International Journal of Grid and Distributed Computing
Vol. 6, No. 3, June, 2013

Efficient Processing Distributed Joins with Bloomfilter using MapReduce [†]

Changchun Zhang, Lei Wu, and Jing Li[*]

School of Computer Science and Technology, University of Science and Technology of China, Hefei, 230026, China
{zccc, xcwulei}@mail.ustc.edu.cn, lj@ustc.edu.cn

Abstract

The MapReduce framework has been widely used to process and analyze large-scale datasets over large clusters. As an essential problem, join operation among large clusters attracts more and more attention in recent years due to the utilization of MapReduce. Many strategies have been proposed to improve the efficiency of distributed join, among which bloomfilter is a successful one. However, the bloomfilter's potential has not yet been fully exploited, especially in the MapReduce environment. In this paper, three strategies are presented to build the bloomfilter for the large datasets using MapReduce. Based on these strategies, we design two algorithms for two-way join and one algorithm for multi-way join. The experimental results show that our algorithms can significantly improve the efficiency of current join algorithm. Moreover, cost models of these algorithms are characterized in order to find out the way of improving the performance of two-way and multi-way joins.

***Keywords**: Bloomfilter, MapReduce, Query optimization, Cost model*

1 Introduction

Cloud computing has been gaining more and more attention in industry and academia. In this area, the large-data analysis is a very important issue worthy of in-depth research. There are two kinds of distributed systems designed to process and analyze large datasets: parallel relational database system and MapReduce-based system. According to CAP [1], consistency, availability and tolerance to network partitions, cannot be simultaneously fulfilled in distributed systems. Parallel relational database system, aiming at the pursuit of a higher level of consistency and fault tolerance, cannot reach ideal scalability. Thus this system is not suitable for the requirement of large-scale data analysis. In recent years, the MapReduce [2] framework developed by Google, has become an extremely popular tool for processing and analyzing large datasets in cluster environments, mostly due to its simple interface, good fault tolerance, load balancing and scalability over thousands of nodes.

[†] The work is supported by the National Science and Technology Supported Program "National Financial Data Analysis and the Key Technologies and Applications of Information Service Based on Cloud Computing (2012BAH17B03)" and USTC-Lenovo Cloud Computing Laboratory.

[*] to whom correspondence should be addressed.

The join algorithm has been studied for many years. The researchers ended up developing different methods to improve the efficiency including: semijoin [3] and bloomjoin [4]. Especially in the database, the bloomfilter [5] has been successfully used to reduce network overhead and improve the overall join efficiency. However, how to efficiently use the bloomfilter in a distributed system, is not precisely clear, especially in the MapReduce environment. In this paper, our purpose is to investigate the potential of the bloomfilter for distributed joins using MapReduce. The contributions of this paper are as follows:

1. Three strategies are presented to build a bloomfilter for large datasets using MapReduce. The experimental results show that our method is both feasible and efficient. Moreover, this bloomfilter building approach not only can be used in distributed query processing situation, but could also be integrated to other applications.

2. Different algorithms are designed for the optimization of two-way and multi-way joins using the bloomfilter. Various experiments are conducted to evaluate these bloomjoin algorithms. The results illustrate that our algorithms can improve join algorithm's efficiency.

3. Cost models of our algorithms are evaluated based on the I/O cost of the map and reduce phases. With these models, we can choose the superior bloomjoin implementation, by using MapReduce for two-way and multi-way joins.

The rest of this paper is organized as follows: Section 2 presents the overview of MapReduce and bloomfilter. Section 3 discusses how to efficiently build a bloomfilter for a large dataset using MapReduce. Section 4 outlines several bloomjoins using MapReduce and reports the results of experiments. Section 5 characterizes these algorithms cost models and points out how to improve the performance of the two-way and multi-way joins. Section 6 reviews related work on the topic. Section 7 presents our conclusion and future work.

2 MapReduce and Bloomfilter

2.1 MapReduce

The MapReduce model is shown in Figure 1. It is so easy to program in MapReduce that the only thing the user needs to do is overwrite a map and reduce certain functions. The map function takes a key-value pair $(k1, v1)$ as the input from the Distributed File System (DFS) generating some other pairs of $(k2, v2)$ as an intermediate result. The reduce function takes as input the intermediate result as a $(k2, list(v2))$ pair, where list $(v2)$ is a list of all $v2$ values associated with a given key $k2$. The reduce function produces other key-value pairs as a final result.

$$map(k1, v1) \longrightarrow list(k2, v2)$$

$$reduce(k2, list(v2)) \longrightarrow list(v2)$$

2.2 Bloomfilter

The bloomfilter data structure is a compact data structure used for testing an element to dataset membership. The example for a bloomfilter is shown in Fig. 2. It

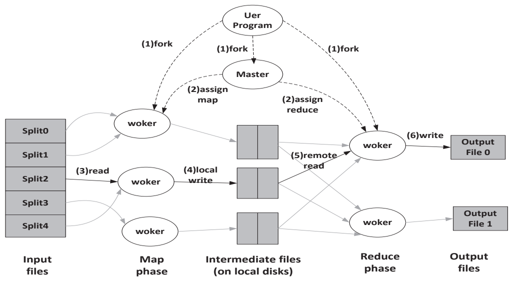


Figure 1. Overview of MapReduce

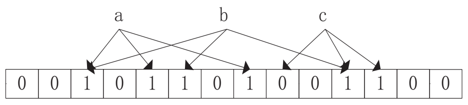


Figure 2. Example of a bloomfilter

consists of an array of l bits and a set of h hash functions, which hashes the dataset element to an integer in the range of $[1, l]$. All bits of the array are initialized to zero. Each hash function maps an element to some bits of the filter. In order to check the membership of an element, we must look at the h positions. Positively positive answer is only provided if all h bits are set to 1. In Fig. 2, element a and b are in this dataset, while element c is not in. The bloomfilter allows false positives, but never false negatives.

Given a set $R(x)$, the bloomfilter for the relation R on the attribute x is denoted as $BF_R(x)$. We can compute the required size l of $BF_R(x)$ and the number h of the hash functions for a given number of elements $R(x)$ and a false positive bounded by p in [6].

$$l_p = -\frac{\ln p}{(\ln 2)^2} \tag{1}$$

The total size of the bloomfilter for the entire set $R(x)$ is

$$l = l_p \times |R| = -\frac{\ln p}{(\ln 2)^2} \times |R| \tag{2}$$

3 Computing Bloomfilters using MapReduce

Before dealing with the problem of bloomfilters for distributed joins using MapReduce, an important and independent problem must be solved first, which is how to efficiently build a bloomfilter for a relation $R(x)$ on the attribute x using MapReduce, with a false probability of at most p in the building phase, and to check the membership in this bloomfilter in the filtering phase. We present and analyze three strategies for the problem question.

$Strategy1$. The map function creates a bloomfilter $BF_{R'}(x)$ for its local data R' of its own partition, where $|BF_{R'}(x)| = l_p \times |R|$, corresponding to the size of the bloomfilter for the entire relation R. The intermediate results of the map function output will be sent to a single reducer. The reduce function unions the intermediate results by a bit-wise OR operation. The example of this strategy is shown in Fig. 3(a). Actually, the union of the bloomfilters is precisely the bloomfilter for the relation R and the false probability is equal to p. According to formula (2), we can obtain that

$$|BF_{R'}(x)| = l_p \times |R| = \frac{-\ln p}{(\ln 2)^2} \times |R| \tag{3}$$

*Strategy*2. The map function computes a bloomfilter $BF_{R'}(x)$ for its local data $R^{'}$ of its own partition, where $|BF_{R'}(x)| = l_{p'} \times |R^{'}|$, corresponding to the size of the bloomfilter for one block in the Hadoop Distributed File System (HDFS) but not to the whole relation $|R|$. The intermediate results of the map function output will just be connected together and stored in HDFS directly in one reducer. The final bloomfilter contains m parts, where m is both the block number of relation R stored in HDFS and the number of mappers. In order to check the existence of a value in this bloomfilter, each of the m bloomfilters must be checked. Only if all m bloomfilters answer negatively can we answer negatively. The example of this strategy is shown in Fig. 3(b). The false probability of the final bloomfilter must be equal to p, thus $p = 1-(1-p^{'})^m$, where $p^{'}$ is the false probability on the bloomfilter for $R^{'}$. According to formula (2), we can obtain that

$$|BF_{R'}(x)| = l_{p'} \times |R|/m = \frac{-\ln p^{'}}{(\ln 2)^2}|R|/m = \frac{-\ln(1-(1-p)^{1/m})}{(\ln 2)^2} \times |R|/m \qquad (4)$$

*Strategy*3. This strategy is a hybrid of *strategy*1 and *strategy*2. Suppose that we have k reducers. The map function computes a bloomfilter $BF_{R'}(x)$ of its own partition, where $BF_{R'}(x) = l_{p'} \times |R|/k$. Each reduce function receives m/k bloomfilters, unions them by a bit-wise OR operation and stores the results in HDFS. The final bloomfilter for relation R contains k parts because there are k reducers. The example of this strategy is shown in Fig. 3(c). Similar to *strategy*2, in order to guarantee that the false probability of the final bloomfilter is equal to p, thus let $p = 1-(1-p^{'})^k$. According to formula (2), we obtain that the following:

$$|BF_{R'}(x)| = l_{p'} \times |R|/k = \frac{-\ln p^{'}}{(\ln 2)^2} \times |R|/k = \frac{-\ln(1-(1-p)^{1/k})}{(\ln 2)^2} \times |R|/k \qquad (5)$$

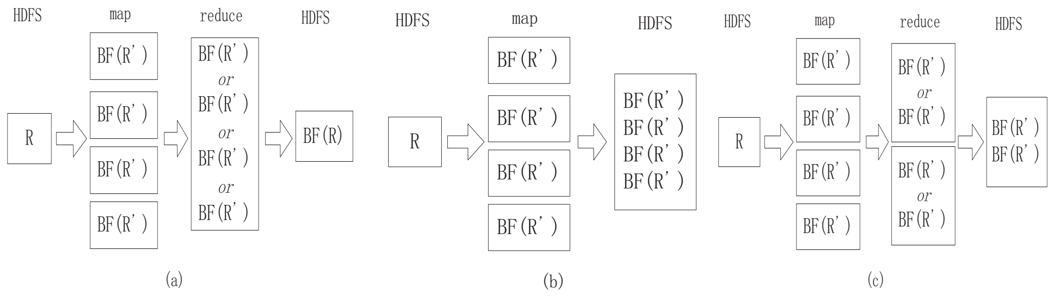


Figure 3. Examples of the strategies

We can find out that *strategy*3 is the same as *strategy*1 when $k = 1$ and the same as *strategy*2 when $k = m$. Compared with *strategy*1, *strategy*3 adopts k reducers to union all intermediate results from the map output in the building phase. Compared with *strategy*2, the look-up cost of *strategy*3 in the filtering phase is lower than the one of *strategy*2 with $k < m$.

We have carried out an experiment aiming to evaluate the performance of these strategies. There are 10 blades running Hadoop 0.20.2 [7], each with 2.4GHz*12 core CPU, 20G RAM, 270G hard disk. All blades are directly connected to a Gigabit switch. Each blades runs at most 11 map tasks and 11 reduce tasks. The other major Hadoop parameters are listed in Table 1.

Table 1. Hadoop parameter confuguration

Parameter	value	Parameter	value
fs.blocksize	$64M$	io.sort.spill.percentage	0.8
io.sort.mb	$100M$	io.sort.factor.	100
io.sort.record.percentage	0.05	dfs.replication	3

The size of both relation $R(A, B)$ and $S(B, C)$ is 100 million records; each record has two attributes. Firstly, We build $BF_R(B)$ for relation $R(A, B)$ and use $BF_R(B)$ to filter relation $S(B, C)$. The false probability p is 0.01. The result is shown in Fig. 4. The building phase of $strategy1$ requires much more time than the one of $strategy2$ and $strategy3$, since a lot of elements need to be combined by an OR operation in a single reducer. In $strategy3$, the execution efficiency of the building phase improves as the number of reducers increases, while the execution efficiency of the filtering phase decreases only a little. In $strategy2$, the building phase costs the least time, however the filtering phase costs more time than the one of $strategy3$, resulting in the longer overall execution time compared with $strategy3$. Moreover, the larger the size of relation S to be filtered grows, the less efficient $strategy2$ will be. In a word, $strategy3$ is superior to $strategy1$ or $strategy2$.

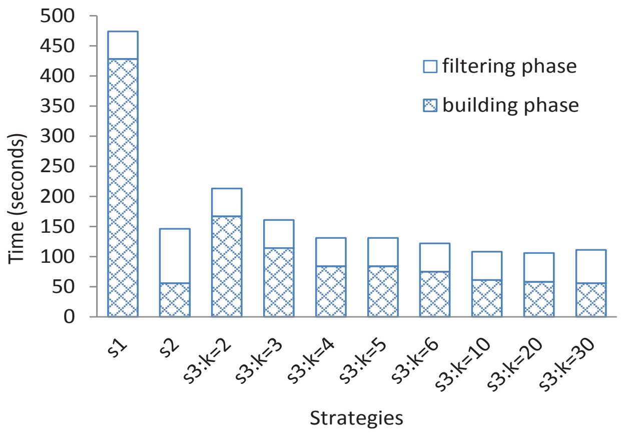


Figure 4. MapReduce time for the building phase and the filtering phase

At last, a reasonable range of k value should be set in $Strategy3$. We make a tradeoff between the building phase and the filtering phase. In Fig. 4, we can find out the filtering phase efficiency decreases slowly as k value increases. In the building phase, the communication cost between map and reduce tasks $|BF_{R'}(x)|$ will decreases as k value increases, since $|BF_{R'}(x)| = \frac{-\ln(1-(1-p)^{1/k})}{(\ln 2)^2} \times |R|/k = \frac{(\ln k - \ln p)}{k(\ln 2)^2} \times |R|$ ($\ln(1-(1-p)^{1/k}) = p/k$ as p is small). Suppose that p is set to 0.01. When increasing k from 1 to 10, 10 to 20 and 20 to 30, the length of $|BF_{R'}(x)|$ decreases from $9.2|R|$ to $1.4|R|$, $1.4|R|$ to $0.7|R|$ and $0.7|R|$ to $0.53|R|$, respectively. Thus, when $k > 20$, large increase of k would result in little declining of $|BF_{R'}(x)|$ but decreasing of the filtering phase efficiency. In conclusion, the value k should be set to the range of 10 to 20 for most cases.

4 Bloomjoins using MapReduce

In this section, we focus on how to use the bloomfilter to improve join algorithm efficiency. The concept of bloomjoin is based on the semijoin technique. Instead of transmitting the entire relation R, we use a bloomfilter based on relation S to filter part of relation R. A large fraction of relation R's tuples will be possibly rejected without being hashed. This will improve the efficiency of the join algorithm. There are two kinds of cases needing to be considered: two-way joins and multi-way joins.

4.1 Two-way Joins using MapReduce

Let us assume that each record of R and S has two attributes, two algorithms are designed to use the bloomfilter to compute $R(A, B) \bowtie S(B, C)$: one with two stages and the other one with three stages.

$TwoStageAlg$. This algorithm consists of two stages, each corresponding to a separate MapReduce job. In the first MapReduce job, a bloomfilter on the attribute B is built for relation R or S. Without loss of generalization, suppose the bloomfiler is built for R, denoted as $BF_R(B)$. Then, the map function uses $BF_R(B)$ to filter relation S in the second MapReduce job, and outputs the extracted join key and the tagged record of relation R and S as a $(key, value)$ pair. All the records for each join key are grouped together and eventually sent to a reducer. For each join key, the reduce function separates and buffers the input records into two sets according to the relation tag and then performs a cross-product between records in these sets. An example of the second job is shown in Fig. 5(a).

$ThreeStageAlg$. This algorithm consists of three stages, each corresponding to a separate MapReduce job. The first MapReduce job is the same as that of $TwoStageAlg$. In the second MapReduce job, the bloomfilter $BF_R(B)$ is broadcasted to filter relation S. This phase runs as a map-only job. The output of this phase is a list of files S_i, one for each split of S. In the third MapReduce job, all the S_i are joined with R, also using a map-only job. Examples of the second and third MapReduce jobs are shown in Fig. 5(b).

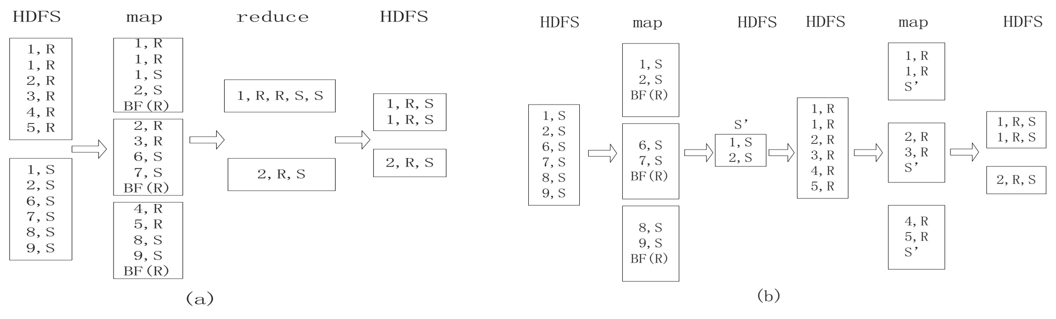


Figure 5. Examples of the bloomjoin algorithms

There two algorithms are compared with an improved repartition join [8] in terms of the size of two relations and the reference rate. The improved repartition join is a standard two-way join method using MapReduce, which has been used in Hive [9]. The experimental environment has been described in Section 3. The result is shown in Fig. 6. It can be observed that $ThreeStageAlg$ and $TwoStageAlg$ are less efficient than the improved repartition join when the size of the relation is small, because additional MapReduce rounds are needed to build the bloomfilters. However, they are far more efficient than improved repartition join, when the size of the relation grows to over 50 million records, since the bloomfilters can filter a lot of useless data to save network overhead and processing overhead. $TwoStageAlg$ is more suitable for large reference rate and large data size than $ThreeStageAlg$. The reason is that although the map-side join in the third stage of $ThreeStageAlg$ is more efficient than the reduce-side join in the second stage of $TwoStageAlg$ when the reference rate is small, the performance of the map-side join in the third stage of $ThreeStageAlg$ degrades rapidly as the reference rate and data size increase, since the cost of transferring S_i to every node across the network and the loading S_i in memory starts to dominate.

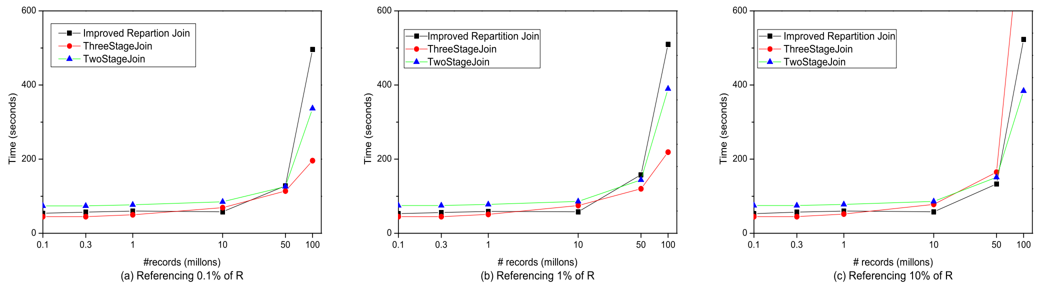


Figure 6. MapReduce time for an Improved repartition join and Bloomjoin

4.2 Multi-way Joins using MapReduce

Let us consider the case of a 3-way join: $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$. Naturally, we can implement this join by a sequence of two two-way joins, choosing either to join R and S first, and then join T with the result. However, this method is time consuming. Recently, Afrati and Ullman proposed an algorithm to deal with this join using only one MapReduce job [10]. The main procedure is as follows. Let h be a hash function with range $1 \dots n$, and associate each reduce processor with a pair(i, j), where each of the integers i and j is between 1 and n. Each tuple $S(B, C)$ is sent to the reduce processor numbered $(h(b), h(c))$. Each tuple $R(A, B)$ is sent to all reduce processors numbered $(h(b), x)$, for any x. Each tuple $T(C, D)$ is sent to all reduce processors numbered $(y, h(C))$ for any y. Each Reduce processor computes the join of the tuples it receives. The example is shown in Fig. 7. Although this algorithm is efficient, there are still a lot of useless tuples to be copied in this process. Naturally, the bloomfilter can be used to filter useless data and eventually improve the efficiency of the multi-way joins.

We have introduced an algorithm called multi-way-bf join based on Afrati and Ullman's algorithm. Multi-way-bf join consists of two MapReduce jobs. In the first MapReduce job, $BF_S(B)$ and $BF_S(C)$ are built for the relation S on the attribute B and C. In the second MapReduce job, we use $BF_S(B)$ and $BF_S(C)$ to filter relation R and T and adopt Afrati and Ullman's algorithm to get the final results.

Multi-way-bf join is compared with the two two-way joins and Afrati and Ullman's algorithm, denoted by multi-way-nobf joins. The result is shown in Fig. 8. Multi-way-bf join is less efficient than the other two methods when the relations are

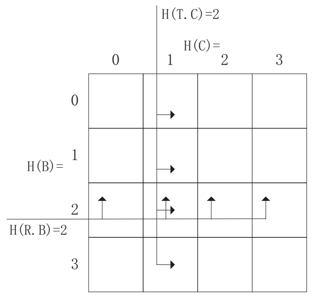


Figure 7. Example of Afrati's algorithm

small, because the multi-way-bf join adds an extra MapReduce job to build the bloomfilters. While the size of the relations grows to over one million, the multi-way-bf join is more efficient than the others, as it uses the bloomfilter to filter a lot of useless data to save both the network and processing overhead.

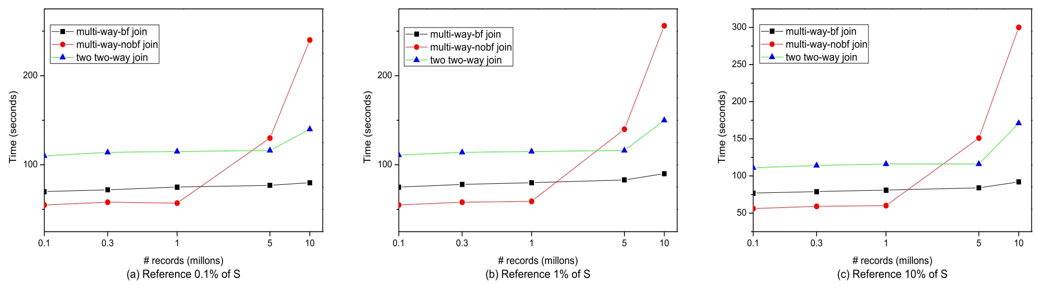


Figure 8. MapReduce time for multi-way joins

5 Cost Model of Bloomjoins using MapReduce

In the last section, we are aware that these two-way or multi-way bloomjoins algorithms perform differently in terms of reference rate and data size. Since our target is to find an optimal join plan, the time costs of these algorithms need to be estimated, including the built bloomfilters and bloomjoins.

Generally, for MapReduce jobs, the heavy cost on a large-scale sequential disk scan, as well as the frequent I/O of intermediate results, are to dominate the entire execution time [11]. Therefore, the execution time model for a MapReduce program should be built based on the analysis of disk I/O cost and network I/O cost. Since entire input data may not be loaded into the system memory within one round [12], the model should also take account of this situation. We assume these map tasks are performed round by round. Before the discussion, parameters used in the analysis are listed in Table 2.

5.1 Cost Model of Computing Bloomfilters using MapReduce

We hereby estimate the total cost of $stratege3$ to build a bloomfilter for relation R. For each map task, it receives the split of relation R. The disk I/O cost t_{bf-map}

Table 2. Parameters

Definition	Parameter	Definition	Parameter
cost ratio of HDFS reads	ρ	join selectivity of R_1 and R_2	$g(R_1, R_2)$
cost ratio of HDFS writes	η	number of mappers	m
cost ratio of Network I/O	μ	number of reducers	k
number of tuples in R	$\|R\|$	accumulative selectivity of R	α_R
size of R' tuple	$f(R)$	projection selectivity of R	β_R

of each map task is

$$t_{bf-map} = \rho \times \frac{|R|f(R)}{m} \tag{6}$$

As a general assumption, relation R is considered to be evenly partitioned among the m map task and m' is the current number of map tasks running in parallel in the system. The total cost of the map phase T_{bf-map} is

$$T_{bf-map} = t_{bf-map} \times \frac{m}{m'} \tag{7}$$

let $t_{bf-shuffle}$ be the cost for copying the output of a single map task to k reduce tasks, including the data copying over network cost, as well as overhead of all serving network protocols.

$$t_{bf-shuffle} = \mu\frac{|BF_{R'}(B)|}{k} + q \times k = \mu\frac{-\ln(1-(1-p)^{1/m})}{(\ln 2)^2}|R|/k^2 + q \times k \tag{8}$$

q is a random variable which represents the cost of a map task serving k connections from k reduce tasks. Intuitively, there is a rapid growth of q as k gets larger. Since there are m/m' rounds in the map phase, thus the total cost of the shuffle phase $T_{bf-shuffle}$ can be computed as follows:

$$T_{bf-shuffle} = t_{bf-shuffle} \times \frac{m}{m'} \tag{9}$$

Each reduce task receives m/k bloom filters, unions them by a bit-wise OR operation and stores the results to HDFS. Thus the cost of the single reduce task $T_{bf-reduce}$ is

$$T_{bf-reduce} = \eta|BF_{R'}(B)| = \eta\frac{-\ln(1-(1-p)^{1/k})}{(\ln 2)^2} \times |R|/k \tag{10}$$

Hence the total cost of building a bloomfiter for relation R using MapReduce is

$$T_{bf} = T_{bf-map} + T_{bf-shuffle} + T_{bf-reduce} \tag{11}$$

5.2 Cost Model of Two-way Bloomjoins

We evaluate the cost of $TwoStageAlg$. In the first MapReduce job, the bloomfilter $BF_R(B)$ is built for relation R. The cost of this job is denoted as T_{bf}^R, which has been previously described in Section 5.1. In the second MapReduce job, each map task receives the split of relations R and S and the whole bloomfilter $BF_R(B)$

whose size is $k_{bf}|BF_{R'}|$, where k_{bf} is the number of reduce tasks needed to build the bloomfilter in the first MapReduce job. The disk I/O cost in the map phase is

$$T_{bfjoin1-map} = \rho(\frac{|R|f(R) + |S|f(S)}{m} + k_{bf}|BF_{R'}|) \times \frac{m}{m'} \tag{12}$$

In the shuffle phase, each map task will send the tuple of R and S to k reducers. Because the false probability of the bloomfilter is equal to p, there are $d_{RS}|S|$ tuples to be sent to the reducers, where $d_{RS} = g(R,S) + p(1 - g(R,S))$. Let $T_{bfjoin1-shuffle}$ be the cost for copying the output of a single map task to k reduce tasks.

$$T_{bfjoin1-shuffle} = \mu(\frac{\alpha_R\beta_R|R|f(R)}{mk} + \frac{\alpha_S\beta_S d_{RS}|S|f(S)}{mk} + qk) \times \frac{m}{m'} \tag{13}$$

α denotes the output ratio of a map task, which is query specific and can be computed with the selectivity estimation. β denotes the projection selectivity (the tuple size is reduced to $\beta \times 100\%$ of its original size after ruling out the unnecessary columns). Let $T_{bfjoin1-reduce}$ be I/O cost to HDFS of each reduce task

$$T_{bfjoin1-reduce} = \eta\frac{\alpha_R\alpha_S|R||S|g(R,S)(\beta_R f(R) + \beta_S f(S))}{k} \tag{14}$$

Thus, the cost of $TwoStageAlg$ is

$$T_{bfTwoStageAlg} = T_{bf}^{R} + T_{bfjoin1-map} + T_{bfjoin1-shuffle} + t_{bfjoin1-reduce} \tag{15}$$

Similarly, we can also build a bloomfilter $BF_S(B)$ for the relation S and then use $BF_S(B)$ to the bloomjoin. The cost of this approach is denoted as $T'_{bfjoinTwostage}$.

We then evaluate the cost of $ThreeStageAlg$. The first MapReduce job is to build a bloomfilter. The second and the third MapReduce jobs are map-only jobs. The cost of the first MapReduce job is denoted as $T_{bf}^{R_B}$ which has been described in Section 5.1. In the second MapReduce job, each map task filters relation S as $BF_R(B)$. The I/O cost of each map task is $T_{bfjoin2-map}$

$$T_{bfjoin2-map} = \rho(\frac{|S|f(S)}{m} + k_{bf}|BF_{R'}|) \times \frac{m}{m'} \tag{16}$$

As there are no shuffle and reduce phases, the output of the map task will be directly stored to HDFS. Let $T_{bfjoin2-reduce}$ be the I/O cost to HDFS.

$$T_{bfjoin2-reduce} = \eta\frac{\alpha_S\beta_S|S|f(S)d_{RS}}{m} \tag{17}$$

In the third MapReduce job, all the S_i are joined with R, also using a map-only job. The disk I/O cost of the map phase is

$$T_{bfjoin3-map} = \rho(\frac{|R|f(R)}{m} + \alpha_S\beta_S|S|f(S)d_{RS})\frac{m}{m'} \tag{18}$$

Due to the lack of shuffle and reduce phases, the output of the map phase will be stored to HDFS. Let $T_{bfjoin3-reduce}$ denote the I/O cost to HDFS.

$$T_{bfjoin3-reduce} = \eta\frac{\alpha_R\alpha_S g(R,S)(\beta_R|R|f(R) + \beta_S|S|f(S))}{m} \tag{19}$$

The total cost of $ThreeStageAlg$ is

$$\begin{aligned} T_{bfjoinThreestage} = T_{bf}^{R_B} &+ T_{bfjoin2-map} + T_{bfjoin2-reduce} \\ &+ T_{bfjoin3-map} + T_{bfjoin3-reduce} \end{aligned} \tag{20}$$

Similarly, we can build a bloomfilter $BF_S(B)$ for relation S and then use $BF_S(B)$ to the bloomjoin. The cost of this approach is denoted as $T'_{bfjoinThreestage}$.

Suppose that T_{irj} denotes the cost of the improved repartition join. As the way to estimate T_{irj} is similar to that of the second stage of $TwoStageAlg$, we no longer describe it.

Now the total costs of these three algorithms can be formulated as formula (21), so we can select a best implementation for two-way joins using MapReduce.

$$T = \begin{cases} T_{irj} & bloomfilter\ is\ not\ used \\ T_{bfjoinTwoStage} & bloomfilter\ for\ R(B) \\ T'_{bfjoinTwoStage} & bloomfilter\ for\ S(B) \\ T_{bfjoinThreeStage} & bloomfilter\ for\ R(B) \\ T'_{bfjoinThreeStage} & bloomfilter\ for\ S(B) \end{cases} \tag{21}$$

5.3 Cost Model of Multi-way Joins using MapReduce

The cost of our multi-way joins algorithm can be estimated in the similar way as that of two-way joins. Suppose $R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,A)$ needs to be computed. We first build one bloomfilter for attribute B of relation R_1 in the first MapReduce job. The cost of this job has been described in Section 5.1, denoted as $T_{bf}^{R_1(B)}$. In the second MapReduce job, the disk I/O cost $T_{bfnwayjoin-map}$ of each map task is

$$T_{bfnwayjoin-map} = \rho(\frac{S_I^{R_1,R_2,R_3}}{m} + k_{bf}|BF_{R_1'}(B)|) \times \frac{m}{m'} \tag{22}$$

Where $S_I^{R_1,R_2,R_3} = |R_1|f(R_1) + |R_2|f(R_2) + |R_3|f(R_3)$. Let c_x denote the number of reducers for attribute $x(x \in \chi, \chi = \{A,B,C\})$, thus $k = \prod_{x=1}^{x=3} c_x$. In order to improve the performance, the number of required reducers is set to be proportional to the size of corresponding relation according to [10], i.e. $c1 : c2 : c3 = \alpha_1|R_1| : (p\alpha_2|R_2| + (1-p)\alpha_2|R_2|g(R_1,R_2)) : \alpha_3|R_3|$. Thus, c_x $(1 \leq x \leq 3)$ can be computed. For relation R_i, if it contains a join attribute set $\chi'(\chi' \subset \chi, \chi = \{A,B,C\})$, we need to replicate its data to c_{R_i} reducers, where $c_{R_i} = \prod_{\forall r_i \notin \chi' \bigwedge r_i \in \chi} c_x$. Let $T_{bfnwayjoin-shuffle}$ be the cost for copying the output of single map task to k reduce tasks

$$T_{bfnwayjoin-shuffle} = \mu(\frac{I_{R_1}}{mk} + \frac{I_{R_3}}{mk} + \frac{I_{R_2}}{mk}d_{R_1R_2} + qk) \times \frac{m}{m'} \tag{23}$$

where $I_{R_i} = c_{R_i}\alpha_{R_i}\beta_{R_i}|R_i|f(R_i)$. The I/O cost of each reduce task is estimated as

$$T_{bfnwayjoin-reduce} = \eta\frac{J(\beta_{R_1}f(R_1) + \beta_{R_2}f(R_2) + \beta_{R_3}f(R_3))}{k} \tag{24}$$

where $J = \alpha_{R_1}\alpha_{R_2}\alpha_{R_3}|R_1||R_2||R_3|g(R_1,R_2,R_3)$. The total cost of the second MapReduce job is

$$T_{bfnwayjoin} = T_{bfnwayjoin-map} + T_{bfnwayjoin-shuffle} + T_{bfnwayjoin-reduce} \tag{25}$$

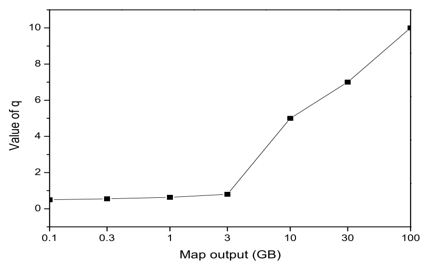


Figure 9. Value distribution of q

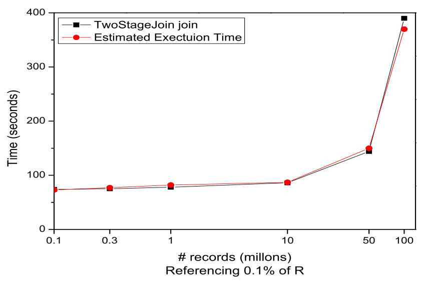


Figure 10. TwoStageJoin Validation

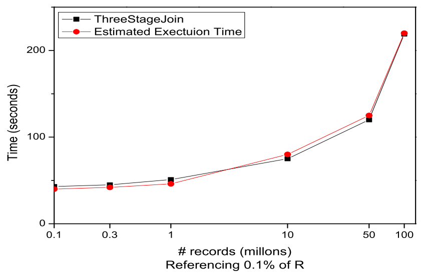


Figure 11. ThreeStageJoin Validation

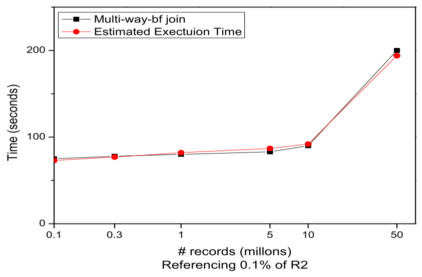


Figure 12. Multi-way-bfJoin Validation

The total cost of the multi-way bloomjoins is

$$T_{bfnway}^{R_1(B)} = T_{bf}^{R_1(B)} + T_{bfnwayjoin} \tag{26}$$

However, please note that R_2 can be filtered with bloomfilter $BF_{R_1}(B)$, $BF_{R_3}(C)$ or even both. Similarly, R_1 and R_3 can also be filtered with $BF_{R_2}(B)$ and $BF_{R_2}(C)$. Thus, the question is: Given a condition join involving n relations and a set of joining attributes $(A, B, \ldots)$, which subset of the join attributes should we use to filter the rest of the relations so as to minimize the overall execution time? To answer this question, we can obtain all the possible implementation costs and calculate the minimal cost.

$$\min cost = \min T_{bfnway}^{R_i(x)\ldots R_j(y)} \tag{27}$$

5.4 Cost Model Validation

The experimental environment has been described in Section 3. We use the TestDFSIO program to test the I/O performance of the system. In our cost model, the cost ratios of of Table 2 are set as follows: HDFS read (ρ)=1, HDFS write (η)=1.5, network I/O (μ)=1.5. Then we should compute the distribution of q which serves the estimation of MapReduce's running time. The number of reduce tasks is set to 99 and the value of q is computed by studying an output controllable program over a series of test data. The result is shown in Fig. 9. Then bloomfilter $BF_R(B)$ for relation R and bloomfilter $BF_{R_1}(B)$ for relation R_1 are built in two-way joins and three-way joins, respectively. The results are shown in Fig. 10-12. We can come to the conclusion that our estimation and the real MapReduce exection time are very close.

6 Related Work

The bloomfilter introduced in [5] is a probabilistic data structure used for checking membership in a set. The survey on the bloomfilter applied to different applications is discussed in [6]. It was used for the efficient distributed join computation in [4] for the first time. The authors proposed an algorithm called bloomjoin, aimed at reducing communication cost. In [13], the authors presented some extensions and pointed out how they could improve the performance of the distributed join computation. In [14], the authors extended bloomjoin to minimize the network usage for query execution based on database statistics. However, the above mentioned for the extension of the bloomfilter in the distributed environment are different from ours. They all assumed that the relations were placed in different servers, but not partitioned, and they also did not take into consideration the efficiency of computing the bloomfilter for one relation. [15] is similar to our work, however, the results over a practical implementation of the bloomfilter using MapReduce was not studied.

In recent years, the join optimization using MapReduce is a highly-discussed topic. In [16], the design patterns of the join algorithm using MapReduce can be divided into the reduce-side join and the map-side join. In [8], the authors implemented and compared several 2-way join algorithms. In [10], a problem on how to optimize the multi-way equi-joins in a single MapReduce job was solved. In [11, 17], the authors optimized the multi-way theta-joins using MapReduce. In [18], they presented the Map-Join-Reduce, a system that extends and improves the MapReduce system to efficiently process data analytical tasks. In [19], the authors proposed a new function merge() for simplifying the join processing operations. The interested reader can refer to the surveys on these algorithms [20].

7 Conclusion and Future Work

In this paper, we have achieved three goals. The first goal is how to efficiently build the bloomfilter for a large dataset using MapReduce. Three strategies are suggested for this problem and the results show that $strategy3$ is superior to $strategy1$ or $strategy2$. The second goal is to apply the bloomfilter to joins using MapRecuce. Several bloomjoin algorithms have been designed and the experimental results illustrate that our algorithms can improve the performance of two-way and multi-way joins. The third goal is to evaluate these bloomjoins' costs. We have evaluated these costs and used these costs to select the appropriate bloomjoin implementation using MapReduce.

In the future, we plan to develop a dynamic cost analyzer. This will help us to implement the best MapReduce approach to any multi-way joins problems. We are also planning to investigate techniques of incorporating Hadoop parameters into the cost model and enhance the join efficiency.

8 Acknowledgements

Thanks for the platform and the technical support provided by the Supercomputing Center of University of Science and Technology of China. This paper is a revised and expanded version of a paper entitled "Optimizing Distributed Joins with

Bloomfilters using MapReduce", presented at Proceedings of the 2012 Conference on Grid and Distributed Computing, pp. 88-94, (2012) December 16-19, Korea.

References

[1] S. Gilbert and N. Lynch, "Brewer' s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services", ACM SIGACT News, pp. 51-59, (2002).

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Proceedings of the 6th Symposium on Operating Systems Design and Implementation, (2004) December 6-8; California, USA.

[3] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve and J. B. Rothnie, "Query processing in a system for distributed databases(sdd-1)", ACM Transactions on Database Systems, Vol. 6, No. 4, pp. 602-625, (1981).

[4] L. F. Mackertm and G. M. Lohman, "R* optimizer validation and performance evaluation for local queries", Proceedings of the 1986 ACM SIGMOD international conference on Management of data, (1986) May 28-30; Washington, D.C., USA.

[5] B. H. Bloom, "Sapce/time trades-offs in hash coding with allowable errors", Communications of the ACM, 13(7): 422-426, (1970).

[6] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey", Internet Mathematics, pp. 636-646, (2002).

[7] http://hadoop.apache.org/2012.

[8] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita and Y. Tian, "A Comparison of Join Algorithms for Log Processing in MapReduce", Proceedings of the 2010 ACM SIGMOD international conference on Management of data, (2010) June 6-11; Indianapolis, Indiana.

[9] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu and R. Murthy, "Hive - A Petabyte Scale Data Warehouse Using Hadoop", Proceedings of the 26th International Conference on Data Engineering, (2010) March 1-6; California, USA.

[10] F. N. Afrati and J. D. Ullman, "Optimizing Multiway Joins in a Map-Reduce Environment", IEEE Transaction on Knowledge and Data Engineering, Vol. 23, No. 9, pp. 1282-1297, (2011).

[11] X. Zhang, L. Chen and M. Wang, "Efficient Multi-way Theta-Join Processing Using MapReduce", Proceedings of the VLDB Endowment, (2012) August 27-31; Istanbul, Turkey.

[12] P. Agrawal, D. Kifer and C. Olston, "Scheduling shared scans of large data files", Proceedings of the VLDB Endowment, (2008) August 24-30; Auckland, New Zealand.

[13] L. Michael, W. Nejdl, O. Papapetrou and W. Siberski, "Improving distributed join efficiency with extended boom filter operations", Proceedings of the 21st International Conference on Advanced Information Networking and Applications, (2007) May 21-23; Niagara Falls, Canada.

[14] S. Ramesh, O. Papapetrou and W. Siberski, "Optimizing Distributed Joins with Bloom Filters", Distributed Computing and Internet Technology, Vol. 5375, pp.

145-156, (2009).

[15] P. Koutris, "Bloom Filters in Distributed Query Execution", CSE 544 Project, University of Washington, (2011).

[16] J. Lin and C. Dyer, "Data-Intensive Text Processing with MapReduce", Synthesis Lectures on Human Language Technologies, (2010).

[17] C. Zhang, J. Li, L. Wu, M. Lin and W. Liu, "SEJ: An Even Approach to Multiway Theta-Joins using MapReduce", Proceedings of the 2nd International Conference on Cloud and Green Computing, (2012) November 1-3; Hunan, China.

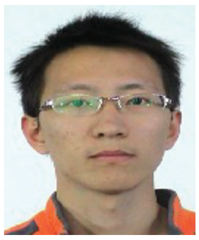
[18] D. Jiang, A. K. H. Tung and G. Chen, "MAP-JOIN-REDUCE: Toward Scalable and Efficient Data Analysis on Large Clusters", IEEE Transaction on Knowledge and Data Engineering, Vol. 23, No. 9, pp. 1299-1311, (2011).

[19] H. Yang, A. Dasdan, R. Hsiao and D. S. Parker, "Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters", Proceedings of the 2007 ACM SIGMOD international conference on Management of data, (2007) June 12-14; Beijing, China.

[20] K. -H. Lee, Y. -J. Lee, H. Choi, Y. D. Chung and B. Moon, "Parallel Data Processing with MapReduce: A Survey", Proceedings of the 2011 ACM SIGMOD international conference on Management of data, (2011), June 12-16; Athens, Greece.

Authors

Changchun Zhang received the B.S. degree in computer science from the University of Science and Technology of China (USTC) in 2009. He is currently working toward the Ph.D. degree at the Department of Computer Science of USTC. His research interests include parallel algorithms, Cloud Computing and high performance computing.



Lei Wu received the B.S. degree in computer science from the University of Science and Technology of China (USTC) in 2009. Now he is a M.E. candidate in Computer Science at USTC. His research interests parallel algorithms, Cloud Computing and high performance computing.

Jing Li received his B.E. in Computer Science from University of Science and Technology of China (USTC) in 1987, and Ph.D. in Computer Science from USTC in 1993. Now he is a Professor in the School of Computer Science and Technology at USTC. His research interests Distributed Systems, Cloud Computing and Mobile Computing.