

# MapJoinOptimization

## Index

- 1. Map Join Optimization
  - 1.1 Using Distributed Cache to Propagate Hashtable File
  - 1.2 Removing JDBM
  - 1.3 Performance Evaluation
- 2. Converting Join into Map Join Automatically
  - 2.1 New Join Execution Flow
  - 2.2 Resolving the Join Operation at Run Time
  - 2.3 Backup Task
  - 2.4 Performance Evaluation

## 1. Map Join Optimization

### 1.1 Using Distributed Cache to Propagate Hashtable File

Previously, when 2 large data tables need to do a join, there will be 2 different Mappers to sort these tables based on the join key and emit an intermediate file, and the Reducer will take the intermediate file as input file and do the real join work. This join mechanism (referred to as Common Join) is perfect with two large data size. But if one of the join tables is small enough to fit into the Mapper, A's memory, then there is no need to launch the Reducer. Actually, the Reducer stage is very expensive for the performance because the Map/Reduce framework needs to sort and merge the intermediate files. So the basic idea of Map Join is to hold the data of small table in Mapper, A's memory and do the join work in Map stage, which saves the Reduce stage. The previous implementation (before optimization) is shown here:

? Unknown Attachment

Fig 1. The Previous Map Join Implementation

In Fig 1 above, the previous map join implementation does not scale well when the larger table is huge because each Mapper will directly read the small table data from HDFS. If the larger table is huge, there will be thousands of Mapper launched to read different records of the larger table. And those thousands of Mappers will read this small table data from HDFS into their memory, which can make access to the small table become the performance bottleneck; or, sometimes Mappers will get lots of time-outs for reading this small file, which may cause the task to fail.

(HIVE-1641) has solved this problem, as shown in Fig2 below.

? Unknown Attachment

Fig 2. The Optimized Map Join

The basic idea is to create a new task, MapReduce Local Task, before the original Join Map/Reduce Task. This new task will read the small table data from HDFS to in-memory hashtable. After reading, it will serialize the in-memory hashtable into files on disk and compress the hashtable file into a tar file. In next stage, when the MapReduce task is launching, it will put this tar file to Hadoop Distributed Cache, which will populate the tar file to each Mapper, A's local disk and decompress the file. So all the Mappers can deserialize the hashtable file back into memory and do the join work as before.

Obviously, the Local Task is a very memory intensive. So the query processor will launch this task in a child jvm, which has the same heap size as the Mapper's. Since the Local Task may run out of memory, the query processor will measure the memory usage of the local task very carefully. Once the memory usage of the Local Task is higher than a threshold number. This Local Task will abort itself and tells the user that this table is too large to hold in the memory. User can change this threshold by `set hive.mapjoin.localtask.max.memory.usage = 0.999;`

### 1.2 Removing JDBM

Previously, Hive uses JDBM (HIVE-1293) as a persistent hashtable. Whenever the in-memory hashtable cannot hold data any more, it will swap the key /value into the JDBM table. However when profiling the Map Join, we found out this JDBM component takes more than 70 % CPU time as shown in Fig3. Also the persistent file JDBM generated is too large to put into the Distributed Cache. For example, if users put 67,000 simple integer key/value pairs into the JDBM, it will generate more 22M hashtable file. So the JDBM is too heavy weight for Map Join and it would better to remove this component from Hive. Map Join is designed for holding the small table's data into memory. If the table is too large to hold, just run as a Common Join. There is no need to use a persistent hashtable any more. (HIVE-1754)

? Unknown Attachment

Fig 3. The Profiling Result of JDBM

### 1.3 Performance Evaluation

Here are some performance comparison results between the previous Map Join with the optimized Map Join

Table 1: The Comparison between the previous map join with the new optimized map join

? Unknown Attachment

As shown in Table1, the optimized Map Join will be 12 ~ 26 times faster than the previous one. Most of map join performance improvement comes from removing the JDBM component.

## 2. Converting Join into Map Join Automatically

### 2.1 New Join Execution Flow

Since map join is faster than the common join, it would be better to run the map join whenever possible. Previously, Hive users need to give a hint in the query to assign which table the small table is. For example, `select /*mapjoin(a)/* from src1 x join src2y on x.key=y.key`; It is not a good way for user experience and query performance, because sometimes user may give a wrong hint and also users may not give any hints. It would be much better to convert the Common Join into Map Join without users' hint.

(HIVE-1642) has solved the problem by converting the Common Join into Map Join automatically. For the Map Join, the query processor should know which input table the big table is. The other input tables will be recognize as the small tables during the execution stage and these tables need to be held in the memory. However, in general, the query processor has no idea of input file size during compiling time (even with statistics) because some of the table may be intermediate tables generated from sub queries. So the query processor can only figure out the input file size during the execution time.

Right now, users need to enable this feature by `set hive.auto.convert.join = true`;  
This would become default in hive 0.11 with (HIVE-3297)

? Unknown Attachment

Fig 5: The Join Execution Flow

As shown in fig5, the left side shows the previous Common Join execution flow, which is very straightforward. On the other side, the right side is the new Common Join execution flow. During the compile time, the query processor will generate a Conditional Task, which contains a list of tasks and one of these tasks will be resolved to run during the execution time. It means the tasks in the Conditional Task's list are the candidates and one of them will be chosen to run during the run time. First, the original Common Join Task should be put into the list. Also the query processor will generate a series of Map Join Task by assuming each of the input tables may be the big table. For example, `select * from src1 x join src2y on x.key=y.key`. Both table `*_src2_*` and `*_src1_*` may be the big table, so it will generate 2 Map Join Task. One is assuming src1 is the big table and the other is assuming src2 is the big table, as shown in Fig 6.

? Unknown Attachment

Fig 6 Create Map Join Task by Assuming One of the Input Table is the Big Table

### 2.2 Resolving the Join Operation at Run Time

During the execution stage, the Conditional Task can know exactly the file size of each input table, even the table is a intermediate table. If all the tables are too large to be converted into map join, then just run the Common Join Task as previously. If one of the tables is large and others are small enough to run Map Join, then the Conditional Task will pick the corresponding Map Join Local Task to run. By this mechanism, it can convert the Common Join into Map Join automatically and dynamically.

Currently, if the total size of small tables are large than 25M, then the Conditional Task will choose the original Common Join run. 25M is a very conservative number and user can change this number by `set hive.mapjoin.smalltable.filesize = 30000000`.

### 2.3 Backup Task

As mentioned above, the Local Task of Map Join is a very memory intensive. So the query processor will launch this task in a child jvm, which has the same heap size as the Mapper's. Since the Local Task may run out of memory, the query processor will measure the memory usage of the local task very carefully. Once the memory usage of the Local Task is higher than a threshold. This Local Task will abort itself and it means the map join task fails. In this case, the query processor will launch the original Common Join task as a Backup Task to run, which is totally transparent to user. The basic idea is shown as Fig 7.


? Unknown Attachment

Fig7, Run the Original Common Join as a Backup Task

### 2.4 Performance Evaluation

Here are some performance comparison results between the previous Common Join with the optimized Common Join. All the benchmark queries here can be converted into Map Join.

**Table 2: The Comparison between the previous join with the new optimized join**

•  Unknown Attachment \*

For the previous common join, the experiment only calculates the average time of map reduce task execution time. Because job finish time will include the job scheduling overhead. Sometimes it will wait for some time to start to run the job in the cluster. Also for the new optimized common join, the experiment only adds up the average time of local task execution time with the average time of map reduce execution time. So both of the results have avoided the job scheduling overhead.

From the result, if the new common join can be converted into map join, it will get 57% ~163 % performance improvement.