Small Table Broadcasting for Hive on Spark

We have been evaluating different approaches for the purpose of broadcasting small tables to support Hive's map join with Spark. These approaches based on either Spark's broadcast variable or MR's distributed cache. The following summarizes the pros and cons of each approach.

Spark Broadcast Variable

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms (BitTorrent) to reduce communication cost.

The advantage of using broadcast variable is its fast broadcast algorithm. However, this advantage also comes with a cost, mainly memory consumption at both the driver and the executor node. Here are disadvantages:

1. To broadcasting data on HDFS or from a RDD, the data needs to be shipped to the driver. This requires a round trip for the data, and thus add costs in terms of time in broadcasting.
2. The driver has to keep the broadcast data live in memory, even after the variable is broadcast. This is to support executor fault tolerance.
3. The broadcasting is for all nodes, taking memory space, even if a node only needs part of the data.

With these characteristics, memory consumption at the driver, as well as at the executor, is a big concern.

The driver has to keep live all variables, which is a great concern. For a Spark job, there could be many small tables to broadcast. While each table might be small, the total amount of data, thus the memory space required, can be significant. Luckily, broadcast variables at Spark driver is also controlled by a block manager (similar to that at an executor), which can spill these variables to disc when per-allocated memory space is exhausted. This is a good news, mitigating our concern that it may require a memory space to hold all broadcast variables. Nevertheless, each variable still needs to be fit into the memory.

Moreover, the data size on disk can be compressed and small, the corresponding memory space can be much larger. While it's possible to broadcast compressed data, this adds complexity.

At the executor, all the broadcast variable needs to be available, even if only a portion of the data is needed. This the case for Hive's bucket join.

Finally, using Spark's broadcast variable to support map join incurs greater development cost, as it's completely different from distributed cach.

Distributed Cache

MR's way of broadcasting small tables is to use distributed cache. Hive executed local jobs that build

hash tables representing small tables. These hash tables are originally stored in local disk (at HiveServer2 host), but copied to distributed cache. MR framework makes sure that the files in distributed cache are make available at each worker node when the job starts.

One advantage of this is less memory consumption. The driver doesn't need to keep a copy of the data, and the executor only needs to bring the needed data in memory.

Another advantage of using MR's way is that we can reuse or borrow a lot of code from MR.

One disadvantage of this approach is that Spark doesn't have support for distributed cache as in MR. While it's possible to use it when Spark is running on Yarn, it require hacks and doesn't work with Spark for other type of deployment.

The way to handle this is to use Spark's utilities that mimics the functionality of distributed cache. First, the driver set a HDFS file to a large replication factor (say, the same as the number of data nodes). Then, the driver call SparkContext.addFile(Path filePath) make the files accessible at works. Lastly, the executor (or our code for closures) can access the file by calling SparkFiles.getFile(Path filePath). Hopefully, this way will enable executor reading from local disk, achieving what distributed cache offers.

Conclusions

While broadcast variable might offer performance advantages, especially with storage persistent management at the driver, it will require significant amount work to integrate with it. On the other hand, the distributed cache approach is similar to what Hive on MR does, enabling us to reuse or borrow existing code with significantly less amount of work. For instance, we may be able to reuse HashTableSink for generating small table data and HashTableLoader for load the data into the join operator. We don't need to ship the data to the driver in order to "broadcast" it.

More importantly, this approach allows us move quickly to make map join work with relevantly less work. We can come back to do performance profiling and POC using the other approach (using broadcast variable). Given the amount of work we have, this is a prevailing advantage.

Here is the outline of what needs to be done when choosing distributed cache approach:

1. MapJoinResolver derives two SparkTasks from each map join operator: one for all small tables and one for the join. The second one depends on the first one. The first task's work contains a union of work graphs, one for each small table.

2. The work graph for each small table ends a HashTableSink in the operator tree, as it does for MR.

3. The first task will be run as a regular Spark job (distributed). With this, hash tables to be built will be on HDFS after the job is executed. This avoids shipping the hash tables from local to HDFS. To make this happen, however, we need to make HashTableSink distributable.

4. The map operator will do what it does today, using HashTableLoader to load the broadcast hash tables in #3. We might need to reuse or enhance existing HashTableLoader for MR for this.

As it can seen, the work involved is localized and contained. It should quickly make something work without too much effort.

After this work is done, we can do some performance measurement and profiling. If it's determined that broadcast variable is prevalent when memory isn't a concern, we may take advantage of that by adopting a hybrid approach. This would be a future enhancement, though.