Toward Intersection Filter-Based Optimization for Joins in MapReduce

Thuong-Cang Phan  
Blaise Pascal University  
CNRS, UMR 6158, LIMOS  
Clermont-Ferrand, France  
ThuongCang.Phan@isima.fr

Laurent d’Orazio  
Blaise Pascal University  
CNRS, UMR 6158, LIMOS  
Clermont-Ferrand, France  
Laurent.Dorazio@isima.fr

Philippe Rigaux  
CEDRIC  
CNAM  
Paris, France  
Philippe.Rigaux@cnam.fr

ABSTRACT

MapReduce has become an attractive and dominant model for processing large-scale datasets. However, this model is not designed to directly support operations with multiple inputs as joins. Many studies on join algorithms including Bloom join in MapReduce have been conducted but they still have too much non-joining data generated and transmitted over the network. This research will help us eliminate the problem by providing an intersection filter based on probabilistic models to remove most disjoint elements between two datasets. Namely, three ways are proposed to build the intersection Bloom filter. To apply the filter to joins, a corresponding MapReduce job will be adjusted in a consistent way without increasing related costs. We then consider two-way joins and join cascades and analyze their costs. As a result, thanks to the high accuracy intersection filter, join processing can minimize disk I/O and communication costs. Finally, the research is proved to be more effective than existing solutions through a cost-based comparison of joins using different approaches.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems - Distributed databases, Parallel databases. Query processing.

General Terms


Keywords

Data analysis, Cloud computing, Join, MapReduce, Bloom filter.

1. INTRODUCTION

MapReduce has become popular and has played an increasingly important role for big data processing in Cloud computing environments. Unfortunately, basic complex operations in MapReduce are used extensively and expensively. The join operation is considered as a paradigm of such operations. It is not only one of the essential operations for data analysis [5][24] but also one of the most elementary operations for query evaluation [1][23]. Although there have been many studies on join algorithms in parallel and distributed DBMSs [3][4][21], a MapReduce environment is not straightforward to implement joins. This is because MapReduce does not directly support operations with multiple inputs. The research efforts have markedly expanded to address this problem and given several solutions [2][5][22] where a join operation will be compiled to MapReduce job(s). For these solutions, however, it is realized that many intermediate results generated in the map phase do not actually participate in the joining process. Consequently, it would be more efficient if we eliminated the unnecessary data right in the map phase. The problem can be partially addressed through using a filter called Bloom Filter (BF) [7] that is a space-efficient probabilistic structure to test set membership. BF is built for one of input datasets and is delivered across all the mappers. When the mapper receives tuples from another dataset of the inputs, it eliminates any tuples whose join keys are not in the filter. Obviously, this solution can only filter out non-joining tuples from one of input datasets but not from both input datasets. Thus, it is necessary to have a better solution to improve this situation.

From observation, the result of the join operation only contains tuples whose join keys belong to the intersection of the datasets projected on the join key. This paper presents a new type of Bloom filter called Intersection Bloom filter, representing the intersection of the datasets to be joined. Unlike previous solutions that filter only on one dataset, our proposal filters out disjoint elements or non-joining tuples from both datasets. Each tuple from the input datasets is queried into the intersection filter and will be removed if it is a disjoint element.

This paper, therefore, makes three main contributions: (a) three approaches proposed to compute the intersection filter that approximates the intersection of datasets; (b) the feasibility of the approaches used in two-way joins and join cascades; (c) the evidence that this intersection filter results in significantly better filtration efficiency than basic Bloom filters in joins.

The remainder of this paper is organized as follows. First, Section 2 summarizes existing join algorithms in MapReduce and provides a brief introduction to Bloom filters. In Section 3, the approaches for modelling the intersection Bloom filter are described. Two-way join implementation using these approaches is presented in Section 4. In addition, Section 5 introduces a cost model-based evaluation of the approaches and a comparison with other solutions. Finally, Section 6 shows the advantage of the extended intersection filter for important join cases. Section 7 includes conclusions and introduces implications on future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Cloud’13, August 26, 2013, Riva del Garda, Trento, Italy.  
Copyright 2013 ACM 978-1-4503-2108-2/13/08 ...$15.00.
2. RELATED WORKS

2.1 Join Algorithms in MapReduce

Join algorithms in MapReduce are referred from the existing join algorithms in the standard and parallel RDBMS literature such as sort-merge join [6], hash-based join [20], semijoin [21], etc. In MapReduce, they are generally categorized into three major types i.e. Reduce-side join also known as repartition join [5][13], Map-side join [5][13], and Broadcast join [5] as a hash join. There have been studies concerning improving the join algorithms. One of them has focused on the problem of minimizing the number of the tuples replicated to the reducers before passing data to joining process have also been explored, e.g., semi-join and per-split semi-join [5], parallelize set-similarity joins [22], etc. A recent cost-effective solution of these options is the filter efficiency by using intersection Bloom filters.

The Bloom filter (BF) [7] is a randomized data structure used to test membership in a set with a small rate of false positives. The Bloom filter BF(S) for representation of a set S is described as follows:

- The set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements is represented by an array of m bits, initially all set to 0.
- The filter uses k independent hash functions $h_1, h_2, \ldots, h_k$ with $h_i: x \rightarrow \{1, \ldots, m\}$.
- To insert an element $x \in S$, we compute $h_1(x), h_2(x), \ldots, h_k(x)$, and set the corresponding positions in the bit array to 1. Once we have done this operation for each element of $S$, we should have a bit array that acts as an approximate representation of the set.
- To check if $y \in S$, we check whether for each of the k hash functions, the position $h_i(y)$ is set to 1 in the bit array. If at least one of these positions is set to 0, it is clear that $y \notin S$. Otherwise, all the positions are set to 1, we know that $y$ may be a member of S with some probability.

One prominent feature of the Bloom filter is that the size of the filter is space-efficient and fixed regardless of the number of the elements of the set $S$, but there is a clear tradeoff between the size of the filter and the false positive probability.

A partitioned Bloom filter [11], a variant of Bloom filters, is defined by an array of m bits that is partitioned into k disjoint arrays of size $m_y = m/k$ bits. Figure 3 suggests that $BF(S)$ consists of three 4-bit partitions, $k=3$ hash functions and size $m=12$ bits.

2.2 Unpartitioned & Partitioned Bloom Filters

Unfortunately, the filter efficiency of the mentioned algorithm and even with the recent extended researches [12][14] has not really been taken into consideration yet. There remain many non-joining tuples after filtering. Namely, with using only the filter $BF(S)$ for both the two inputs $R$ and $S$, the algorithm can only eliminate non-joining tuples of the dataset $R$ (e.g., tuples with the join key values of 2 and 4) without eliminating non-joining tuples of the dataset $S$ (e.g., tuples with the join key values of 3 and 6). The actual result of the join operation only contains tuples whose join keys belong to the intersection of the input datasets projected on the join key, i.e., \( \{ t \mid t.x \in R.x \land t.x \in S.x \} \). As shown in Figure 1, the output is tuples whose join keys have the same value as 1 and belong to the intersection of $R.x$ and $S.x$: $\{1\} = \{1, 2, 4\} \cap \{1, 3, 6\}$.

Thus, our join approach is designed to significantly improve the filter efficiency by using intersection Bloom filters.

![Figure 1: Basic join operation using BF in MapReduce](image)

![Figure 2. An Unpartitioned Bloom filter BF(S)](image)

![Figure 3. A Partitioned Bloom filter BF(S)](image)
partitioned filters with different sizes can be performed by using the bit-wise AND (OR) of two bit-arrays. Obviously, the resolution of one of partitioned Bloom filters may be adjusted. For example, the partitioned filter BF(R) has three 4bit-partitions (the filter size $m_1=12$ bits), BF(S) has two 4bit-partitions (size $m_2=8$ bits). To perform intersecting two filters, we only eliminate the third partition of BF(R), then AND two remaining partitions of BF(R) and BF(S). This affects the resolution of BF(R).

Meanwhile, we cannot reduce the size of the unpartitioned filter because we have to completely rehash the bit-array of the filter.

3. MODELLING INTERSECTION FILTER

In reality, Bloomjoin [15] is a popular method of joining in distributed databases since it reduces the amount of data transferred over the network. Many studies have extended this algorithm to optimize complex distributed joins [16][17][19]. Recently, thanks to MapReduce advantages in processing large datasets, researchers have also used Bloomjoins with modifications necessary to adapt them to the environment [12][14]. However, these filters only have the ability to remove redundant data from one of the input datasets instead of both; hence, there remains a large amount of redundant data of another dataset that is passed to join processing. A better replacement for the basic filters is the intersection filter with the ability to filter out disjoint elements between two datasets. This section therefore shows three approaches to build the intersection filter.

For convenience during this paper, two Bloom filters BF(R) and BF(S) are used as the concise representation of two input datasets R and S projected on the join key column respectively. The intersection Bloom filter of R and S on the join key, denoted as BF(R $\cap$ S), can be represented by a logical operation or the bitwise AND operation of two bit-arrays.

3.1 Approach 1: A pair of Bloom filters

First, we observe the following expression for set intersection representation

$$R \cap S = (R \cup S) \setminus (R \Delta S) = (R \cup S) \setminus ((R \setminus S) \cup (S \setminus R))$$

From here, we can specify the set intersection by eliminating all disjoint elements between the input datasets. In other words, we filter out the disjoint elements in the dataset R by BF(S) and disjoint elements in the dataset S by the BF(R). To perform this work, we use a pair of Bloom filters as follows.

$$\begin{align*}
R & \rightarrow BF(S) \\
S & \rightarrow BF(R)
\end{align*}$$

![Figure 4. A pair of Bloom filters](image)

Each tuple in one input dataset is queried into the Bloom filter of the other input dataset by $k$ hash functions. If its join key is a member of the filter, we retain the tuple containing this key because the key is a common member of the two input datasets. Otherwise, we remove the tuple from its dataset because its join key is a disjoint member and this tuple is a non-joining tuple. As Figure 4, for instance, a tuple with the join key $x_1$ of the dataset R is queried into BF(S) and a tuple with the join key $y_1$ of the dataset S is queried into BF(R). If $x_1$ is not a member of BF(S), we remove the tuple from R. If $y_1$ is in BF(R), we keep the tuple with the key $y_1$ in S. After performing |R| queries into BF(S) and |S| queries into BF(R), it corresponds to the operation of $(R \cup S) \setminus (R \Delta S)$. In other words, the intersection filter BF(R $\cap$ S) can be obtained through the pair of Bloom filters.

This approach does not require the filters to have the same size $m$ and $k$ hash functions.

3.2 Approach 2: Intersection of unpartitioned Bloom filters

This approach is based on the idea that intersecting Bloom filters will produce a result filter called the intersection filter.

There is unfortunately little difference between the intersection filter and the intersection of Bloom filters as shown in [9] then $BF(R \cap S) = BF(R) \cap BF(S)$ with probability $(1-1/|k|)^{|R \cap S|}$. Obviously, the intersection of filters is not sufficient to accurately calculate the intersection filter $BF(R \cap S)$.

However, we can get an approximation by joining $BF(R)$ and $BF(S)$ with the bit-wise AND and it still maintains the inherent querying features [8][9]. This means that if all $k$ hash functions for the join key $x$ map to 1 bits in the intersection filter, $x$ belongs to $R \cap S$ with high probability.

In the second approach, we should use unpartitioned Bloom filters with the same size $m$ and $k$ hash functions. Building the intersection filter is shown in the following figure.

$$\begin{align*}
&1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \\
\land &0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \\
\rightarrow &0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1
\end{align*}$$

![Figure 5. Intersection of unpartitioned Bloom filters](image)

It illustrates that the intersection filter $BF(R \cap S)$ is formed by intersecting two basic filters $BF(R) \cap BF(S)$ with the bit-wise AND and it is an approximate representation of the set intersection $R \cap S$.

With this approach, it now allows us to use only one intersection Bloom filter to remove most non-joining tuples from both input datasets R and S instead of using two filters as the first approach.

3.3 Approach 3: Intersection of partitioned Bloom filters

Our last approach begins with the same idea as the second one to create the intersection filter but partitioned Bloom filters are used.

As previously mentioned, the size of partitioned Bloom filters can be changed after they are created. In the current case, the filters may have different sizes but their partitions should have the same size. Therefore, we can adjust the partitioned filters $BF(R)$, $BF(S)$ and $BF(R \cap S)$ by reducing or adding partition(s) without rehashing in order to ensure their same size $m$ and $k$ hash functions. We build the intersection filter from partitioned Bloom filters as described in Figure 6.
filter for both input datasets

Even for the second approach, it would be rare for all characteristic is really useful for joins and is not present in the processing can be finished without doing anything. This filters with 3 partitions are pairwise intersected with the bit-wise bits equal to 0, the two input datasets are disjoint. So the join exists at least one partition of the result filter containing all datasets

partitions of two partitioned filters. As shown in Figure 6, two

THEOREM 2. A false intersection by intersecting partitioned filters is identified with probability

\[ f_{\text{par}} = \left(1 - \left(\frac{1}{m_1}\right)^k \right) \left(1 - \left(\frac{1}{m_2}\right)^k \right) \]

where BF(R), BF(S) and BF(R ∩ S) have the same size m and k hash functions.

THEOREM 3. A false intersection by intersecting partitioned filters is identified with probability

\[ f_{\text{par}} = \left(1 - \left(\frac{1}{m}\right)^k \right) \left(1 - \left(\frac{1}{m}\right)^k \right) \]

where BF(R), BF(S) and BF(R ∩ S) have the same size m and k hash functions, k partitions are the same size m = m/k.

THEOREM 4. The false intersection probability of the unpartitioned filter intersection is less than the false intersection probability of the partitioned filter intersection

\[ f_{\text{par}} < f_{\text{par}} \]

\[ m = m/k \]

The dark area (blue) in Figure 7 shows the actual intersection of the input datasets R and S. The bright area inside the dashed oval represents the false positives of the set intersection also known as false intersections.

We can show the probability of a false intersection as follows.

\[ f_{\text{par}} = \left(1 - \left(\frac{1}{m_1}\right)^k \right) \left(1 - \left(\frac{1}{m_2}\right)^k \right) \]

where \( m_1 \) and \( m_2 \) correspond to the size of BF(R) and BF(S); \( k_1 \) and \( k_2 \) are the number of hash functions of BF(R), BF(S) respectively.

THEOREM 2. A false intersection by intersecting unpartitioned filters is identified with probability

\[ f_{\text{BF}} = \left(1 - \left(\frac{1}{m}\right)^k \right) \left(1 - \left(\frac{1}{m}\right)^k \right) \]

\[ m = m/k \]

The intersection filter is generated by intersecting pairs of partitions of two partitioned filters. As shown in Figure 6, two filters with 3 partitions are pairwise intersected with the bit-wise AND to produce the result filter including three 4-bit partitions.

Similar to the second approach, we also use one intersection filter for both input datasets R and S to filter out redundant tuples.

3.4 The false intersection probability

We can represent the intersection of two input datasets with the false positives as follows.
Because $R$ and $S$ are two arbitrary input datasets, we will discuss and evaluate the join operation in general by using Reduce-side join algorithm. It can however still use our intersection filter for other join algorithms.

The join operation with the support of our intersection filter solution is described as follows.

A. Pre-processing step

A join job together with its input datasets $R$ and $S$ is configured and submitted by a client to the jobtracker. At this moment, the join job is blocked until the end of a pre-processing job. The pre-processing job, which is a job chained to our join job, includes two subjobs ($sjob1$ and $sjob2$) running in parallel for building the intersection filter. The $sjob1$ processes the dataset $R$ for creating $BF(R)$ while the $sjob2$ independently processes the dataset $S$ for creating $BF(S)$. A subjob, a map-only job without output, reads tuples from splits and builds local Bloom filters in memory on tasktrackers. Once all map tasks of a subjob are complete, the jobtracker signals all tasktrackers and they send their local filters via heartbeat responses to the jobtracker. Then, the jobtracker merges the local Bloom filters to generate two global filters $BF(R)$ and $BF(S)$. Next, it computes and builds the intersection filter $BF(R \cap S)$ based on our proposals from the global filters. For the first approach, the intersection filter is a pair of the global filters and thus it does nothing. At the end of the pre-processing step, if the intersection filter is empty, the entire join job will be finished and the jobtracker responds with instructions to stop tasks or the job. Otherwise, the intersection filter is distributed to all nodes in a cluster using a distributed cache. This step corresponds to activities (1) to (6) in Figure 8.

Notably, the pre-processing step is written as a standard map. Such a map is run in a single job and does not leave any intermediate file. As a consequence, it enables a dramatic reduction in I/O. Besides, our implementation can detect empty intersection to early finish the join job. This interesting feature is not present in existing studies.

We will mention a method to merge the local filters presented in Section 4.2 that takes place at the jobtracker through using a union of filters.

B. Filtering and Map phase

In order to start the join job, the jobtracker will create $mp_1$ and $mp_2$ map tasks for inputs $R$ and $S$ respectively, $r$ reduce tasks and assign each split to one map task run on a tasktracker. The mapper reads each tuple from its split, produces a <key, tuple> pair, and then call a map function to process the pair. The map function queries the key of the tuple into the intersection filter $BF(R \cap S)$. If it exists, then the mapper emits a tagged pair <dataset-name::join-key, tuple> and sends this pair to a corresponding reducer, else filter it out.

C. Reduce phase

The reduce function will take its input and do a full cross-product of tuples of different input datasets for a given join key to get our joined output. It is completed by writing the output to Distributed File System (DFS).

4.2 Merging of Bloom filters

Fortunately, merging of Bloom filters is much simpler than the intersection of Bloom filters. The merging operation corresponds to the construction of the union filter $BF(R \cup S)$. We can build the union Bloom filter by the union of the filters without probability as shown in Lemma 2.

**Lemma 2.** [9] Assuming $BF(R_1)$, $BF(R_2)$ and $BF(R_1 \cup R_2)$ use the same size $m$ and $k$ hash functions, then

$$BF(R_1 \cup R_2) = BF(R_1) \cup BF(R_2)$$

We can actually extend Lemma 2 out to the following fact.

**Lemma 3.** Assuming $BF(R_1)$, $BF(R_2)$, ..., $BF(R_q)$ and $BF(R_1 \cup R_2 \ldots \cup R_q)$ use the same size $m$ and $k$ hash functions, then $BF(R_1 \cup R_2 \ldots \cup R_q) = BF(R_1) \cup BF(R_2) \ldots \cup BF(R_q)$.

The union of Bloom filters with the same size and set of hash functions is implemented by bit-wise OR.

In the pre-processing step, therefore, the jobtracker can collect all local Bloom filters from tasktrackers, merge the filters by using the bit-wise OR of two bit-arrays, and generate the global Bloom filter. This global filter is then intersected with another global filter to create the intersection filter $BF(R \cap S)$.

5. COST ANALYSIS FOR TWO-WAY JOIN

5.1 Cost Model

We adapt the cost model presented in [18] to suit the cost model of our implementation. Assume that $c_t$ is the cost of reading or writing data locally, $c_r$ is the cost of reading/writing data remotely, $c_i$ is the cost of transferring data from one node to another, $|D|$ is the size of the intermediate data of the join job, $|O|$ is the size of the output data, $t$ is the number of tasktrackers, and the size of the sort buffer is $B + 1$ pages.

Let $C_{pre}$ be the total cost to perform the pre-processing step, $C_{read}$ be the total cost to read the data, $C_{write}$ be the total cost to write the data, $C_{sort}$ be the total cost to perform the sorting and copying at the map and reduce nodes, and $C_p$ be the total cost to transfer intermediate data among the nodes. Accordingly, we get the total cost of the join job as follows:

$$C = C_{pre} + C_{read} + C_{sort} + C_p + C_{write}$$

where $C_{read} = c_r \cdot |R| + c_r \cdot |S|$; $C_{write} = c_r \cdot |O|$; $C_p = c_t \cdot |D|$;

$$C_{sort} = c_i \cdot |D| \cdot 2([\log_2 |D|] - \log_2 (mp_1 + mp_2)) + [\log_2 (mp_1 + mp_2)]$$ [18]

$$C_{pre} = C_{read} + 2 \cdot c_t \cdot m \cdot t + c_i \cdot m \cdot r \cdot t + a \quad \text{if the first approach, otherwise } a = 0.$$

In equation (6), an additional cost $C_{pre}$ should be added to the cost model in [18]. Intuitively, we can see that $|D|$, the size of the intermediate data, decides the total cost of the join operation. Thus, we should focus on analyzing this parameter for our different approaches in order to have more sufficient assessments.

---

1. Assume the filters are the same size $m$. The two parallel subjobs do not have a reduce phase and create every filter for each map output partition (1 partition). Then, the intersection filter is sent to each partition of tasktrackers for next job.
5.2 Cost Comparison of Approaches

In order to estimate $|D|$, it is assumed that $\hat{\epsilon}_R$ is the ratio of the joined records of $R$ with $S$, and $\hat{\epsilon}_S$ is the ratio of the joined records of $S$ with $R$. The size of intermediate data with the false intersection probability is $|D| =$

$$\begin{align*}
&\hat{\epsilon}_R|R| + f_{\cap BF}(S)\cdot(1 - \hat{\epsilon}_S)|R| + \hat{\epsilon}_S|S| + f_{\cap BF}(R)\cdot(1 - \hat{\epsilon}_R)|S| \quad (7) \\
&\hat{\epsilon}_R|R| + f_{\cap BF}(S)\cdot(1 - \hat{\epsilon}_S)|R| + \hat{\epsilon}_S|S| + f_{\cap BF}(R)\cdot(1 - \hat{\epsilon}_R)|S| \quad (8) \\
&\hat{\epsilon}_R|R| + f_{\cap BF}(S)\cdot(1 - \hat{\epsilon}_S)|R| + \hat{\epsilon}_S|S| + f_{\cap BF}(R)\cdot(1 - \hat{\epsilon}_R)|S| \\
&|R| + \hat{\epsilon}_S|S| + f_{\cap BF}(R)\cdot(1 - \hat{\epsilon}_R)|S| \quad (10) \\
&|R| + |S| \quad (11)
\end{align*}$$

where equation (7) for the pair of the filters (approach 1), equation (8) for the unpartitioned intersection filter (approach 2), equation (9) for the partitioned intersection filter (approach 3), and equation (11) for a filter $BF(R)$, and equation (10) in case without Bloom filter, and $f_{\cap BF}(S)$, $f_{\cap BF}(R)$, $f_{\cap BF}$ and $f_{\cap BF}$ refer to Section 3.4.

From the equation of the intermediate data size $|D|$ above, we can point out the following important evaluation.

**Theorem 5.** The join operation using the intersection filter is more efficient than using a basic Bloom filter because it produces less redundant and intermediate data than the latter. Additionally, we can drive comparing equation for $|D|$:

$$|D|_7 \approx |D|_9 < |D|_{10} < |D|_{11} \quad (12)$$

where $|D|_i$ is the intermediate data size for equation $i^{th}$ ($i = 7..11$).

**Proof.** We can see that querying tuples of $R$ into $BF(S)$ and tuples of $S$ into $BF(R)$ corresponds to finding common elements between the filters. It is also the intersection operation of filters as presented in Section 3.2. Thus the intermediate data generated by the first two approaches is equivalent $|D|_7 \equiv |D|_9$. (13)

From Theorem 4, we get $0 < f_{\cap BF} < f_{\cap BF} < |D|$. So we can deduce:

$$\hat{\epsilon}_R|S| + f_{\cap BF} \cdot (1 - \hat{\epsilon}_R)|S| < \hat{\epsilon}_S|S| + f_{\cap BF} \cdot (1 - \hat{\epsilon}_R)|S| < |S| \quad (14)$$

and

$$\hat{\epsilon}_R|R| + f_{\cap BF} \cdot (1 - \hat{\epsilon}_S)|R| < \hat{\epsilon}_S|R| + f_{\cap BF} \cdot (1 - \hat{\epsilon}_S)|R| < |R| \quad (15)$$

Combining inequalities (13), (14), and (15) into equations (8), (9), (10) and (11), Theorem 5 is proved $\Box$.

From equation (12) and (6), we can evaluate the total cost of the join operation for our approaches by $C_i \approx C_8 < C_9 < C_{10} < C_{11}$, where $C_i$ is the total cost in case of equation $i^{th}$ ($i = 7..11$).

It should be noted the total cost to perform pre-processing step

$$C_{pre} = \begin{cases} C_{read} + 2 \cdot c_1 \cdot m \cdot t + 2 \cdot c_1 \cdot m \cdot r \cdot t, & \text{in case of (7)} \\ C_{read} + 2 \cdot c_1 \cdot m \cdot t + c_1 \cdot m \cdot r \cdot t, & \text{in case of (8), (9), (10)} \\ 0, & \text{in case of (11)} \end{cases}$$

For the data locality optimization, the MapReduce framework runs the map task on a node where the input data resides in DFS and the data is directly fetched. Thus the read cost of this phase is low. As a result, the total cost $C_{pre}$ is negligible compared to the creation and transfer of redundant data over the network. However, the system will become inefficient if there is less redundant data or the number of tasktrackers $t$ and the number of reduce tasks $r$ is large. In the case of many tasks, a standard MapReduce job can be applied to replace heartbeat technique. We will solve this issue in our future work.

6. ADVANTAGE OF THE INTERSECTION FILTER FOR IMPORTANT JOIN CASES

In this section, we extend our approaches to support multiway join optimization. We present potential advantages of using the intersection filter in important special joins. Namely, we consider chain joins and star joins that are popular cases of a natural join. From these joins, we show a surprising simplification of the intersection filter in significantly reducing the number of tuples to be processed during a cascading join.

6.1 Chain Joins

A chain join is a cascading join of relations so that each relation is linked to the following one by a single or multiple attributes. This join case has the form of

$$R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4) \bowtie \ldots \bowtie R_n(x_n, x_{n+1})$$

and shown by Figure 9

![Figure 9. A chain join](image)

We begin with the implementation of a chain join using a cascade of Bloomjoins in MapReduce. It is an iterative implementation of two-way Bloomjoins as presented in Figure 10.

![Figure 10. Implementation of a chain join using Bloomjoins](image)
In the cascade of Bloomjoins, we can see that the dataset $R_1$ and intermediate join results $R_{1,2}, R_{1,2,3}, \ldots, R_{1,2,\ldots,n-1}$, are filtered by $BF(R_{1,2}, x_2), BF(R_{1,2,3}, x_3), BF(R_{1,2,4}, \ldots, BF(R_{1,2,\ldots,n-1}, x_{n-1})$ respectively. Meanwhile, the input datasets $R_2, R_3, \ldots, R_n$ are not filtered and thus there still remain large amounts of redundant data transferred over the network. This situation will be significantly improved by using the intersection filter as the following.

$$BF(R_{1,2,\ldots,n-1}, x_{n-1})$$

Figure 11. Implementation of a chain join using a cascade of Intersection Bloom joins

Figure 11 illustrates the implementation of a chain join as a sequence of two-way Intersection Bloom joins. All the input datasets and the intermediate join results are filtered by their corresponding intersection filters. For instance, the intersection filter $BF(R_{1,2}, x_2 \cap R_{1,2}, x_3)$ is used to eliminate most of non-joining data in both $R_{1,2}$ and $R_1$. Based on Theorem 5, it is easy to deduce that intermediate data sent to the Reducers in this case is less than in case of the Bloomjoin cascade.

We also discover two interesting solutions for optimizing a chain join using intersection filters as follows.

Figure 12. Optimization of a chain join with extended intersection filters

From the approach 1 mentioned in Section 3.1, we introduce an extended intersection filter (EBF) that includes an array of Bloom filters hashed on different join keys. Each tuple of a dataset may contain a few join keys linking to others. The tuple will be eliminated if at least one of its join keys, $x_i$, is not a member of a component filter $BF_i$ of the extended filter.

We use the extended intersection filter to optimize a chain join. In the first solution as Figure 12(a), instead a filtering operation is conducted after an intermediate join result is produced, we move this operation into a previous join job. Hence, the input datasets $R_2, \ldots, R_n$ are filtered by their corresponding extended filters. The extended filter $EBF_i$ is a pair of filters including a filter of the intermediate result $BF(R_{1,2,\ldots,i-1}, x_i)$ and a filter of a next relation $BF(R_{i+1}, x_{i+1})$. Unexpectedly, we now do not need any extra filtering operation for the intermediate join results. In other words, the intermediate results generated by two-way joins of a chain join only contain actual joining data and can be sent to the next Reduce phase without filtering. This is an important special characteristic while other solutions do not have.

As in Figure 12(b), the second solution suggests that a chain join includes a cascade of three-way joins using the extended filter. The input datasets $R_i$, where $i$ is an even number, are central datasets and are filtered by their corresponding extended filters. The extended filter $EBF_i$ includes a filter $BF(R_{1,\ldots,i-1}, x_i \cap R_{i, x_i})$ and a filter $BF(R_{i+1}, x_{i+1})$. Meanwhile, the input datasets $R_i$, where $i$ is an odd number, are filtered by intersection filters. Obviously, the size of the central dataset after filtering in this solution is less than Bloomjoin and the intersection Bloom join because the dataset has been filtered two times on two different join keys before it is sent to the Reducers.

It is noted that the solution (a) is designed to optimize intermediate join results without redundant data while the solution (b) is planned to reduce the number of intermediate join jobs. However, the later still uses the extra intersection filters to filter the intermediate results for next three-way joins.

6.2 Star Joins

We consider a star join including a set of joins in which a fact table (a large central table) is joined with several dimension tables (smaller tables containing descriptions for keys in the fact table). It has the form as shown in Figure 13.

The implementation of a star join using the extended intersection filter in MapReduce is suggested by Figure 14.
We build an extended intersection filter that is an array of $n$ filters $BF_i(R_i,x)$ and $n$ filters $BF(R_0,x)$, $i = 1...n$. The size of the extended filter is not too big because dimension tables are small. As shown in Figure 14, a star join is executed by joining all the datasets into one in which the large central dataset $R_0$ is filtered by the extended filter and the other datasets are filtered by filters $BF_i(R_i,x)$ respectively. Consequently, there is no redundant data when the datasets are sent to join processing. This implementation is more efficient than Bloom joins because the extended filter can eliminate non-joining tuples from the central dataset at the map phase of a job and it reduces the number of intermediate join jobs to zero.

7. CONCLUSION
We provide three approaches for building the intersection filter and show their feasibility in two-way joins and join cascades. It is proved that the join operation using the intersection filter is more efficient than other solutions since it significantly reduces redundant data, and thus produces much less intermediate data. Moreover, the intersection filter provides an extremely important characteristic for a join cascade in which intermediate join results generated from component joins only contain actual joining tuples without filtering. Although the intersection filter has false positives and an extra cost for the pre-processing step, its efficiency in space-saving and filtering often outweighs these drawbacks.

These are even more meaningful for our future work to implement general multiway joins, especially a cascade of map-side joins. Besides, experimental evaluation and a solution for a large number of the tasktrackers or the reduce tasks will be also considered.

8. REFERENCES