Cost-Based Join Algorithm Selection in Hadoop

Jun Gu[1], Shu Peng[1], X. Sean Wang[1], Weixiong Rao[2], Min Yang[1], and Yu Cao[3]

[1] School of Computer Science, Fudan University, Shanghai, China
Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China
{gujun,pengshu,xywangCS,m_yang}@fudan.edu.cn
[2] School of Software Engineering, Tongji University, Shanghai, China
wxrao@tongji.edu.cn
[3] EMC Labs, Tsinghua Science Park, Beijing, China
yu.cao@emc.com

Abstract. In recent years, MapReduce has become a popular computing framework for big data analysis. Join is a major query type for data analysis and various algorithms have been designed to process join queries on top of Hadoop. Since the efficiency of different algorithms differs on the join tasks on hand, to achieve a good performance, users need to select an appropriate algorithm and use the algorithm with a proper configuration, which is rather difficult for many end users. This paper proposes a cost model to estimate the cost of four popular join algorithms. Based on the cost model, the system may automatically choose the join algorithm with the least cost, and then give the reasonable configuration values for the chosen algorithm. Experimental results with the TPC-H benchmark verify that the proposed method can correctly choose the best join algorithm, and the chosen algorithm can achieve a speedup of around 1.25 times over the default join algorithm.

Keywords: Join algorithm, Cost model, Hadoop, Hive.

1 Introduction

In recent years, MapReduce [1] has become a popular computing framework for big data analysis. Hadoop [2], an open-source implementation of MapReduce, has been widely used. One example of the big data analysis is log processing, such as the analysis of click-streams, application access logs, and phone call records. Log analysis often requires a join operation between log data and reference data (such as information about users). Unfortunately, MapReduce was originally designed for the processing of a single input, and the join operation typically requires two or more inputs. Consequently, it has been an open issue to improve Hadoop for the join operation.

Many works have appeared in the literature that tackle the join operation in Hadoop. Such works roughly fall into the two categories. The first is to design novel join algorithms on top of Hadoop [6][7][8][9][10][11]. The second is to change the internals of Hadoop or build a new layer on top of Hadoop for the optimization of traditional join algorithms [3][12][14][15][16][18][19].

B. Benatallah et al. (Eds.): WISE 2014, Part II, LNCS 8787, pp. 246–261, 2014.
© Springer International Publishing Switzerland 2014

Given the many existing join algorithms, it is hard for users to choose the best one for their particular join tasks since different algorithms differ significantly in their performance when used on different tasks. Usually Hadoop users have to define a `map` function and a `reduce` function and to configure their own MapReduce jobs. It's a much harder task for the users to change the internals of Hadoop or build a new layer on top of Hadoop. Hive system [3] is probably the most popular open source implementation to execute join queries on top of Hadoop. However, even with the help of Hive, it is still hard for a Hive user to choose the best join algorithm among all those implemented in Hive. Finally, suppose that a join algorithm is chosen, the users are required to tune some key parameters. Unfortunately, choosing the best join algorithm and tuning the parameters involve non-trivial efforts.

To help the end users select the best join algorithm and tune the associated parameters, in this paper, we propose a general cost model for four widely used join algorithms. Based on our cost model, we adaptively choose one of the join algorithms with the least cost, and then set the reasonable configuration values.

In summary, we make the following contributions:

- First, we design a cost model for the four popular join algorithms in Hive, and propose a tree structure based on which a pruning method is designed for the automated selection of the best join algorithm.
- Second, as a part of the selection process, we provide a method to tune the key parameters needed by the chosen join algorithm.
- Third, based on the TPC-H benchmark [4], we conduct experiments to evaluate our proposed method. The experimental results verify that our method can correctly choose the best join algorithm. Moreover, the chosen algorithm can achieve a speedup of around 1.25, compared with the default join algorithm.

The rest of the paper is organized as follows. In Section 2, we introduce the background of our work. We present the cost model in Section 3, and in Section 4 design an algorithm, along with a tree-structured pruning strategy, to choose the best join algorithm. We report an evaluation of our method in Section 5. In Section 6, we investigate related works, and finally conclude the paper with Section 7.

2 Background

In this section, we first introduce the two popular open source systems: Hadoop and Hive (Section 2.1), and review four join algorithms that are implemented in Hive (Section 2.2).

2.1 Hadoop and Hive

Hadoop, an open-source implementation of MapReduce, has been widely used for big data analysis. The Hadoop system mainly contains two components:

Hadoop Distributed File System (HDFS) [5] and MapReduce computing framework. Hadoop reads input data from HDFS and writes output data to HDFS. The input and output data are maintained on HDFS with data blocks of 64 MB by default. To process a query job, Hadoop starts `map` tasks (mappers) and `reduce` tasks (reducers) concurrently on clustered machines. Each mapper reads a chunk of input data, extracts <key, value> pairs, applies the `map` function and emits intermediate <key', value'> pairs. Those intermediate pairs with the same key are grouped together as <key', list<value'>>. After that, the grouped pairs are then shuffled to reducers. Each reducer, after receiving the grouped pairs, applies the `reduce` function onto the grouped pairs, and finally writes the outputs back to HDFS.

Hive is a popular open-source data warehousing solution, which facilitates querying and managing large datasets on top of Hadoop. Hive supports queries expressed in an SQL-like declarative language, called HiveQL. Using HiveQL lets users create summarizations of data, perform ad-hoc queries, and analysis of large datasets in the Hadoop cluster. For those users familiar with the traditional SQL language, they can easily use HiveQL to execute SQL queries. HiveQL also allows programmers who are familiar with MapReduce to plug-in their custom mappers and reducers. In this way, Hive can perform complex analysis that may not be supported by the built-in capabilities of the HiveQL language. Based on the queries written in HiveQL, Hive compiles the queries into MapReduce jobs, and submits them to Hadoop for execution. Since Hive can directly use the data in HDFS, operations can be scaled across all the datanodes and Hive can manipulate huge datasets.

2.2 Join Algorithms in Hadoop

In this section, we review four join algorithms that are widely used in MapReduce framework. All of such algorithms are supported in the Hive system. We will design our selection method based on these four join algorithms.

- *Common Join*: We consider that two tables are involved in a join task. In the `map` function, each row of the two join tables is tagged to identify the table that the row comes from. Next, the rows with the same join keys are shuffled to the same reducer. After that, each reducer joins the rows from the two join tables on the key-equality basis. *Common Join* can always work correctly with any combinations of sizes of the join tables. However, this join algorithm may incur the worst performance efficiency due to a large amount of shuffled data across clustered machines.
- *Map/Broadcast Join*: For two join tables, this algorithm first starts a local MapReduce task to build a hashtable of the smaller table. The task next uploads the hashtable to HDFS and finally broadcasts the hashtable to every node in the cluster (Note that the hashtable is maintained on the local disk of each node in form of a distributed cache). After finishing the local MapReduce task, this algorithm starts a `map`-only job to process the join query as follows. First, each mapper reads the hashtable (i.e., the smaller

table) from its local disk into main memory. Second, the mapper scans the large table and matches record keys against the hashtable. By combining the matches between the two tables, the mapper finally writes the output onto HDFS. This algorithm does not start any reducer, but requires that the hashtable of the smaller table is small enough to fit into local memory.

- *Bucket Map Join*: Differing from *Map Join*, *Bucket Map Join* considers that the data size of join tables is big. Thus, in order to reduce the memory limitation of *Map Join* from keeping the whole hashtable of the smaller table, this algorithm bucketizes join tables into smaller buckets on the join column. When the number of buckets in one table is a multiple of the number of buckets in the other table, the buckets can be joined with each other. In this way, as *Map Join*, a local MapReduce task is launched to build the hashtable of each bucket of the smaller table, and then broadcast those hashtables to every nodes in the cluster. Now, instead of reading the entire hashtable of smaller table, mappers only read the required hashtable buckets from distributed cache into memory. Thus, *Bucket Map Join* reduces the used memory space.
- *Sort-Merge-Bucket* (*SMB*) *Join*: If data to be joined is already sorted and bucketized on the join column with the exactly same number of buckets, the creation of hashtable is unneeded. Each mapper then reads records from the corresponding buckets from HDFS and then merges the sorted buckets. The *SMB* algorithm allows the query processing to be faster than an ordinary `map`-only join. The *SMB Join* is thus fast for the tables of any size with no limitation of memory though with the requirement that the data should be sorted and bucketized before the query processing.

3 Cost Model

Given the four popular join algorithms, in this section, we design a cost model to estimate the cost of the four algorithms that are used to process a join query. Here, we only consider the fundamental join SQL query without *WHERE* conditions (and other sub-queries such as *GROUP BY*, etc.):

```
SELECT C FROM T1 JOIN T2 ON T1.ci = T2.cj;
```

where C denotes the projected columns from join tables T_1 and T_2, and c_i and c_j are the query join keys.

Before presenting our cost model, we first make the following assumptions:

- Firstly, we assume that T_2 is the smaller table. Thus, when the *Map/Broadcast Join* or *Bucket Map Join* algorithm is used, the cost of broadcasting and building the hashtable involves only the table T_2.
- Secondly, for *Bucket Map Join* and *SMB Join*, we assume that the big table T_1 and small table T_2 are associated with the same number of buckets, i.e., $N_1 = N_2$. As described in Section 2, *SMB Join* does require $N_1 = N_2$. Instead for *Bucket Map Join*, it requires that $N_1 \% N_2 = 0$ or $N_2 \% N_1 = 0$, where % is the modulus operator. As a special case, the assumption $N_1 = N_2$ still makes sense for *Bucket Map Join*.

– Lastly, we don't distinguish the cost of building sorted buckets and unsorted buckets in our cost model. The assumption is reasonable because in the MapReduce framework, the sorted bucketizing job only takes a slightly more time than the unsorted one, which doesn't influence the correctness of our cost model.

With the above assumptions, we proceed to presenting our cost model. In this model, the cost of each join algorithm consists of the following five parts.

– *Bucketize cost* to bucketize both join tables, if any.
– *Broadcast cost* to build and broadcast hashtables to all mapper nodes, if any.
– *Map cost* for mappers to read input data from HDFS.
– *Shuffle cost* to shuffle mappers' output to reducers' nodes.
– *Join cost* to operate join.

Table 1 defines the symbols and cost functions we will use in our cost model.

Table 1. Cost Model Symbols

Symbol	**Description**
N_{nodes}	Number of nodes in the Hadoop cluster
B	Block size in HDFS
$N_{mappers}$	Number of mappers Hadoop sets up for the join query
$R(T)$	Number of rows of table T
$S(C(T))$	The total field size of query columns C of table T
$S(T)$	The size of input join table T
$S_c(T)$	The size of table T only with the query columns C
$N_{buckets}$	Number of buckets for join tables
$B_i(T)$	The ith bucket of table T
$S(B_i(T))$	The size of the ith bucket of table T
$R(B_i(T))$	Number of rows of the ith bucket of table T
$S_{Hashtable}(r)$	The size of hashtable of table T / bucket B with r rows
$N_{bucket-query}$	Number of queries with the same join key need bucketizing
Cost Function	**Description**
$T_{Hashtable}(r)$	Cost to build hashtable of table T / bucket B with r rows
$T_{Join}(r1, r2)$	Cost to join two tables or buckets with *r1* and *r2* rows
$T_{ReadHDFS}(m)$	Cost to read m GB data from HDFS
$T_{Transfer}(m)$	Cost to transfer m GB data through network
$T_{Bucketize}(t)$	Cost to bucketize join table t

We highlight the cost used by the four join strategies in terms of the five aforementioned parts:

1. *Common Join*
   Bucketize cost = 0
   Broadcast cost = 0

Map cost = $T_{ReadHDFS}(S(T1) + S(T2))$
Shuffle cost = $T_{Transfer}(S_c(T1) + S_c(T2))$
Join cost = $T_{Join}(R(T1), R(T2))$

2. *Map/Broadcast Join*
   Bucketize cost = 0
   Broadcast cost = $T_{ReadHDFS}(S(T2)) + T_{Hashtable}(R(T2)) + T_{Transfer}(S_{Hashtable}(R(T2))) * N_{nodes}$
   Map cost = $T_{ReadHDFS}(S(T1) + S_{Hashtable}(R(T2)) * N_{mappers})$
   Shuffle cost = 0
   Join cost = $T_{Join}(R(T1), R(T2))$
3. *Bucket Map Join*
   Bucketize cost = $(T_{Bucketize}(T1) + T_{Bucketize}(T2))/N_{bucket-query}$
   Broadcast cost = $T_{ReadHDFS}(S(T2)) + T_{Hashtable}(R(T2)) + T_{Transfer}(S_{Hashtable}(R(T2))) * N_{nodes}$
   Map cost = $T_{ReadHDFS}(S(B_i(T1)) + S_{Hashtable}(R(B_i(T_2)))) * N_{mappers}$
   $= T_{ReadHDFS}(S_c(T1) + S_{Hashtable}(T2))$
   Shuffle cost = 0
   Join cost = $T_{Join}(\bigcup R(B_i(T1)), \bigcup R(B_i(T2)))$
   $= T_{Join}(R(T1), R(T2))$
4. *SMB Join*
   Bucketize cost = $(T_{Bucketize}(T1) + T_{Bucketize}(T2))/N_{bucket-query}$
   Broadcast cost = 0
   Map cost = $T_{ReadHDFS}(S(B_i(T_1)) + S(B_i(T_2))) * N_{mappers}$
   $= T_{ReadHDFS}(S_c(T1) + S_c(T2))$
   Shuffle cost = 0
   Join cost = $T_{Join}(\bigcup R(B_i(T1)), \bigcup R(B_i(T2)))$
   $= T_{Join}(R(T1), R(T2))$

Before giving the details to compute the cost of each algorithm, we first look at the `map` tasks lunched by Hadoop:

$$N_{mappers} = \begin{cases} \frac{S(T1)}{B} & \textit{Common Join / Map Join}, \quad (1) \\ N_{buckets} & \textit{Bucket Map Join / SMB Join} \quad (2) \end{cases}$$

In case (1), for the *Common Join* and *Map Join*, the number of `map` tasks is determined by the number of splits of the big join table. By default, the split's size is equal to the HDFS block size. In case (2), for *Bucket Map Join* and *SMB Join*, because the tables are bucketized, the number of `map` tasks is determined by the number of buckets.

Now, we compute the cost of the four algorithms one by one. First for *Common Join*, in the `map` phase, mappers need to read the whole join tables' data $S(T)$ from HDFS. During the `map` function, unused columns will be filtered, and the records that are relevant to the join query are shuffled to reducers. Thus, the size of shuffle data is $S_c(T) = S(C(T)) * R(T)$. The cost of join is estimated by comparing the entire records $R(T1)$ and $R(T2)$ in two tables from begin to end.

Second for *Map Join*, it is required to build the hashtable of the smaller table $T2$, and the associated cost includes the one used to read $T2$ from HDFS, to build the hashtable, and finally to broadcast the hashtable to the number N_{nodes} of clustered nodes. After that, each mapper reads a split of $T1$ and the whole hashtable of $T2$. Hence, all mappers in total read the entire $T1$ and the number $N_{mappers}$ of times to load the hashtable of $T2$. The size of hashtable, $S_{Hashtable}(r)$, depends on the number of rows to build the hastable, and we compute $S_{Hashtable}(r) = \beta * r \;\; bytes$. In Hadoop, each row of the hashtable occupies around 1 byte, for simplicity, we set $\beta = 1$.

Next, for *Bucket Map Join*, we first need to bucketize both tables. We divide the total bucketizing cost by $N_{bucket-query}$, which means the bucketizing cost can be shared by $N_{bucket-query}$ queries and all these queries will benefit from the bucketizing job. Each bucket needs to be broadcasted to all other N_{nodes} nodes, so in total the broadcast cost is the same as *Map Join*. For map cost, $N_{mappers}$ mappers need to read buckets of $T1$ and corresponding hashtable of buckets of $T2$. Since we don't take *WHERE* conditions into account, all buckets of $T1$ and buckets' hashtables of $T2$ will be read. For the join cost, the $\bigcup R(B_i(T))$ represents the required buckets in T. Given the two tables T_1 and T_2, the buckets in T_1 are loaded to compare with the ones in T_2 for the join processing. All buckets will be read and compared with no *WHERE* clause in current model.

Finally, for *SMB Join*, its cost is different from *Bucket Map Join* in two parts: (i) It doesn't have to build and broadcast hashtables, and (ii) the mappers need to read all the buckets of $T1$ and $T2$, instead of hashtable of $T2$.

4 Cost-Based Selector

Based on the aforementioned cost model, in this section, we first design a tree structure (Section 4.1) and next propose a method to select one of the four join algorithms (i.e., *Common Join, Map Join, Bucket Map Join* and *SMB Join*) as the best algorithm to process a join query (Section 4.2).

4.1 Pruned Join Algorithm Candidates Tree

We design a *Join Algorithm Candidates Tree* (*JACTree*), shown in Fig. 1, as a pruning method for the automated selection of best join algorithm.

In this tree structure, the root means the bucket size of the smaller join table. We compare it with the size of the smaller table. If bucket_size is larger than the table_size (i.e., the left brunch), it means it's unnecessary to create any bucket. Otherwise, the tables need to be bucktized (i.e., the right brunch).

Now for the internal node with bucket_size >= table_size, we next need to consider the hashtable size of the bucket, and compare it with the available memory of the task nodes in the cluster. In case that the hashtable size is larger than the memory size, we then reach the leave node ① *Common Join*, and otherwise the leave node ② *Common Join* and *Map Join*. Similarly in the right branch, we can compare the hashtable size with the memory size and reach either

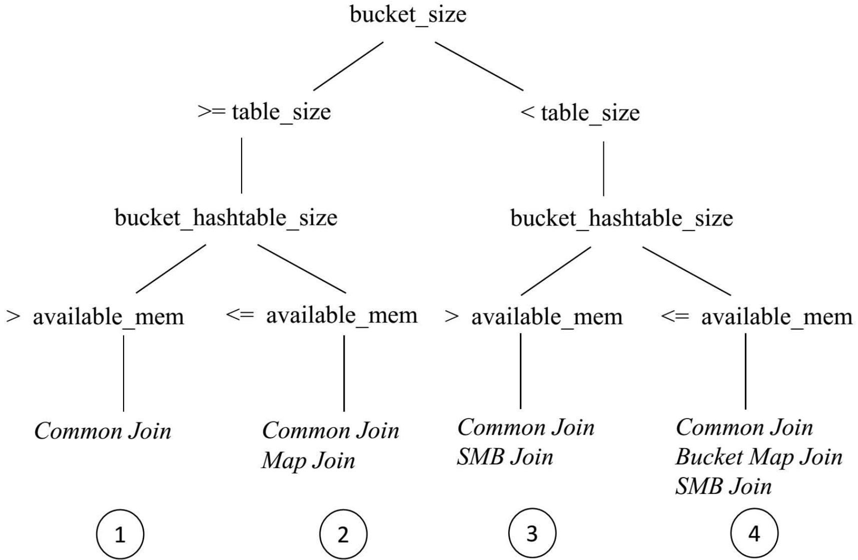

Fig. 1. Join Algorithm Candidates Tree

the leave node ③ or the leave node ④. As we can see, the two input parameters including the table size and available memory may lead to the selection of a different join algorithm.

In the tree structure of Fig. 1, we note that due to limited buckets, the leaf node ③ *SMB Join* always perform worse than the leaf node ④. *Common Join* has already been contained in all the rest leave nodes. Consequently, we further prune the leave node ③, and have a new structure as shown in Fig. 2. In the new tree structure, namely *Pruned Join Algorithm Candidates Tree* (*Pruned-JACTree*), now only one leave node contains the *SMB Join.*

4.2 Select Join Algorithm Based on Cost Model

In order to enable the proposed cost model, we need to know the values of key parameters used by the cost model. To this end, we estimate such parameters as follows.

We first estimate the parameters including the size of join tables *S(T1)*, *S(T2)*, the number of rows of them *R(T1)*, *R(T2)*, and the total field size of query columns *S(C(T1))*, *S(C(T2)).* In detail, when a table is uploaded to HDFS, with the help of Hive log, we can find the values of *S(T1)*, *S(T2)* and *R(T1)*, *R(T2).* Next, we use MapReduce `RandomSampler` utility to estimate the total field size of query columns. By the `RandomSampler`, we first set the number of input splits that will be sampled. The sampler will then randomly sample the selection columns of two join tables for the estimation of *S(C(T1))* and *S(C(T2)).*

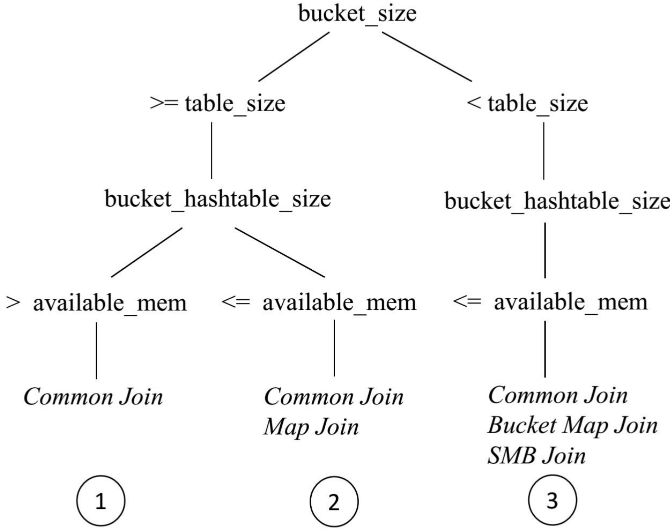

Fig. 2. Pruned Join Algorithm Candidates Tree

Second, we estimate the *available_mem* by the minimum JVM memory in task nodes. In terms of the memory used by the hashtable, we can compute the value by the number of rows in the table.

$$Hashtable_Mem(T) = R(T) * \alpha \tag{3}$$

In Eq. (3), the parameter α indicates the number of bytes per record in the hashtable needed by the main memory. In our experiment, we empirically set α by 200 bytes.

Finally, we need to decide the suitable bucket number $N_{buckets}$. A small data size per bucket may lead to too many but small size of files in HDFS and slow down the query. Alternatively, a very large data size per bucket will incur very few `map` tasks, and the hashtable of one single bucket is too large to fit into memory for *Bucket Map Join*. Thus, we determine the number of bucket number with the following function:

$$Bucket_Num(T, available_mem) = \frac{\beta * Hashtable_Mem(T)}{available_mem} \tag{4}$$

In the Eq. (4), a higher β (> 1) means a larger number of buckets, to ensuring that the buckets should be loaded into memory on the node with higher probability. In our experiment environment, we empirically set $\beta = 1.3$.

When the above parameters are ready, we use Fig. 3 to describe the steps that our selection method (namely a selector) chooses the best algorithm among the

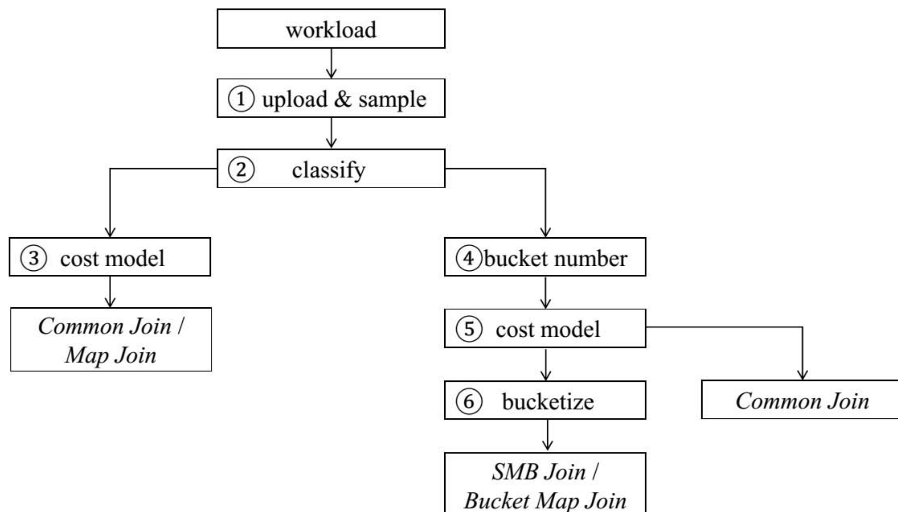

Fig. 3. Cost-based Selector Workflow

four available join algorithms. ① For an input workload, selector first collects join tables' information for cost model. ② Selector next makes a decision to choose an algorithm by following either of the two brunches. ③ If one of the join table in the query is small enough to fit into memory, the selector computes the cost of *Common Join* and *Map Join*, and finds the one with a smaller cost as the chosen join algorithm. ④ Otherwise, our selector uses Eq. (4) to calculate the reasonable bucket number for the join tables. ⑤ Based on the proposed cost model, the selector next decides to use either bucketed join (i.e., *Bucket Map Join* or *SMB Join*) or *Common Join*. ⑥ If the bucketed join is chosen, our selector will only select the columns needed by the queries in the workload to the buckets while bucketizing.

5 Experiment

We present experiments to evaluate our proposed selection method, and study (1) how the number of buckets affects the performance of bucketed joins, and (2) whether the proposed selection method can correctly choose the best join algorithm for a given workload.

Cluster Setup. We evaluate the experiments on a 5-node cluster with 1 namenode and 4 datanodes. Each of the nodes is installed with Ubuntu Linux (kernel version 2.6.38-16) and equipped with 750 GB local disk and 4 GB main memory. We implement our cost-based join algorithm selector on the top of Hadoop 1.2.1 and Hive 0.11.0. The default block size is 64 MB.

Table 2. # Rows and Size of Data Sets

Table Name	# Rows	Size
lineItem0.5x	3,000,000	365 M
lineItem30x	180,000,000	22.0 G
lineItem100x	600,000,000	74.1 G
orders1x	1,500,000	164 M
orders30x	45,000,000	4.9 G
orders100x	150,000,000	16.6 G

Datasets. We generate the synthetic data sets (`LineItems` and `Orders`) by the TPC-H benchmark, and generate different scales of join tables to simulate different combinations of input tables. Table 2 shows the size and number of rows of different scales of input join tables.

Workload. We only generate the join query workload with the simple equi-join queries between the `LineItems` and `Orders` tables. In our experiment, we design the following queries:

```
Q1: SELECT l.l_shipmode, o.o_orderstatus
FROM lineitem100x l JOIN orders1x o ON l.l_orderkey = o.o_orderkey;

Q2: SELECT l.l_shipmode, o.o_orderstatus
FROM lineitem0.5x l JOIN orders1x o ON l.l_orderkey = o.o_orderkey;

Q3: SELECT l.l_shipmode, o.o_orderstatus
FROM lineitem30x l JOIN orders30x o ON l.l_orderkey = o.o_orderkey;

Q4: SELECT l.l_linenumber, o.o_orderkey
FROM lineitem30x l JOIN orders30x o ON l.l_orderkey = o.o_orderkey;

Q5: SELECT l.l_shipmode, o.o_orderdate
FROM lineitem30x l JOIN orders30x o ON l.l_orderkey = o.o_orderkey;

Q6: SELECT l.l_shipmode, o.o_orderstatus
FROM lineitem100x l JOIN orders100x o ON l.l_linestatus = o.o_orderstatus;
```

Practical Cost Estimation. During our experiment, to use the proposed cost model, we need to estimate the cost of some key operations such as reading HDFS blocks, etc. At first, we denote the cost of transferring 1 GB data through network cost as 1 G, and the CPU cost of comparing 10^6 times as 1 C. Based on the denotation, by around tens of empirical test, we empirically draw the following equations:

(1) $T_{ReadHDFS}(m) = 0.6 * T_{Transfer}(m) = 0.6 * m \quad G$
(2) $T_{Bucketize}(t) = T_{ReadHDFS}(S(t)) + 4 * T_{Transfer}(S_c(t)) \quad G$
(3) $T_{Hashtable}(r) = 30 * r * 10^{-6} \quad C$
(4) $T_{Join}(r1, r2) = (r1 + r2) * 10^{-6} \quad C$

In the above first equation, considering that the data is partially located on local nodes, we empirically compute the cost of reading HDFS by 60% of the cost of transferring data through network. Next, we estimate the bucketizing cost by the cost of reading, transferring and writing data. Since writing data to HDFS is always ineffcient, we empirically set the cost of writing HDFS 3 times as the cost of transferring data. After that, we calculate the cost of building one key-value pair of hashtable as 30 times of CPU comparison. Finally, the cost of join operation is estimated as processing all rows of two join tables.

In our experiments' environment, the network bandwith is about 100 M/s and CPU latency for one comparison is around 0.1 microsecond, we determine 1 $C = 10^{-2}$ G.

5.1 Effect of the Number of Buckets

In order to demonstrate how the number of buckets can affect the performance of *Bucket Map Join* and *SMB Join*, we have conducted an experiment with the query Q_6 on the tables `LineItems` and `Orders` of scale 30. We vary the number of buckets from 1 to 2000, and the corresponding query time is shown in Fig. 4.

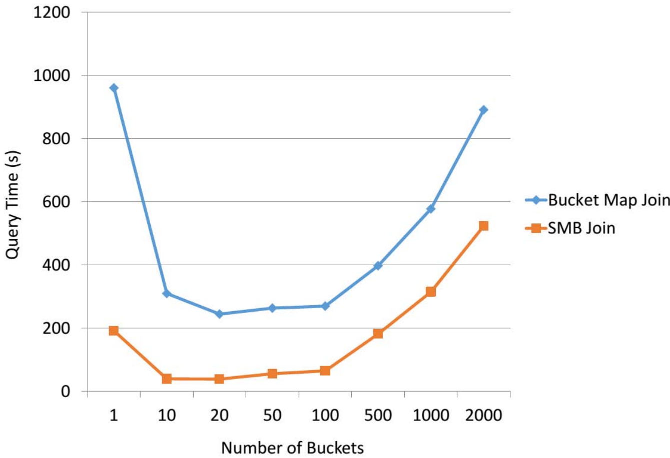

Fig. 4. Effect of the Number of Buckets

As shown in Fig. 4, with a smaller number of buckets, each bucket contains a larger data size, and the mapper has to process a large amount of data. For *Bucket Map Join*, since the hashtable of the buckets may be too large to fit into memory, it may fail to broadcast the hashtable and Hive has to use the default *Common Join*. On the other hand, too many buckets will trigger too many mappers and each mapper only joins a very fewer number of records.

In addition, Fig. 4 indicates that the number 20 of buckets can achieve the best performance, which is consist with our analysis. In our experiment cluster,

the maximum JVM memory on each tasknode is around 900 MB and the average usage of it is around 25%. So we assume the minimum JVM memory available on the tasknode is 600 MB. `orders30x` has 45,000,000 rows. According to Eq. (3), the hashtable of it will need 9,000 MB memory space. With Eq. (4), our selector will set the reasonable bucket number as 20. The result shows when the bucket number is 20, the join performance is the best for both bucketed joins.

5.2 Correctness of Cost-Based Selector

In this section, given multiple scenarios, we evaluate the correctness of our cost model and verify whether or not our selector can pick the best join algorithm.

One Small Table First, we consider the scenario that one of the join table's hashtable is small enough to fit into memory. For the Q_1 and Q_2 in our workload, `orders1x`'s hashtable can be fit into memory. Table 3 illustrates the real execution time and our cost model theoretical result. We can see the order of the cost result is roughly the same as the order of execution time, which means our selector can pick the join algorithm with the lowest query latency.

Table 3. Compare with the Cost-model Result

Query	**Join Algorithm**	**Execution Time**(s)	**Cost Result**(G)
Q1	Common Join	1022	59.5
	Map Join	659	52.0
Q2	*Common Join*	31	0.41
	Map Join	39	0.78

Two Large Tables. Next we consider the scenario with two large tables. For the queries from Q_3 to Q_6, neither of the join tables can fit into memory. Given this case, our selector will decide whether or not to bucketize the tables, tune the best number of buckets and choose one best algorithm among *Common Join, Bucket Map Join* and *SMB Join*.

From Table 4, we can see for the queries Q_3, Q_4 and Q_5, our selector chooses *SMB Join*. It is because all these three queries benefit from bucketizing tables. Therefore the cost of bucketizing is divided by 3. In terms of Q_6, it is the only query to join the tables on `l_linestatus` and `o_orderstatus`, and our selector chooses *Common Join* because bucketizing is also a time-consuming MapReduce job.

In addition, the experiments results indicate that our selector does not pick *Bucket Map Join* for any of the six queries. By careful analysis, we find that the

Table 4. Compare with the Cost-model Result

Query	**Join Algorithm**	**Execution Time**(s)	**Cost Result**(G)
Q3	Common Join	367	24.3
	Bucket Map Join	498/3+269=435	32.2/3+24.7=29.1
	SMB Join	498/3+40=206	32.2/3+4.7=15.4
Q4	Common Join	356	23.8
	Bucket Map Join	498/3+249=415	32.2/3+24.7=29.1
	SMB Join	498/3+40=206	32.2/3+4.7=15.4
Q5	Common Join	368	24.7
	Bucket Map Join	498/3+218=484	32.2/3+24.7=29.1
	SMB Join	498/3+43=209	32.2/3+4.7=15.4
Q6	*Common Join*	1237	72.5
	Bucket Map Join	1271+1700=2971	86.4+59.7=146.1
	SMB Join	1399+184=1583	86.4+13.9=100.3

reason why *Bucket Map Join* always performs worst is that it not only needs to bucketize in advance but also needs to build and then broadcast the hashtable for each bucket. These efforts incur expensive cost.

As a summary, for the given workload with six queries from Q_1 to Q_6, the order of our cost model result is roughly the same as the real execution time. With our cost-based selector, the total execution time of the workload is accelerated 24.4%, compared with the default join algorithm (*Common Join*). Compared with the wrong join algorithm, which new users may randomly pick, the total execution time is accelerated 58.7%.

6 Related Work

First, some previous works implement new join algorithms on top of MapReduce. For example, Lin *et al* propose a new scheme called "schimmy" to save the network cost during the *Common Join* [6]. In this scheme, mappers only emit messages and reducers read the data directly from HDFS and do the `reduce`-side join between the messages and data. Okcan *et al* propose how to efficiently perform $\theta-$join with a single MapReduce job [7]. Their algorithm uses a Reducer-centered cost model that calculates the cost of Cartesian product of mapped output. With this cost model, they assign the mapped output to the reducers that minimizes job completion time. Blanas *et al* propose the process of *Semi-Join* and *Per-Split Semi-Join* in MapReduce framework [8]. Lin *et al* propose the concurrent join that performs a multi-way join in parallel with MapReduce [9]. Afrati *et al* focus on how to minimize the cost of transferring data to reducers for multi-way join [10]. Verica *et al* propose a method to efficiently parallelize set-semilarity joins with MapReduce [11].

Second, there exist some works to hackle the internals of MapReduce. Map-Reduce-Merge is the first that attempts to optimize join operation in the MapReduce framework [12]. Map-Reduce-Merge extends MapReduce model by adding

Merge stage after Reduce Stage. Yang *et al* have proposed an approach for improving Map-Reduce-Merge framework by adding a new primitive called *Traverse* [13]. This primitive can process index file entries recursively, select data partitions based on query conditions and feed only selected partitions to other primitives. Jiang *et al* propose Map-Join-Reduce for one-phase joining in MapReduce Framework. This work introduces a filtering-join-aggregation model as another variant of the standard MapReduce framework [14]. This model adds a Join Stage before the original Join Stage to perform the join. Besides the introduction of more stages, some works columnar to improve the join queries. Llama [15] is a recent system that combines columnar storage and tailored join algorithm. Llama also proposed joining more than two tables at a time using a concurrent join algorithm. Clydesdale [16], a novel system for structured data processing is aimed at star schema, using columnar storage, tailored plans for star schemas and multi-core aware execution plans to accelerate joins.

Finally, some works propose to build new layer on top of Hadoop in order to process a join query. A typical work is Hive, a data warehouse infrastructure built on top of Hadoop. Hive optimizes the chains of map joins with the enhancement for star joins. Olston *et al* have presented a language called *Pig Latin* [17] that takes a middle position between expressing task using the high-level declarative querying model in the spirit of SQL and the low-level/procedural programming model using MapReduce. *Pig Latin* is implemented in the scope of the *Apache Pig* project [18]. Pig Latin programs are compiled into sequences of MapReduce jobs which are executed using the Hadoop MapReduce environment. Pig optimizes join by using several specialized joins, such as fragment replicate joins, skewed joins, and merge joins. The *Tenzing* system [19] has been presented by Google as an SQL query execution engine which is built on top of MapReduce and provides a comprehensive SQL92 implementation with some SQL99 extensions. Tenzing's query optimizer applies various optimizations and generates a query execution plan that consists of one or more MapReduce jobs.

7 Conclusion

In this paper, we proposed a general cost model for the four popular join algorithms in Hive and designed a method to choose the best join algorithm with least cost. Our experiment results verified that our cost-based selector can correctly choose the best join algorithm and tune the key parameters (such as the number of buckets). As the future work, we will further extend our work to generally support more complex join queries such as multiway join.

Acknowledgments. This work was partially funded by EMC, and was a part of a joint research project between Fudan University and EMC Labs China (the Office of CTO, EMC Corporation). This work was also partially funded by the NSFC (Grant No. 61370080).

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI, pp. 137–150 (2004)
2. Hadoop, http://hadoop.apache.org/
3. Hive, http://hive.apache.org/
4. TPC-H, Benchmark Specification, http://www.tpc.org/tpch/
5. HDFS architecture, http://hadoop.apache.org/docs/r0.19.1/hdfs_design.html
6. Lin, J., Dyer, C.: Data-intensive text processing with MapReduce. Synthesis Lectures on Human Language Technologies 3(1), 1–177 (2010)
7. Okcan, A., et al.: Processing Theta-Joins using MapReduce. In: Proceedings of the 2011 ACM SIGMOD (2011)
8. Blanas, S., et al.: A comparison of join algorithms for log processing in MapReduce. In: Proceedings of the 2010 ACM SIGMOD, pp. 975–986 (2010)
9. Lin, Y., et al.: Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. In: Proceedings of the 2011 ACM SIGMOD (2011)
10. Afrati, F.N., Ullman, J.D.: Optimizing joins in a map-reduce environment. In: Proceedings of the 13th EDBT, pp. 99–110 (2010)
11. Vernica, R., et al.: Efficient parallel set-similarity joins using mapreduce. In: Proceedings of the 2010 ACM SIGMOD, pp. 495–506 (2010)
12. Yang, H., et al.: Map-reduce-merge:simplifiedrelational data processing on large clusters. In: Proceedings of the 2007 ACM SIGMOD, pp. 1029–1040 (2007)
13. Yang, H.-c., Parker, D.S.: Traverse: Simplified Indexing on Large Map-Reduce-Merge Clusters. In: Zhou, X., Yokota, H., Deng, K., Liu, Q. (eds.) DASFAA 2009. LNCS, vol. 5463, pp. 308–322. Springer, Heidelberg (2009)
14. Jiang, D., et al.: Map-join-reduce: Towards scalable and efficient data analysis on large clusters. IEEE Transactions on Knowledge and Data Engineering (2010)
15. Lin, Y., Agrawal, D., Chen, C., Ooi, B.C., Wu, S.: Llama: leveraging columnar storage for scalable join processing in the MapReduce framework. In: SIGMOD Conference, pp. 961–972 (2011)
16. Balmin, A., Kaldewey, T., Tata, S.: Clydesdale: structured data processing on hadoop. In: SIGMOD Conference, pp. 705–708 (2012)
17. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD, pp. 1099–1110 (2008)
18. Pig, http://pig.apache.org/
19. Chattopadhyay, B., Lin, L., Liu, W., Mittal, S., Aragonda, P., Lychagina, V., Kwon, Y., Wong, M.: Tenzing A SQL Implementation On The MapReduce Framework. PVLDB 4(12), 1318–1327 (2011)