Efficient Join in Hadoop

Gang & Ronnie

Abstract

Through our course project, we have implemented new types of join in the Hadoop Map/Reduce framework, which was designed to handle large amount of data in parallel. The existing join strategies in Hadoop or its data warehousing tool Hive may not be efficient if the join will drop most of the records in table. Using the idea of a pre-processing step before doing the join operation, our new join operation could perform better than traditional "default" join. Meanwhile, the system should be clever enough to decide whichever method to use, rather than obeying directions from the user.

1. Introduction

Currently, the amount of data the industries and academia facing are increasingly large, and will go on increasing, which makes it a hot issue for large-scale data processing. Map-Reduce is a popular parallel data processing framework. Its simple programming model allows users to write simple program that could run on hundreds of machines simultaneously to process data. Its fault tolerance feature also makes it a robust system even for commodity machines. These features enable the system running Map-Reduce expand to a really large scale by adding commodity machines to the cluster at a low cost, thus could greatly reduce the time consume by jobs.

Even though it seems promising to improve the efficiency for data processing by brutally enlarging the cluster size and running the jobs on more nodes, it is a better idea to design sophisticate plan that make good use the Map-Reduce paradigm while avoid the side effect as much as possible. As one of the most critical operations in data processing, join operation is usually more time-consuming than other kinds of work and thus has a greater impact on the overall performance. Basically, to join two datasets in Map-Reduce is quite simple, as we will introduce later. But the default join method has a limitation on the size of the datasets. When the datasets are too large, the I/O operations will dominate the time, thus lower the utilization of the computing resources.

Our work will focus on supply more sophisticated join methods that outperform the default join for certain situation. More importantly, we propose a cost model for Map-Reduce for the efficiency of join, and design a dynamic scheduler that could determine the base join methods to use, based on the cost model we propose, for certain situation.

The remainder of this report will be organized as follows. Sections 2 will briefly introduce the ideas of join in Map-Reduce and the deficiencies. We propose out new ideas to improve the join efficiency in section 3, as well as a simple cost model to evaluate the different join methods. A demo system will be introduced in section 4, followed by the performance evaluation in section 5. Finally, we will conclude our work and discuss the future work.

2. Default Joins

In Map-Reduce framework, two join methods are widely used, namely default join and map join. In this section, we will briefly introduce the implementations of the two join operations in Map-Reduce. For each of these implementations, we will discuss the problems they meet and the ideas how to improve it.

2.1. Default Join

Default join is a 2-way join widely used in Map-Reduce which comply with the MapReduce spirit fairly well.

Given two tables S and R, default join will first read different parts of these two tables into different Mappers where for each record, the join attribute will be extracted as the key of a record in intermediate result, with the file tag (to indicate which table does one record come from) and other necessary attributes as the value. The intermediate will be shuffled and then send to Reducers. Each reducer will only receiver those intermediate results with the same key. Records in each reducer are grouped by tag so that the records from different tables are identical. The Cartesian product of these two parts are the final result.

Default join work well for most situations. The exception is that when both R and S are huge, there are lots of data transferred over network from Mappers to Reducers. Without considering early projection or any selection based on predicates given by user, the size of data transferred on network is the sum of size of R and S. Now that the network transfer is the bottleneck in this case, one potential solution is to first filter the source datasets and get rid of those records at Mapper phase which are not possible to be joined in the Reducer phase.
2.2. Map side join

Map side join is one of the improvement for default join by eliminating the reduce phase and thus get rid of the transfer of data over network between mapper and reducer. This is extremely efficient when one table is really small.

Map side join aims to use only the Mapper phase so the no data will be transferred on network. It first reads one table (say R) by MapReduce framework into different Mappers. For each Mapper, the other table (say S) will be completely read into that Mapper and the join will be executed between part of R and the whole S. For different Mappers hold different fraction of R, the sum of final result is the join result between R and S.

For map side join has to read the whole table S into every Mapper, it is really efficient when it meets a huge table R and a small table S, but disastrous when both of R and S are huge, for the manually read using Hadoop from HDFS doesn't use the distributed feature of MapReduce and thus cause reading more duplicate data if some nodes fail, which is more likely than before.

The idea to improve map side join is similar to what we propose for default join. First filter and get rid of any record that could not be joined later and make at least one (in some cases, filtering both table is better) table much smaller and then do the map side join again.

Even though the problems of default join and map side join are different, the improvement is similar for both of them. That is to find out all the keys that will definitely join later and use it to filter the input source, instead of blindly transferring all data over network regardless whether they are useful or not. Of course, there is a trade off between the benefit we get by doing this and the cost we have to pay for the extra work. In the following sections, we will discuss new join methods and the way to evaluate the benefit and cost in order to make a better choice of these joins.

3. Advanced Joins

Based on the deficiencies and the potential ideas for improvements mentioned in section 2, we propose two different join methods for those situations that the default join and map side join are not suitable for. We will first introduce the prerequisite operation, semi join, and the ideas of these two joins, namely advanced default join and advanced map join. We will give out a cost model after that based on which we design a schedule to dynamically choose the best join method for a given context. Finally, a demo system which comprise the new joins methods and the scheduler will be introduced.

3.1. Semi Join

The advanced joins heavily rely on the result of Semi join which aims to find out all the distinct values used as the join key that are shared by both tables. The process of semi join is described as follow.

First, the two tables are read into the different Mappers, where the join key value for each record will be extracted as the key of intermediate result with the tag (to indicate which table does one record come from) as the value. After shuffling, all the records from both tables with the same join key value will be sent to the same Reducer. Each Reducer scan its input and try to find different tags. If it find different tags, which means the both tables contains records with the same join key value, then the Reducer output the key. If all the tags are the same after scanning all the input for certain Reducer, which means only one table contains records with this join key value, the Reducer does nothing for this key will not be used for join later. After all the Reducers finishing their work, we get all the distinct join key values. This is useful for filtering out those useless records for this join.

Semi-join makes sense even if the join key is Primary and Foreign key. Consider the following scenario: if the join key is a Primary key in one table, and in the other table it corresponds to the first table, then we cannot gain much performance through semi-join, since it is highly probable that most tuples will join with the other table, and semi-join will not purge a significant number of tuples. Even if there is selection over the key, we should always apply the selection in the mapping phase. It seems semi-join is not applicable to this scenario. However, if we consider the selection on attributes that are not the key of the join, then after the selection, we could purge "useless" tuples in the other table using semi-join. If we compare this case with transferring the whole second table over the network, semi-join will benefit.

3.2. Advanced Default Join

Semi-join will not be utilized as a separate way of joining two tables. (Who wants a list of matching keys in two tables only?) Instead, it can be used by the default join so that the communication cost will be reduced.

It is a combination of semi-join and default join: first the two tables are read from local storage and mapped to key-value pairs containing only the join key (join key as "key", table name might be "value"). Then these pairs are fed to reducers who will join them and produce a list of matching keys.

This list is then applied to filter the original tables using a map-side join, which will purge all tuples with a key that will not join. Then we can apply the default join in Hive to join the two new tables. This process can be completed in two Map-Reduce jobs instead of three, the
first one to do the semi-join, the second job will use mapping phase to filter and reduce phase to join.

The detailed process are described as following. In Map-Reduce job 1, two tables are read from HDFS to Mappers, and the projection on the join key feeds the result toReducers, and the result will be a list of join keys that are actually will produce the join result. In job 2, given the input datasets, each mapper manually read the filter file acquired from job 1 and then apply the filter to the tables. Then the tuples are fed to Reducers for join just like what happen in default join. Given that we also implemented the selection and projection, all these operation, if there is any, will happen in the map phase of job 2.

Advanced default join needs an extra job for semi join first, even though the time to run it is not that huge as we will mention in section 5. Also, the filtering work in the job 2 is an extra work compared with the default join. We will discuss in what condition advanced default join will outperform default join later.

3.3. Advanced Map-side Join

The semi join will rule out those "useless" tuples. If we get one of the tables fit enough to be considered map-side join. We can just apply the semi-join to both tables (we need to have an argument why semi-join should be applied to both tables, since it seems filtering the side (smaller) table will be enough).

As mentioned earlier, the advanced map-side join is basically the combination of a semi-join and a map-side join. Three Map-Reduce jobs are needed to complete the process (two of them only has mapping phase). In Map-Reduce job 1, we do the semi join based on two input datasets, exactly the same as we do in the job 1 of advanced default join. In job 2, we use the filter file from semi join to filter the big tables. This is actually a map side join for we 'join' the tuples in one of the dataset and the filter file so that the two tables only contains matching tuples after filtration. This job thus contains on map phase.

Even though the advanced map side join needs 3 jobs, we note that the last two jobs only needs map phase, which avoid the reduce phase and the data transfer from mapper to reducer (instead, it pays for the cost to write the result of each job to HDFS).

3.4. Cost Model

Before we move on to talk about the scheduler in detail, it is necessary to indicate the cost model base on which we evaluate the performance and choose the best join plan dynamically.

Generally, we divide the cost into four categories for any MapReduce job. They are:

1. Local I/O, including Local Read and Local write (e.g. Mapper write the intermediate result).
2. Local computation, which is the actual execution within Mappers and Reducers.
3. Communication over network, which needs the Reducers to read the data stored on disks of Mapper nodes.
4. Result output, which write the final result to HDFS.

For any join has to write the final result and the result is always the same for the same tables and the same join keys. So, we don't need to compare the cost to output the final result. Compared with I/O, the cost for local computation within each node is small, thus could be neglected. So, our cost model focus on the local I/O and communication cost. With the communication cost the biggest bottleneck, we give it more weight when analyzing the cost. Most of the time, the input to mapper also cause the remote data transfer for not every node that runs the map task contains corresponding data splits. Because we can use the Distributed Cache, one functionality that could pre-load the data to corresponding nodes before map task starts. So, we just assume that the input to Mapper is local I/O for simplicity.

The cost of the semi join contain reading the two whole tables from local disks of each Mapper (according to MapReduce framework, a node will assigned appointed as a Mapper when it contains part of the input files, thus it read its data locally). The output of intermediate is trivial which could be neglected. The reason is that each records in the intermediate result only contain key value and the tag information. Normally, the key used for join are really small (e.g. id, which is an integer type taking up 4 bytes) and the tag only need one bit to indicate which of the two tables does this record come from. So, even we got millions of records for the intermediate result, the size of it is several mega-bytes. The output size of semi join is $pA(R) + pA(S)$, which is even smaller and thus could be neglected. So, the total cost for semi join is local I/O for $\text{size}(R) + \text{size}(S)$ data.

The analysis of advanced default join is as following.

1. \text{MapReduce Job 1 (semi-join)}
   - Local read: $\text{size}(R) + \text{size}(S)$
   - Data transfer cost: $\text{size}(A(R) + A(S))$ - comparatively small
   - Local write: $\text{size}(A(R) \cap A(S))$ - comparatively small

2. \text{MapReduce Job 2}
   - Local read: $\text{size}(R) + \text{size}(S)$
   - Global read (for each map): $\text{size}(A(R) \cap A(S))$ - comparatively small
   - Data transfer cost: $\text{size}(R') + \text{size}(S')$

The advanced default join will perform extra local I/O, but the communication cost is reduced from $\text{size}(R) + \text{size}(S)$ to $\text{size}(R') + \text{size}(S')$, which could mean great difference if the join will cut most of the tables. Meanwhile, the advanced default join could be applied to
joining any two tables, replacing the default join in Map-Reduce.

The analysis of the advanced map join is as follows.

1. **MapReduce Job 1 (semi-join)**
   
   Local read: \( \text{size}(R) + \text{size}(S) \)
   
   Data transfer cost: \( \text{size}(A(R) + A(S)) \) - comparatively small

   Local write: \( \text{size}(A(R) \cap A(S)) \) - comparatively small

2. **MapReduce Job 2 (mapping only)**
   
   Local read: \( \text{size}(R) + \text{size}(S) \)
   
   Local write: \( \text{size}(R') + \text{size}(S') \)

3. **MapReduce Job 3 (mapping only)**
   
   Local read: \( \text{size}(R') \)
   
   Global read (data transfer): \( n \cdot \text{size}(S') \)

The "traditional" map-side join will transfer \( n \cdot \text{size}(R) \) amount of data over the network.

According to our cost model, if \( n' \cdot \text{size}(S') \ll n \cdot \text{size}(S) \), this type of join will benefit.

Note that we did not consider the output of each of the intermediate Map-Reduce jobs, which write the result of a job to HDFS. This needs replication in different nodes thus cause data transfer over network, which will bring extra cost that should be counted as part of the total cost. For this cost is comparatively difficult to trace and model, for simplicity, we ignore this cost. It needs further work to find out the impact of this cost for a more accurate cost model.

### 4. Implementation

#### 4.1. Overview

Our demo system will provide a CLI to receive commands to support several operations including join, set delimiter, filter and projection. The command line user input, with arbitrary combination of any number of these operations in any order, will be parsed by Parser to get the needed information for execution engine. Following is an example input supported by our demo system:

```
> set delimiter="|"
> join R.1 S.3 /output/ && filterby R.2 != "Google" && filterby S.3 = "database" && project R 1, 3, 4 && project S 2, 3, 4
```

Execution engine will first send this information to semi join and run the job. Based on the result of semi join, scheduler will analysis which join methods to choose next, and notify execution engine. Utils is a functions pool which supplies necessary functionalities for each join.

To accelerate the join, when we get the semi join result for two tables, we don't delete it even the join completes. If next time a join operation is applied on the same two tables with the same column as the join key, we will check whether the semi join result is usable by checking whether its modification data is newer than those of the two tables. If so, we use this semi join result for it is up to date, rather than executing semi join again and get the same result which we have already got.

#### 4.2. Advanced Default Join

The Advanced Default Join is based on two Map/Reduce jobs: the first one is Semi Join; the second one reads the result of the Semi Join, and filter each table to rule out any tuple that won't join, and then proceed to the reduce phase as a normal Default Join.

The implementation of Semi Join is extracting the join key at map phase, and only keeping keys that appear in both tables in the reduce phase. The filtration and projection work, if there is any, will be done by call corresponding functions in Utils in mapper.

#### 4.3. Advanced Default Join

The implementation of Advanced Map Join is very simple, it consists of four Map/Reduce jobs, three of which is identical to Map Join: the first phase is a Semi Join which outputs the filter table as a result; both second and third phase is a Map Join, using the filter as the small table and one of the original tables as the large table; the last phase will perform a normal Map Join on the result of the filtered tables. For we do the filtration at job 2 and we want to reduce the size of these two tables as much as possible, so we also apply the selection and projection at this job. Job 3 is where the real map side join happens, using the two filtered tables.

#### 4.4. Scheduler

Scheduler is one of the most important part of the demo system. Currently we have implemented a very simple Scheduler: when doing the semi join, we gather such statistics as number of input tuples, number of output tuples, compute the selectivity of this join and times it by the original size. Then, we could estimate size of the two original tables after being filtered. Therefore we could decide whether to use a Map Join: that is, whether one of the tables becomes small enough. Given that this 'threshold' value relies heavily on the cluster, which will vary greatly according to different environment and configurations, we set such threshold value to be 5 MB for simplicity.

### 5. Performance Evaluation
5.1. Methods

We developed our first versions of new types of joins based on some toy tables for the ease of coding: the first column is always the join key; each line of the file is a record; the columns are separated by a TAB character.

Undoubtedly the toy tables are small and simple: they seem too small for our “advanced” join methods. For example, the running time of all four types of join we are taking to consideration might be several seconds, which makes it hard to compare. A more important problem is that the time here is mostly initialization of work environment, the communication between the master node and a data node, the starting of a JVM, etc. Comparable to hard disk access, this time here is similar to a seek time instead of a transfer time which can determine the performance of a disk drive. Even if new join types perform better for very small tables, we cannot claim that they are better for large inputs.

Meanwhile, when the tables are small enough, the extra work we have done in the “advanced” methods is larger than the advantage we have gained. In other words, it is not worth the time of filtering the tables.

Thus, we need tables that are large enough to display the power of new types of joins. And we used TPC-DS data benchmark; although we do not use the queries provided with TPC-DS, the tables themselves provide a good simulation of real world data sets. However, in order to enable our codes to handle tables from TPC-DS, we have to adopt functions like: assigning the join key column, using the “|” character instead of TAB as the separator.

The codes were developed, deployed and tested on Gang’s Hadoop in pseudo-distributed mode on Linux, and Ronnie’s pre-configured Linux VM ware image with Hadoop from Yahoo! Hadoop tutorial. The testing was arranged to be on the CS Hadoop cluster, but as we have experienced some issues and problems, some of the results are from our personal workstations.

![Figure 1: the running time of three different join types, on four different join queries](image)
5.2. Results

Since the cluster in Computer Science Department for Hadoop is sometimes not stable, the data we have collected are not in the best accurate state; as we have observed, we might get different result by even running the same query using the same method of join, and the difference could be too vast to be normal. Therefore, we have discarded abnormal data and will try to find an explanation for any abnormal result.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Table 1 Size</th>
<th>Table 2</th>
<th>Table 2 Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join A</td>
<td>reason.dat.1</td>
<td>1339</td>
<td>store_returns.dat.9</td>
</tr>
<tr>
<td>Join B</td>
<td>reason.dat.1</td>
<td>1339</td>
<td>web_returns.dat.13</td>
</tr>
<tr>
<td>Join C</td>
<td>reason_partial.1</td>
<td>132</td>
<td>store_returns.dat.9</td>
</tr>
<tr>
<td>Join D</td>
<td>reason_partial.1</td>
<td>132</td>
<td>web_returns.dat.13</td>
</tr>
</tbody>
</table>

Table 1: join tables and sizes

These results are not performed on the CS Hadoop cluster. And from the table we can see that all four queries are joining a relatively small table to a large table. In a sense, we might consider table of 30 Megabytes as large on a single node. As can be observed from Figure 1, any kind of Map Join performs better than the Advanced Default Join. This is due to the natural advantage of Map Join, which can cut the reduce step and give better performance.

We may further notice that the simple Map Join performs way better than Advanced Map Join in each set. Because here we still have the same problem as with "toy data": the time spent on filtering will not match the time saved by filtering. If we run this test on larger tables, this problem will not be in the way. In one word, the results of the test have verified that through Map Join we can gain much better performance.

To observe the advantage of Advanced Map Join, we can refer to the Excel document with join results: customer_address.dat join with customer.dat using map join will take more than 4 minutes; however, if we filter one of the table by reducing the size to several KB, the running time is 33 seconds (line 12 of the table).

Join A is on the following tables: customer and customer_address; Join B is table web_sales and web_returns. The results are recorded from CS Hadoop cluster.

Figure 2 mainly compares between Default Join and Advanced Default Join, which turns out to be surprising, and reveals further problems with the CS Hadoop cluster.

Join A was on two tables of about 20 MB in total size, which cannot be considered large tables in a Hadoop cluster. And we have discussed that small tables will not unleash the power of "advanced" joins. However, the real reason for the Advanced Default Join to perform poor is
the selectivity: as we have observed, only less than twenty percent of the tuples in the original tables were dropped by the filtering step. As a result, by doing Semi Join and the filtering step, we might have done an amount of extra work, which is avoided by the "lucky" naive method.

Join B was on web_sales and web_returns. And naturally for these two tables, they should not have a high selectivity, since returns are only a small portion of sales (assuming this company runs a normal online store). And by observing the number of tuples in the tables we can verify this assumption. Meanwhile the two tables are of several hundred MB in size, which is also in our favor. However, the Advanced Default Join and Default Join turn out to be similar in running time.

After inspection the status message while running the two jobs, we learned that in Semi Join, the "Map Output" number does not match the "Reduce Input" number. In other words, the number of tuples in the output of Semi Join is not correct. Thus, it is hard to determine which method is better in such a way.

5.3. Discussion

For the Scheduler to give a good plan of executing a join query, it is important to establish a connection between the characteristics of a query and one of the join method. The Scheduler we have implemented is very simple-minded: determine whether to use Map Join according to the size of two tables. However, a better scheduler is possible with more convincing results.

And even if we use the advanced joins on smaller tables, it can still beneficial because the time we saved while joining large tables will be magnitudes more than that we have wasted on smaller tables. The focus of study should be only on even larger datasets than the ones we were using.

The following argument is still in the favor of "advanced" join methods. Semi join is very fast according to our study, because it extracts the join key column only and drops other unnecessary columns. In this way, even if we could have done extra work, it is worth the while for a chance to save much more time.

6. Future Work

As mentioned in the implementation stage, we have made several improvements to the way the "small table" in Map Join or "filter" in Advanced Default Join is read into memory. However, we still feel that this is a place where improvements can still be made in effort of faster Map Join, which is the key role in Advanced Map Join, as well as the Advanced Default Join. The file stored on HDFS can be read more efficiently rather than our current method of "read and concat string"; meanwhile, we are using an in-memory HashTable to store all records read from the file, and whether the HashTable in Java is efficient or we have a better way of matching the record with the table in memory remains a problem.

Moreover, we have not implemented the DistributedCache fashion of distributing the file to each node ahead of time. In the Hadoop cluster of CS, we have noticed that usually there are no more than 4 Mappers running in parallel, which makes this effort almost in vain. However, in a larger distributed cluster, this method of optimization is necessary for a file to be read by a lot of nodes.

As proposed in our mid-report, we have the "partition join" in mind, which basically uses the following idea: if the tables we are trying to join are stored partitioned by the join key in HDFS, we do not need the mapping phase to read and sort the table so that Reducers will get the tuples with same key. In this way, the time of sorting is saved.

The idea of "bit mapping" the result of Semi Join will give even better performance of Semi Join: given the range of values of join key, we use one bit to represent whether the value is in the join result. In this way, the time required to build the filter table, to transfer the filter, and to filter the original tables (we don't need to build the HashTable anymore since the bit map is already a hash function) will all be reduced.

One problem with our work is the Hadoop cluster in our department, as we have mentioned in the previous parts. The Scheduler will be more clever if we can maintain a more stable and convincing result from the cluster. And while implementing our ideas of "better scheduler", a more flexible schedule should also be adopted: after the Semi Join procedure, estimating the size of two tables after filtering separately and deciding whether to filter the corresponding original table will also bring more cleverness for the scheduler. To be more specific, we can get two copies of statistics of the input and output tuples for each of the input dataset, rather than get a overall selectivity. By doing this, we can see whether or not it deserves to filter the datasets (this depends on how high the selectivity is). If the answer is 'yes', we can further answer these question which one should be filtered or should we filter both of them. Besides, for the scheduler basically relies on the cost model we propose earlier, the accuracy of the data model will influence the efficiency of scheduler. How to make a more accurate model is a critical problem.

7. Acknowledgements

We would like to express our gratitude to Professor Shivnath Babu, for giving us numerous new ideas.
8. Reference


