Hybrid Hybrid Grace Hash Join, v1.0

Overview

We are proposing an enhanced hash join algorithm called “hybrid hybrid grace hash join”. We can benefit from this feature as illustrated below:

- The query will not fail even if the estimated memory requirement is slightly wrong.
- Expensive garbage collection overhead can be avoided when hash table grows.
- Join execution using a Map join operator even though the small table doesn't fit in memory, as spilling some data from the build and probe sides will still be cheaper than having to shuffle the large fact table.

The design was based on Hadoop’s parallel processing capability and significant amount of memory available.

See HIVE-9277 for the current status of this work.

Scope

This new join algorithm will only work with Tez. It does not support Map Reduce currently.

Notation and Assumptions

The goal is to compute the equi-join result of two relations labeled R and S. We assume R is the small table with smaller cardinality, and S is the big table with bigger cardinality.

Brief Review on Hash Join Algorithms

Simple Hash Join

Also known as Classic Hash Join, it is used when the hash table for R can entirely fit into the memory.

In general, all hash join algorithms described here have two phases: Build phase and Probe phase. During the build phase, a hash table is built into memory based on the joining column(s) from the small table R. During the probe phase, the big table S is scanned sequentially and for each row of S, the hash table is probed for matching rows. If a match is found, output the pair, otherwise drop the row from S and continue scanning S.

When the hash table for R cannot wholly fit in the memory, only part of the hash table for R will be put in memory first and S is scanned against the partial hash table. After the scan on S is completed, the memory is cleared and another part of hash table of R is put in memory, and S is scanned again. This process can repeat more times if there are more parts of the hash table.

GRACE Hash Join

Apparently when the size of small table R is much bigger than memory, we end up having many partial hash tables of R loaded in the memory one by one, and the big table S being scanned multiple times, which is very expensive.

GRACE hash join brings rounds of scanning the big table from many times down to just twice. One for partitioning, the other for row matching. Similarly, the small table will also be scanned twice.

Here is the detailed process.

1. Small table R is scanned, and during the scan a hash function is used to distribute the rows into different output buffers (or partitions). The size of each output buffer should be specified as close as possible to the memory limit but no more than that. After R has been completely scanned, all output buffers are flushed to disk.
2. Similarly, big table S is scanned, partitioned and flushed to disk the same way. The hash function used here is the same one as in the previous step.
3. Load one partition of R into memory and build a hash table for it. This is the build phase.
4. Hash each row of the corresponding partition of S, and probe for a match in R’s hash table. If a match is found, output the pair to the result, otherwise, proceed with the next row in the current S partition. This is the probe phase.
5. Repeat loading and building hash table for partitions of R and probing partitions of S, until both are exhausted.

It can be seen there are extra partitioning steps in this algorithm (1 & 2). The rest of the steps are the same as Classic Hash Join, i.e., building and probing. One assumption here is all partitions of R can completely fit into the memory.

Hybrid GRACE Hash Join

It is a hybrid of Classic Hash Join and GRACE Hash Join. The idea is to build an in-memory hash table for the first partition of R during the partitioning phase, without the need to write this partition to disk. Similarly, while partitioning S, the first partition does not have to be put to the disk since probing can be directly done against the in-memory first partition of R. So at the end of the partitioning phase, the join of the first pair of partitions of R and S has already been done.

Comparing to GRACE hash join, Hybrid GRACE hash join has an advantage of avoiding writing the first partitions of R and S to disk during the partitioning phase and reading them back in again during the probing phase.

Hash Join in Hive

It is also known as replicated join, map-side join or mapjoin. When one side of the data is small enough to fit in memory of a mapper, it can be copied to all the mappers and the join will be performed only in the map phase. There is no reduce phase needed.

Motivation for “Hybrid Hybrid GRACE Hash Join”

Current implementation of hash join in Hive is the Classic Hash Join, which can only handle the case when the small table can be entirely fit in memory. Otherwise hash join cannot be performed. This feature is trying to lift that limitation by making hash join more general, so that even if the size of small table is larger than memory, we can still do the hash join, in an elegant way.

It’s obvious that GRACE Hash Join uses main memory as a staging area during the partitioning phase, which unconditionally scans both relations from disk, then partitions and puts them back to disk (Hybrid GRACE Hash Join puts one pair less), and finally reads them back to find matches.

This feature tries to avoid the unnecessary write-back of partitions to disk as much as possible, and will only do that when necessary. The idea is to fully utilize the main memory to hold existing partitions of hash tables.

The key factor that will impact the performance of this algorithm is whether the data can be evenly distributed into different hash partitions. If we have skewed values, which will result in a few very big partitions, then an extra partitioning step is needed to divide the big partitions down to many. This can happen recursively. Refer to Recursive Hashing and Spilling below for more details.

Algorithm

Like other hash joins, there are a build phase and a probe phase. But there are several new terms to be defined.

- *Partition*. Instead of putting all keys of the small table into a single hash table, they will be put into many hash tables, based on their hash value. Those hash tables are known as partitions. Partitions can be either in memory when initially created, or spilled to disk when the memory is used up.
- *Sidefile*. It is a row container on disk to hold rows from the small table targeted to a specific partition that has been already spilled to disk. There can be zero or one sidefile for each partition spilled to disk.
- *Matchfile*. It is a row container on disk to hold rows from the big table that have possible matching keys with partitions spilled to disk. It also has a one-to-one relationship to the partitions.

When the entire small table can fit in memory, everything is similar to Classic Hash Join, except there are multiple hash tables instead of one in memory.

When the small table cannot fit in memory, things are a little more complicated.

Small table keys keep getting hashed into different partitions until at some point the memory limit is reached. Then the biggest partition in memory will be moved to disk so that memory is freed to some extent. The new key that should have been put into that partition will be put in the corresponding sidefile.

During probing, if matches are found from in-memory partitions, they will be directly put into the result.

If a possible match is found, i.e., the key hash of big table falls into an on-disk partition, it will be put into an on-disk matchfile. No matching is done at this moment, but will be done later.

After the big table has been exhausted, all the in-memory partitions are no longer needed and purged. Purging is appropriate because those keys that should match have been output to result, and those that don't match won't be relevant to the on-disk structures.

Now there may be many on-disk triplets (partition, sidefile and matchfile). For each triplet, merge the partition and sidefile and put them back in memory to form a new hash table. Then scan the matchfile against the newly created hash table looking for matches. The logic here is similar to Classic Hash Join.

Repeat until all on-disk triplets have been exhausted. At this moment, the join between small table and big table is completed.

Recursive Hashing and Spilling

There are cases when the hash function is not working well for distributing values evenly among the hash partitions, or the values themselves are skewed, which will result in very big sidefiles. At the time when the on-disk partition and sidefile are to be merged back into memory, the size of the newly combined hash partition will exceed the memory limit. In that case, another round of hashing and spilling is necessary. The process is illustrated below.

Assume there's only 1GB memory space available for the join. Hash function h1 distributes keys into two partitions HT1 and HT2. At some point in time, the memory is full and HT1 is moved to disk. Assume all the future keys all hash to HT1, so they go to Sidefile1.

As normal, during the probe phase big table values will be either matched and put into result or not matched and put into a Matchfile.

After the big table is exhausted, all the values from big table should be either output (matched) to the result, or staged into matchfiles. Then it is time to merge back pairs of on-disk hash partition and sidefile, and build a new in-memory hash table, and probe the corresponding matchfile against it.

In this example, since the predicted size of the merge is greater than the memory limit (800MB + 300MB > 1GB), we cannot simply merge them back to memory. Instead, we need to rehash them by using a different hash function h2.

Now we probe using Matchfile 1 against HT 3 (in memory) and HT 4 (on disk). Matching values for HT 3 go into result. Possibly matching values for HT 4 go to Matchfile 4.

This process can continue recursively if the size of HT 4 plus size of Sidefile 4 is still greater than the memory limit, i.e., hashing HT 4 and Sidefile 4 using a third hash function h3, and probing Matchfile 4 using h3. In this example, the size of HT 4 plus size of Sidefile 4 is smaller than memory limit, so we are done.

Skewed Data Distribution

Sometimes it happens that all or most rows in one hash partition share the same value. In that case, no matter how we change the hash function, it won't help anyway.

Several approaches can be considered to handle this problem. If we have reliable statistics about the data, like detailed histograms, we can process rows with the same value using a new join algorithm (using or dropping rows sharing the same value all in one shot). Or we can divide the rows into pieces such that each piece can fit into memory, and perform the probing in several passes. This probably will bring performance impact. Further, we can resort to regular shuffle join as a fallback option once we figure out Mapjoin cannot handle this situation.

Bloom Filter

As of Hive 2.0.0, a cheap Bloom filter is built during the build phase of the Hybrid hashtable, which is consulted against before spilling a row into the matchfile. The goal is to minimize the number of records which end up being spilled to disk, which may not have any matches in the spilled hashtables. The optimization also benefits left outer joins since the row which entered the hybrid join can be immediately generated as output with appropriate nulls indicating a lack of match, while without the filter it would have to be serialized onto disk only to be reloaded without a match at the end of the probe.

References

- Hybrid Hybrid Grace Hash Join presentation by Mostafa
- MapJoinOptimization https://cwiki.apache.org/confluence/display/Hive/MapJoinOptimization
- HIVE-1641 add map joined table to distributed cache
- HIVE-1642 Convert join queries to map-join based on size of table/row
- Database Management Systems, 3rd ed
- Kitsuregawa, M. Application of Hash to Data Base Machine and Its Architecture
- Shapiro, L. D. Join Processing in Database Systems with Large Main Memories
- Dewitt, David J. Implementation techniques for main memory database systems
- Jimmy Lin and Chris Dyer Data-Intensive Text Processing with MapReduce