skews on {(C1=5 and C2=6), (C1=7 and C2=8)}. Then skewedValues will keep: {(c1) -> ((1), (3)), (c1,c2) -> ((5,6),(7,8))}.
- According to skewedValues, insert predicates to the original and cloned operator trees. The original operator tree will be filtered by skewKey=skewValue, and the cloned operator tree will be filtered by skewKey!=skewValue. For the above example, the original operator tree will have the predicate "C1=1 or C1=3 or (C1=5 and C2=6) or (C1=7 and C2=8)". The cloned operator tree will have the opposite.
- A UNION is created as the child of the original and cloned JOINs (or SELs).
- To sum up, if the original operator tree is:



Original OP tree

After optimization, it'll be:



Optimized OP tree

Similar to runtime optimization, we expect the join for skewed keys to be done as a map join. I think that's left to the standard procedure of converting common join to map join (in physical layer for MR).

Compile time optimization relies on metadata. The skew information of a table can be specified with CREATE or ALTER statement. If no metadata is available, the optimization has no effect.

What's the difference in behavior between runtime & compile time optimization?

What happens if they're both enabled?

Design for spark

Runtime optimization

Some design considerations:

- The optimization in MR is done in a two-step manner. We can do similar thing for spark: first create conditional task and add it as child of the original join task, the conditional task has a list of map join tasks for each joining table except the last one. Then each of the map join task is further processed to get the desired task/work/operator graph.
- The first step has to be done in physical layer because new task needs to be created. To do this, we'll have to copy from MR physical resolvers. A simple prototype can be found here: https://github.com/lirui-intel/hive/compare/lirui-intel:spark...HIVE-8536 .
- Ideally we can rely on map join optimization for spark to do second step (HIVE-7613).

Compile time optimization

Compile time optimization seems much simpler and more independent. Ideally it should come for free on spark, as long as we have RSJ and UNION supported.