공학박사학위논문

Join Processing with Filtering Techniques on MapReduce Cluster

맵리듀스 클러스터에서 필터링 기법을 사용한 조인 처리

2014년 2월

서울대학교 대학원
전기·컴퓨터공학부
이 태 휘

Join Processing with Filtering Techniques on MapReduce Cluster

맵리듀스 클러스터에서 필터링 기법을 사용한 조인 처리

지도교수 김 형 주

위 논문을 공학박사 학위논문으로 제출함
2013년 11월

서울대학교 대학원
전기·컴퓨터공학부
이 태 휘

이태휘의 공학박사 학위논문을 인준함
2014년 1월

Ph.D. DISSERTATION

Join Processing with Filtering Techniques on MapReduce Cluster

FEBRUARY 2014

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

TAEWHI LEE

Abstract

Join Processing with Filtering Techniques on MapReduce Cluster

Taewhi Lee
Department of Electrical Engineering and Computer Science
College of Engineering
Seoul National University

The join operation is one of the essential operations for data analysis because it is necessary to join large datasets to analyze heterogeneous data collected from different sources. MapReduce is a very useful framework for large-scale data analysis, but it is not suitable for joining multiple datasets. This is because it may produce a large number of redundant intermediate results, irrespective of the size of the joined records. Several existing approaches have been employed to improve the join performance, but they can only be used in specific circumstances or they may require multiple MapReduce jobs. To alleviate this problem, MFR-Join is proposed in this dissertation, which is a general join framework for processing equi-joins with filtering techniques in MapReduce. MFR-Join filters out redundant intermediate records within a single MapReduce job by applying filters in the map phase. To achieve this, the MapReduce framework is modified in two ways. First, map tasks are scheduled according to the processing order of the input datasets. Second, filters are created dynamically with the join keys of the datasets in a distributed manner. Various filtering techniques that support specific desirable operations can be plugged into MFR-Join. If the performance of join processing with filters is worse than that without filters, adaptive join processing methods are also proposed. The filters can be applied according to their performance, which is estimated in terms of the false positive rate. Furthermore, two map task scheduling policies are also provided: synchronous and asynchronous scheduling. The concept of filtering techniques is extended to multi-way joins. Methods for filter applications are proposed for the two types of multi-way joins: common attribute joins and distinct attribute joins. The experimental results showed that the proposed approach outperformed existing join algorithms and reduced the size of intermediate results when small portions of input datasets were joined.

Keywords: *adaptive join, join filtering, join processing, MapReduce, multi-way join*

Student Number: 2004-21603

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Background and Motivation

We live in remarkable times, which are known as the era of “Big Data” [12]. Big Data is not clearly defined, but it is commonly characterized by three key properties(3Vs): volume, velocity, and variety [16]. A massive volume of data in various types and formats can be accumulated from various sources at an extremely high speed.

The analysis of large-scale data is playing an increasingly important role in business decision-making activities, because it can identify valuable information that is hidden in the data. For instance, Facebook collects tens to hundreds of terabytes of user log data every day, which it analyzes to provide a number of features, such as Facebook Insights for advertisers and friend recommendations [50]. At an Indian telecom company, billions of voice call data records are generated. These records are analyzed to enhance their portfolio of services [21].

In certain circumstances, it is necessary to analyze heterogeneous datasets that are collected from different sources. For example, Samsung Electronics produces a variety of devices, such as smartphones, tablet PCs, and televisions. Each device, or an application in-

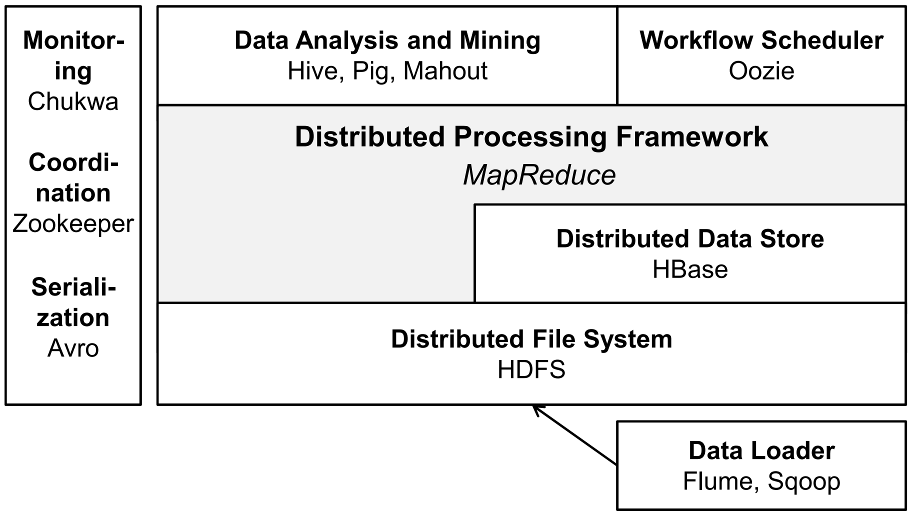


Figure 1.1: Hadoop ecosystem

stalled on it, accumulates different types of log data. These data have to be analyzed together to obtain business insights. Consequently, processing join operations between large heterogeneous datasets has become an important issue.

To process huge amounts of raw data, the MapReduce framework was developed by Google and it was made open to the public in 2004 in a paper titled, "MapReduce: Simplified Data Processing on Large Clusters" [13]. MapReduce facilitates the processing of tremendous amounts of data in a reasonable amount of time using a large cluster of commodity machines. This is achieved via a simple programming interface with *map* and *reduce* functions, so users can implement their jobs easily. For further convenience, MapReduce supports the automatic parallel and distributed processing of user programs with graceful failure handling. It is now being used more widely, since the emergence of Hadoop [2], which is an open-source implementation of the MapReduce framework. The utility of Hadoop-MapReduce is increasing because it can be used together with other components of

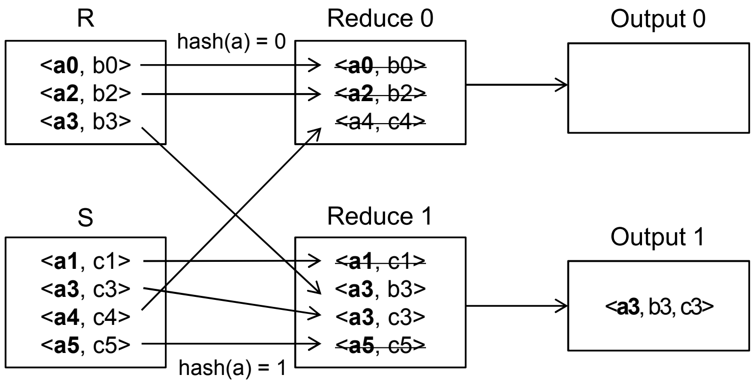


Figure 1.2: Basic join processing in MapReduce

the Hadoop ecosystem, which is shown in Figure 1.1. It becomes a de-facto standard for the Big Data management and business intelligence as its market is forecast to reach billions of dollars in five years, according to several market research reports [35, 36].

MapReduce is well suited to processing a single homogeneous dataset, but not for the performance of join operations on multiple heterogeneous datasets [10, 54]. To join multiple datasets in MapReduce, all of the input records have to be sent from map workers to reduce workers, regardless of the size of the joined records, as shown in Figure 1.2. This can produce a large number of redundant intermediate results that incur disk I/O costs for sort and merge, as well as network I/O costs for communication with other cluster nodes [18].

In the field of databases, many techniques have been developed over the past 30 years to address this problem [19]. However, auxiliary data structures are not available in MapReduce, such as indexes or filters, as it was initially designed to process a single, large dataset [13]. In this regard, some researchers have criticized MapReduce for ignoring rich technologies in database management systems, including efficient indexes and careful query execution planning [42]. It is not trivial to apply filters in MapReduce for the following two reasons. First, the processing order of input datasets cannot be controlled in the original MapReduce framework, because MapReduce schedules map tasks regardless of the dataset from which their corresponding input splits were obtained. Second, the filters should be constructed in a distributed manner because an input dataset is divided into multiple splits and distributed to all cluster nodes. Thus, it is necessary to design a filtering mechanism for MapReduce.

The problem of join processing in the MapReduce framework is addressed in this dissertation. The fundamental idea is to reduce the number of redundant intermediate results within a single MapReduce job, by exploiting filtering techniques. This dissertation focuses on the equi-join, which is used most widely. Details of specific contributions and an outline of this dissertation are presented in the following sections.

1.2 Contributions

The contributions of this dissertation can be divided roughly into three parts. First, a general join framework is proposed for applying filtering techniques. Next, the methods used for applying filters adaptively are presented according to their efficiency, because filters are not always beneficial for the join performance. Finally, the proposed method, which was originally developed for two-way joins, is extended to consider multi-way joins.

1.2.1 Join Processing with Filtering Techniques in MapReduce

MFR-Join: A General Join Framework with Filtering Techniques in MapReduce

The primary problem of join processing in MapReduce is that it generates large volumes of intermediate results, regardless of the number of final join results. To alleviate this problem, a general join method with filtering techniques called MFR-Join is proposed in this dissertation, which improves the join performance of the MapReduce framework by reducing the number of intermediate results. A working MFR-Join prototype was implemented in

Hadoop, and two design changes were produced. First, map tasks are assigned based on the order of the dataset. Second, filters are constructed in a distributed manner in the middle of the map phase. The proposed method was evaluated against several existing join algorithms with various sizes of TPC-H dataset using a commodity cluster. The results showed that the query execution time was improved significantly, especially when a small fraction of an input dataset is joined.

1.2.2 Adaptive Join Processing with Filtering Techniques in MFR-Join

Applying Filters Based on the False Positive Rates

The performance of join processing with filters is not always better and it can sometimes be worse than join processing without filters. To handle such cases, an adaptive join method with filtering techniques is proposed. In the adaptive join mode, MFR-Join estimates the performance of filters based on the false positive rates, and it disables filters with false positive rates greater than a user-configured threshold. An evaluation of the proposed method against the basic join algorithm without filters and the non-adaptive join with filters showed that the proposed method provided stable performance, which was similar to or better than that of the repartition join and the non-adaptive join.

Synchronous and Asynchronous Task Scheduling

MFR-Join needs to merge the filters for all tasktrackers because the input dataset is divided into multiple splits and distributed to all of the cluster nodes. Two map task scheduling policies are proposed in this dissertation: synchronous and asynchronous scheduling. In synchronous scheduling, the map tasks for the second input dataset are not assigned during the merging phase. The tasktrackers should wait for the merged filters, but every second input split can be processed with the filters, so more redundant intermediate results can be filtered out. In asynchronous scheduling, the map tasks are assigned continuously, although some tasks can be processed without the merged filters. Thus, the tasktrackers do not need to wait for the filters, although the size of the intermediate results may increase. There is a tradeoff between synchronous and asynchronous scheduling, which depends on the waiting time and the filter performance.

1.2.3 Multi-way Join Processing in MFR-Join

Applying Filters to Multi-way Joins

Joining multiple datasets in MapReduce may amplify the disk and network overheads because intermediate join results have to be written to the underlying distributed file system, or the map output records have to be replicated multiple times. A method for applying filters based on the processing order of input datasets is proposed in this dissertation, which is appropriate for the two types of multi-way joins: common attribute joins and distinct attribute joins. The number of redundant records filtered depends on the processing order. The input records do not need to be replicated in common attribute joins, so a set of filters is created, which are applied in turn. In distinct attribute joins, the input records have to be replicated, so multiple sets of filters need to be created, which depend on the number of join attributes.

1.3 Dissertation Overview

The remainder of this dissertation is organized as follows. In Chapter 2, the background and related work are presented. The MapReduce framework is explained first, which is the basis of this study. The existing join algorithms in MapReduce for two-way and multi-way joins are reviewed. Next, several filtering techniques are described, which can be applied to join processing.

MFR-Join is described in Chapter 3, which is a general join framework with filtering techniques in MapReduce. Its architecture and differences compared with the original MapReduce are explained. The processing cost is affected by the processing order of the two input datasets in MFR-Join. Thus, a cost model that helps to choose the processing order is also presented.

To overcome the shortcomings of join processing with filters in MapReduce, an adaptive join processing method that applies the filters according to their performance is presented in Chapter 4. Filters are applied only when their performance, which is estimated using a proposed method, is better than the user-configured threshold. The cost of adaptive join and the effects of the threshold on the cost are also discussed in this chapter.

MFR-Join is extended to multi-way joins in Chapter 5. This chapter explains the proposed methods for applying filters to both multi-way join cases: common attribute joins and distinct attribute joins. Finally, the conclusions and future research are described in Chapter 6.

Chapter 2

Preliminaries and Related Work

In this chapter, the basic concepts and the related work around this study are presented. MapReduce is described in Section 2.1, which is the basic framework of this study, and classic parallel and distributed join algorithms in databases are reviewed in Section 2.2. Then, two-way and multi-way join algorithms in MapReduce are reviewed in Section 2.3 and 2.4. Finally, several filtering techniques that can be used for join processing are mentioned in Section 2.5.

2.1 MapReduce

MapReduce [13] is Google's programming model for large-scale data processing run on a shared-nothing cluster. As the MapReduce framework provides automatic parallel execution on a large cluster of commodity machines, users can easily write their programs without the burden of implementing features for parallel and distributed processing.

A MapReduce program consists of two functions: `map` and `reduce`. The `map` function takes a set of records from input files as simple key/value pairs, and produces a set of intermediate key/value pairs. The values in these intermediate pairs are automatically grouped

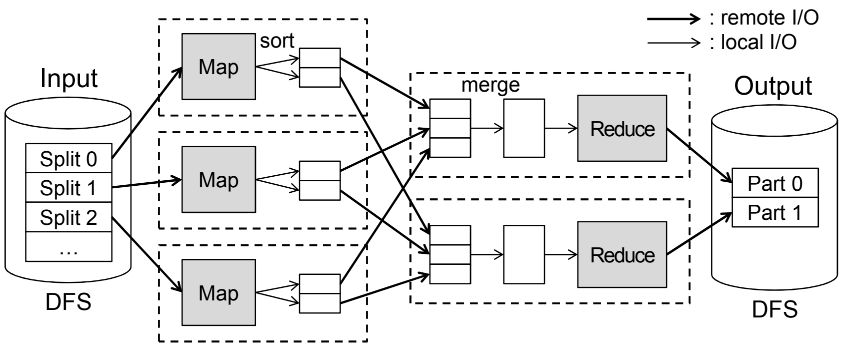


Figure 2.1: Execution overview of MapReduce

by key and passed to the `reduce` function. Sort and merge operations are involved in this grouping process. The `reduce` function takes an intermediate key and a set of values corresponding to the key, and then produces final output key/value pairs. An execution overview of MapReduce is shown in Figure 2.1.

A MapReduce cluster is composed of one master node and a number of worker nodes. The master periodically communicates with the workers using a heartbeat protocol to check their status and control their actions. When a MapReduce job is submitted, the master creates map and reduce tasks, and then assigns each task to idle workers. A map worker reads the input split and executes the `map` function specified by the user. A reduce worker reads the intermediate pairs from all map workers and executes the `reduce` function. When all tasks are complete, the MapReduce job is finished.

Hadoop [2] is a popular open-source implementation of the MapReduce framework. Although Google obtained a patent on MapReduce in early 2010 [14], it has been confirmed by Google that implementations of Hadoop are officially safe. In Hadoop, the master node is called the *jobtracker* and the worker node is called the *tasktracker*. Tasktrackers run one or more *mapper* and *reducer* processes, which execute map and reduce tasks respectively, according to the configuration. As the proposed methods have implemented into Hadoop, Hadoop terminology will be used in the remainder of this dissertation.

2.2 Parallel and Distributed Join Algorithms in DBMS

Since the join operation is one of the most important operation in database management systems, a lot of research has been conducted for efficient join processing over thirty years. Some closely related work is mentioned here. Consult the survey by Graefe [19] or the textbook by Silberschatz et al. [48] for more details.

Schneider and DeWitt presented parallel versions [45] of four join algorithms: the sort-merge join [5], the simple hash join [15], the Grace hash join [27], and the hybrid hash join [15]. Because the parallel join algorithms assume a shared-nothing environment, they can be naturally adapted to the MapReduce environment, which is also shared-nothing architecture. In effect, it can be said that the join algorithms in MapReduce, which are introduced in Section 2.3, derive from these parallel join algorithms. For example, the map-merge join [30] and the repartition join [10] in MapReduce are very similar to the parallel sort-merge join algorithm. In addition, the fragment and replicate join [17] in distributed databases influences the multi-way join algorithms in MapReduce, which are introduced in Section 2.4.

In distributed databases, it is important to reduce the network traffic among cluster nodes. Thus, researchers have attempted to reduce the size of redundant records that are transferred. Semijoin [9] is a classic distributed join algorithm. Suppose that we evaluate a join between relations $\mathsf{R(a,b)}$ at site S_1 and $\mathsf{S(a,c)}$ at site S_2. It uses a three-step process: First, it sends unique join attributes from a relation, say R at S_1, to S_2. Second, it computes joined records in the other dataset, say S at S_2, with the unique join keys from the first step, and sends them back to S_1. Third, it produces final join results at S_1 by computing the join between R and the joined records of S from the second step. The semijoin is efficient when few tuples contribute to the join because it does not send the tuples that are not joined. In distributed databases where relations are horizontally partitioned, methods for optimizing the semijoin were presented by Shasha and Wang [47] as well as Segev [46]. The semijoin has been also adapted to the MapReduce environment by Blanas et al. [10].

Bloomjoin [34] is a distributed join algorithm to filter out tuples that are not matched by a join using Bloom filters [11]. Consider the same join example as in the semijoin. The bloomjoin algorithm generates a Bloom filter with the join keys of a relation, say R. Then, it sends the filter to S_2, where S resides. At S_2, it scans S and sends only the tuples with the join keys that are set in the received filter to S_1. Finally, it joins R and the filtered S at S_1. This idea has been extended to multi-way joins by Kemper et al. [26]

These studies were conducted in distributed databases, which provide low scalability compared to large-scale data processing systems. The methods proposed in this dissertation not only extend the concept of filtering techniques to the large-scale data processing systems by merging filters globally, but also provide several features such as adaptive joins based on filter performance and application of various filtering techniques.

2.3 Join Algorithms in MapReduce

Join algorithms in MapReduce are roughly classified into two categories: map-side joins and reduce-side joins [30]. Map-side joins produce final join results in the map phase, and do not use the reduce phase. Because they do not need to pass intermediate results from map workers to reduce workers, map-side joins are more efficient than reduce-side joins; however, they can only be used in particular circumstances. Reduce-side joins can be used

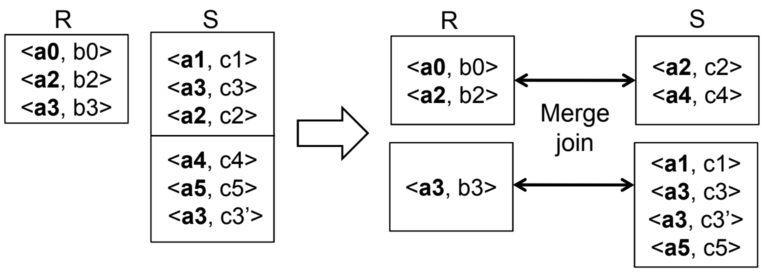


Figure 2.2: Map-merge join

in more general cases, but they are inefficient because large intermediate records are sent from map workers to reduce workers. This work is an attempt to provide both generality and performance by improving reduce-side joins.

2.3.1 Map-side joins

Hadoop's map-side join [53], called the map-merge join [30], merges input datasets that are partitioned and sorted on the join keys in the same way, similar to merge join in traditional DBMS. That is, each input dataset must be divided into the same number of partitions and must be sorted by the same key. All the records for a particular key must reside in the same partition. Although this is a little strict requirement, it may be useful to join the outputs from other MapReduce jobs that had the same number of reducers and the same keys. The map-merge join can be implemented in one map-only job. Each mapper reads the same partition of the input datasets and executes the merge join. Hadoop provides API for the map-merge join and users can specify the paths of input partitions. However, it requires an additional MapReduce job if the input datasets are not partitioned and sorted in advance, as shown in Figure 2.2.

The broadcast join [10] distributes the smaller one of the input datasets to all map workers, and performs the join in the map phase. The broadcast join can be implemented in one

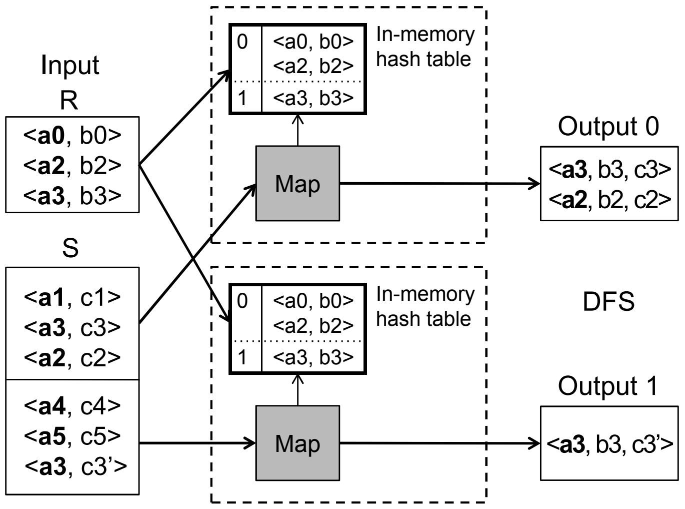


Figure 2.3: Broadcast join

Map-only job. Consider the example in Figure 2.3. Suppose that an input dataset R is much smaller than the other input dataset S. First, in the init phase before the map phase, each map task reads the smaller dataset R and builds an in-memory hash table using the join key. Then, it reads a split of S and probe the hash table with the join key and produce final join results, like the hash join in RDBMS. These results are written to distributed file system, and not sent to reducers. Since the broadcast join is run in a map-only job, it does not produce intermediate results, and avoids the network overhead in reduce-side joins. However, it is efficient only if the size of an input dataset is small. Another difference is that the final results for a particular key may be spread across multiple mappers because these results are not grouped in reducers.

2.3.2 Reduce-side joins

The repartition join [10] is the most common join algorithm in MapReduce. It works similar to the parallel sort-merge join in DBMS, and can be implemented in one MapReduce job.

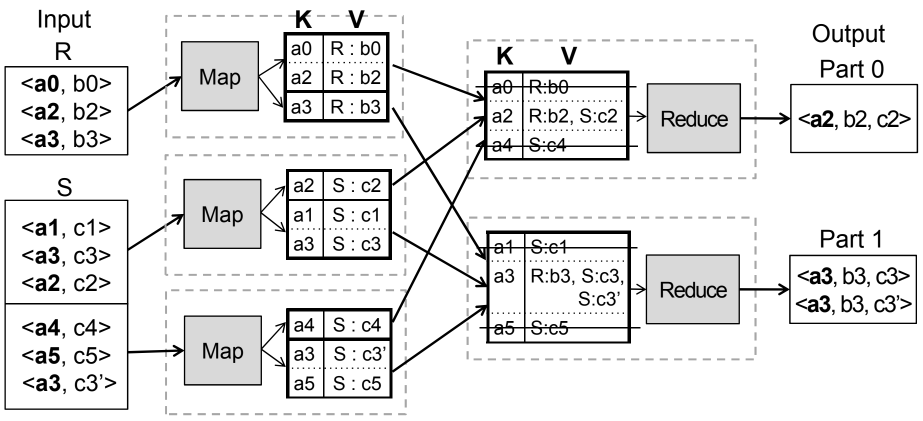


Figure 2.4: Repartition join

Figure 2.4 illustrates an example of the repartition join between the datasets R and S. Each mapper reads a split of R or S, and outputs the join attribute as the key, and its dataset id and other attributes as the value. The dataset identifier is tagged to identify where the record is from. The map outputs then are partitioned, sorted, and merged by the MapReduce framework. Next, reducers do the join for each grouped records. The reducers buffer these values into two sets, the records from R and the records from S, by the dataset identifier, and then produce final join results. The repartition join is most common and general, but it has a drawback. It is that all of the input records have to be sent to reducers, including unnecessary records, which are marked with strikethroughs in Figure 2.4. These records are not joined, so mappers do not have to send these records to reducers. This may lead to a performance bottleneck.

The semijoin in MapReduce [10] works in a similar manner to the semijoin in traditional DBMS. It runs each step of the three-step process in the semijoin in an independent MapReduce job. Figure 2.5 illustrates each job in the MapReduce version of the semijoin. Join input datasets are the same in the earlier examples. The first job extracts unique join

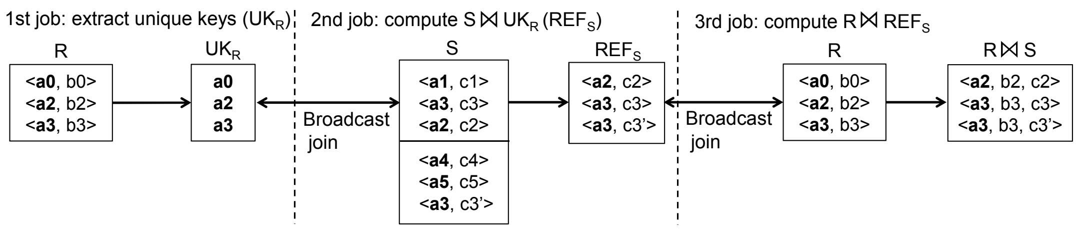

1st job: extract unique keys (UK_R)
2nd job: compute S ⋈ UK_R (REF_S)
3rd job: compute R ⋈ REF_S
R
UK_R
S
REF_S
R
R ⋈ S
<a0, b0> <a2, b2> <a3, b3>
a0 a2 a3
<a1, c1> <a3, c3> <a2, c2>
<a4, c4> <a5, c5> <a3, c3'>
<a2, c2> <a3, c3> <a3, c3'>
<a0, b0> <a2, b2> <a3, b3>
<a2, b2, c2> <a3, b3, c3> <a3, b3, c3'>
Broadcast join
Broadcast join

Figure 2.5: Semijoin in MapReduce

keys from an input dataset. The second job finds joined records in the other dataset with the unique join keys from the first step. Finally, the third job produces final join results by performing the join between R and the joined records of S from the second step. The semijoin in MapReduce may reduce the size of the intermediate results by filtering out the unreferenced records with unique join keys. Therefore, it is efficient when a small portion of records participate in joins. However, the semijoin requires three MapReduce jobs, which means that the results of each job are written and read in the next job. This incurs additional I/O overheads.

Recent studies have attempted to adapt the bloomjoin to the MapReduce framework. Reduce-side joins with a Bloom filter were proposed previously [40, 56, 57, 33], but they create the filter via an independent job. Therefore, they have to process the input datasets multiple times. Koutris [28] theoretically investigated join methods using Bloom filters within a single MapReduce job, but did not provide specific technical details. This dissertation proposes a general join framework with various filtering techniques in a single MapReduce job. Furthermore, while all of the studies apply filters without regard to their performance, the proposed methods in this dissertation apply filters adaptively for stable performance according to their false positive rates.

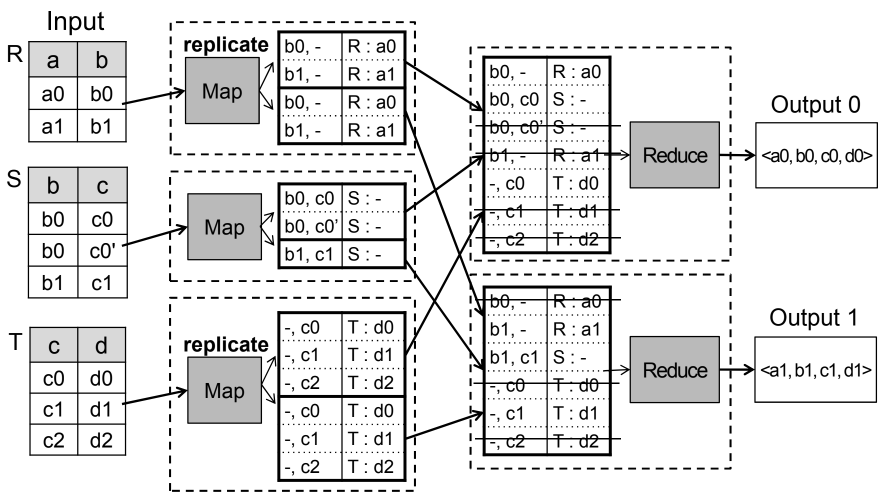


Figure 2.6: Basic multi-way join processing in MapReduce

Map-Reduce-Merge [54] adds a merge phase after the reduce phase to support operations on multiple heterogeneous datasets conveniently. However, it cannot reduce the size of the intermediate results.

2.4 Multi-way Joins in MapReduce

To join several datasets, two-way joins, which are explained in Section 2.3, have to be performed multiple times. Otherwise, they can be joined simultaneously in a single MapReduce job by replicating some input datasets. Figure 2.6 shows an example of basic multi-way join processing in MapReduce. In the example, three input datasets R(a,b), S(b,c), and T(c,d) are joined with two attributes b and c. To join the three datasets simultaneously, some datasets are fully replicated to reducers, i.e., R and T in this example. The records that are joined then are gathered in a certain reducer. Replication may degrade the join performance, so it is important to reduce the number of redundant records, which are marked with strikethroughs in Figure 2.6.

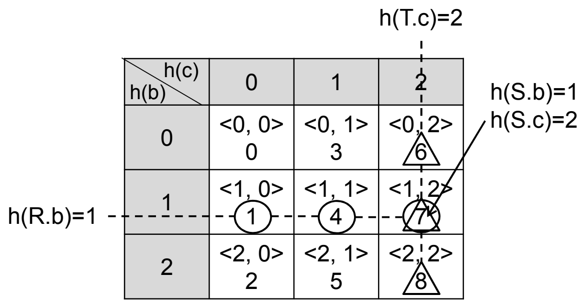


Figure 2.7: Fragment-replicate joins in MapReduce

There have been some attempts to optimize the number of input records replicated in the multi-way joins [6, 7, 25]. They use similar methods for minimizing the number. Figure 2.7 shows a partial replication of the input records for a join example with three datasets and nine reducers. A cell in the figure represents a reducer. Unlike the naive multi-way join that replicates some input datasets fully, the input records of R and T are replicated by only three reducers, depending on the hash values of the join attributes b and c.

This optimization problem can be formulated as a problem of minimizing the total number of records that are sent to reducers. For example, the cyclic join

$$\mathsf{R(a,b)} \bowtie \mathsf{S(b,c)} \bowtie \mathsf{T(a,c)}$$

can be formulated as follows:

$$\text{Minimize } rc + sa + tb$$

$$\text{subject to } abc = k$$

where a, b, and c are the replicate factor for the join attributes a, b, and c, respectively. That is, input records have to be replicated as the replicate factor if the dataset do not have the corresponding join attribute.

Afrati and Ullman [6, 7] solved the minimization problem using a method based on Lagrangean multipliers to find the optimal solution. Jiang et al. used a heuristic approach to find an approximate solution [25]. These studies can be used in combination with the approach proposed in this dissertation, which uses filtering techniques to facilitate more efficient multi-way join processing.

This idea can be similarly used for theta-joins. Okcan et al. proposes the 1-Bucket-Theta algorithm that replicates input records according to join conditions [39]. However, the detailed discussion for theta-joins is beyond the scope of this dissertation.

2.5 Filtering Techniques for Join Processing

Various filtering techniques for approximate membership matching can be used for equi-joins. For join processing over large datasets, the following requirements have to be satisfied: (1) It must not yield false negatives. (2) It is space-efficient regardless of the number of inserted elements. Representative examples are described in the following subsections.

A Bloom filter [11] is a probabilistic data structure used to test whether an element is a member of a set. It consists of an array of *m* bits and *k* independent hash functions. All bits in the array are initially set to zero. When an element is inserted into the array, the element is hashed *k* times with *k* hash functions, and the positions in the array corresponding to the hash values are set to one. Then, the membership status of an element can be tested by evaluating this array. If all bits of the element's *k* hash positions are one, It can be

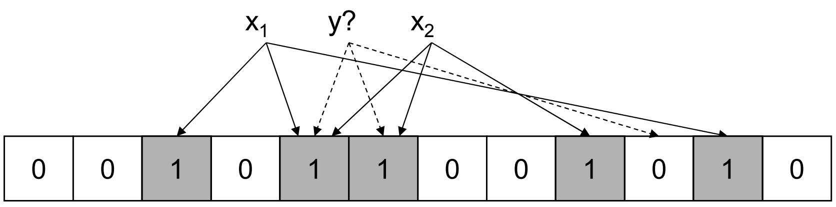


Figure 2.8: Bloom filter

concluded that the element is in the set. Figure 2.8 shows an example where $m = 12$, $k = 3$. Two elements x_1 and x_2 have been inserted using three hash functions. When an element, y, is looked up, it is also hashed using the three hash functions. Because one of the bits of its hash positions are zero, it is concluded that y is not in the set.

The Bloom filter has been used for efficient join processing [34, 26], as described in Section 2.2. This is because it may yield false positives, but it does not produce false negatives. Furthermore, its size is fixed regardless of the number of elements. The probability of a false positive after inserting n elements can be calculated as follows [11]:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{\frac{kn}{m}}\right)^{k} \tag{2.1}$$

There is a tradeoff between m and the false positive probability p.

Many Bloom filter variants exist such as one memory access Bloom filter [43], Bloom filter with partitioned hashing [22]. They can be also used for join processing as long as they satisfy both requirements. See the survey by Tarkoma et al. [49] for more details.

The quotient filter [8] is an alternative to the Bloom filter. Although it is designed to reduce the number of contiguous accesses when it is resident on SSD, it shows comparable performance to the Bloom filter in RAM. This technique stores a p-bit fingerprint for each element. A fingerprint f is split into two parts: $q = p - r$ most significant bits (the quotient) and the remaining r bits (the remainder). The quotient filter consists of an array of 2^q buckets of $(r+3)$ bits. Each bucket stores an r-bit remainder and three metadata bits.

The quotient filter inserts an element in a similar way to open hashing. Its remainder is stored in the slot corresponding to its quotient if the slot is empty. Otherwise, the remainder is stored in the next empty slot from the corresponding slot. Some remainders that are stored may be shifted during an insert operation. For later lookup, three metadata bits are maintained for each slot. See the paper by Bender et al. [8] for the detailed algorithms of insert and lookup operations. The quotient filter has the probability of a false positive after inserting n elements as follows:

$$p = \left(1 - \left(1 - \frac{1}{2^{(q+r)}}\right)^n\right) \tag{2.2}$$

The interval filter [44] uses an array of m bits, and its lower bound lb and upper bound ub should be set. Without loss of generality, suppose that the elements to be inserted into the filter are integers. Although the intervals can be dynamically created and merged, assume that the range [lb, ub] is statically split into m intervals here. Each interval then has a length itv of $(ub - lb)/m$. Each bit in the array represents an interval, and the interval bit for an element with value v is the bit of the position $(v - lb)/itv$. It inserts an element by setting the value of its interval bit to one, and checks whether an element is in the filter by checking its interval bit. The interval filter has the probability of a false positive after inserting n elements as follows:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^n\right) \tag{2.3}$$

Chapter 3

MFR-Join: A General Join Framework with Filtering Techniques in MapReduce

As mentioned earlier, MapReduce [13] is a very useful framework for large-scale data analysis because it facilitates the processing of tremendous amounts of data in a reasonable amount of time using a large cluster of commodity machines. Unfortunately, MapReduce has some limitations to performing a join operation on multiple datasets, one of the essential operations for practical data analysis [10, 54]. The primary problem of join processing in MapReduce is that it emits large intermediate results, regardless of the number of final join results. This could cause an unnecessary network overhead for sending the intermediate results to other cluster nodes, and a disk I/O cost for sorting and merging them, even when only a small fraction of input data participate in the join. To apply these techniques, the modified MapReduce versions [10, 40, 28, 57] of semijoin [9] and bloomjoin [34] have been proposed, but they require multiple MapReduce jobs to process the input data multiple times.

To resolve these issues, a general join framework that utilizes various filtering techniques within a single MapReduce job, which is named MFR-Join, is proposed in this chapter. Join operations for two datasets are first considered. Multi-way joins will be described in Chapter 5. MFR-Join creates filters on one input dataset, and filters out redundant records in the other dataset by applying the filters in the map phase. In this way, the communication cost for redundant records can be reduced by processing the input datasets only once.

The MapReduce framework is modified to implement this idea as follows. First, the map task scheduling is changed so that input datasets are processed sequentially in the map phase. Second, the execution flow is re-designed to construct filters dynamically within a single MapReduce job. Locally created filters for an input dataset are sent to, and merged at, a master node. These features have been implemented on the top of Hadoop [2], an open-source implementation of the MapReduce framework. In MFR-Join, the processing cost is affected by the processing order of the two input datasets. Therefore, a cost model to help choose the processing order is presented.

3.1 MFR-Join Framework

This section describes the overall architecture of MFR-Join framework and the major changes that have been made.

3.1.1 Execution Overview

Figure 3.1 shows the overall execution flow of a join operation on datasets R and S in our framework. Suppose that R is chosen to be processed first; that is, the filters are built on R. The term *build input* will be used for the input dataset processed first, and *probe input* for the other dataset. When a user program is submitted, the following sequence of actions is

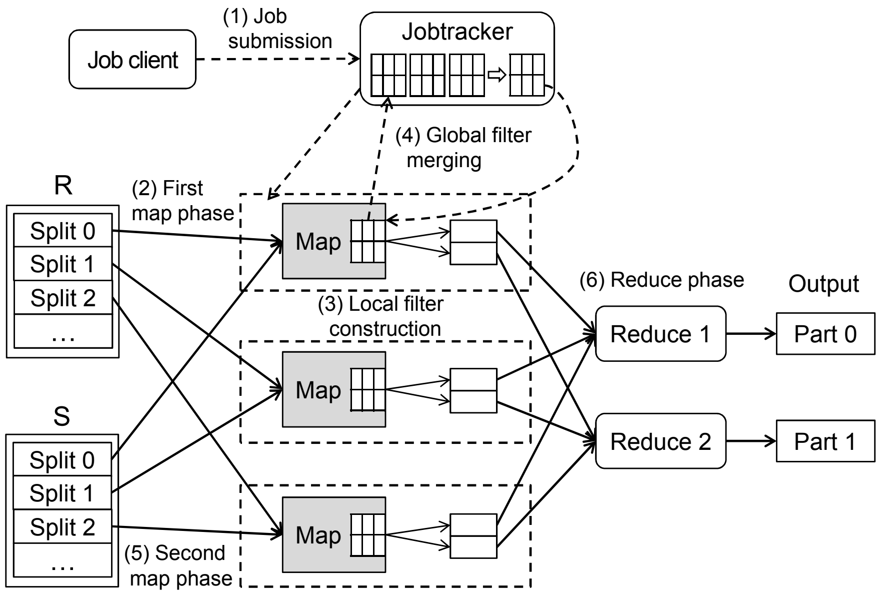


Figure 3.1: Execution overview

performed.

1. **Job submission.** On submission of a MapReduce job, map and reduce tasks are created. Assume that m_1 map tasks for R, m_2 map tasks for S, and r reduce tasks are created. A task includes all the necessary information, such as job configuration and location of the corresponding input/output files, for it to run on a tasktracker. The job configuration includes additional filter information such as parameter types and values for the filter to use.

2. **First map phase.** The jobtracker assigns the m_1 map tasks for R or the reduce tasks to those tasktrackers that have idle mappers or reducers. A mapper reads the input split for the task, converts it to key/value pairs, and then executes the map function for the input pairs.

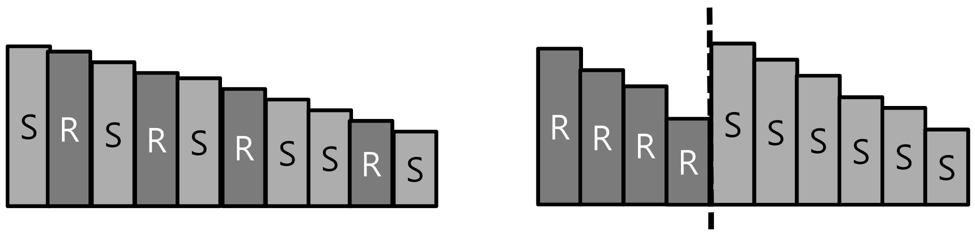
3. **Local filter construction.** The intermediate pairs produced from mappers are divided into r partitions, which are sent to reducers. For each partition, a specified type of filter is created by inserting the keys of its intermediate pairs. These filters are called *local filters*, because they are built for only the intermediate results in a single tasktracker. The tasktracker merges the individual filters from each map task and maintains only r filters.

4. **Global filter merging.** When all m_1 map tasks for the first input dataset R have been completed, the jobtracker stops assigning map tasks and requests that all tasktrackers send it their local filters via heartbeat responses. The jobtracker then merges all the local filters to construct the *global filters* for R, which contain all the join keys processed in all the tasktrackers.

5. **Second map phase.** The jobtracker assigns the m_2 map tasks for S or the remaining reduce tasks to the tasktrackers. Mappers run the assigned tasks with the received global filters, and intermediate pairs with keys that are not contained in the global filters are filtered out.

6. **Reduce phase.** This step is the same as the reduce phase in Hadoop. Reducers read the corresponding intermediate pairs from all mappers using remote procedure calls. Each reducer then sorts the intermediate pairs and runs the reduce function. Final output results are then written to the given output path.

Two major modifications have been made to the design of Hadoop. First, map tasks are scheduled according to the order of the dataset. Second, filters are constructed on the build input in a distributed manner to filter out the probe input. The following subsections describe more details on these points.



(a) Hadoop (b) MFR-Join

Figure 3.2: Map task scheduling

3.1.2 Map Task Scheduling

Hadoop basically assigns map tasks based on the order of the input split size, considering the locality of the input split, as shown in Figure 3.2(a). Consequently, map tasks on different input datasets are intermingled by the task scheduler. MFR-Join assigns map tasks according to a certain order of the input datasets, as shown in Figure 3.2(b). This gives us the opportunity to apply database techniques such as tuple filtering and join ordering. The order can be determined based on the estimated cost, as described in Section 3.2. Within a single input dataset, map tasks are assigned as in the original Hadoop.

In order to schedule map tasks by dataset, each map task needs to know the dataset identifier of the corresponding input split. Accordingly, new input format and input split classes (`DataSetInputFormat` and `DataSetSplit`) have been implemented to contain this information. They are extended versions of the respective Hadoop `FileInputFormat` and `FileSplit` classes. In addition, `JobQueueTaskScheduler`, which is the default task scheduler of Hadoop, have been modified to schedule map tasks using the dataset identifier.

3.1.3 Filter Construction

While a tasktracker runs a map task for the build input, MFR-Join creates filters on the intermediate records produced from the task. The filter is created for each map output partition, which is assigned for each reduce task. Therefore, the total number of the filters is the number of reduce tasks. When multiple map tasks are run, each tasktracker merges its filters, so that only one set of filters is maintained. This set is called the local filters.

When all map tasks for the build input are complete, the jobtracker must gather all local filters to construct the global filters. In Hadoop, the jobtracker controls the tasktrackers by putting some instructions, called `TaskTrackerAction`, in heartbeat messages. For the merging process, two `TaskTrackerAction` classes, called `SendLocalFilterAction` and `ReceiveGlobalFilterAction`, have been added. The jobtracker sends the `SendLocalFilterAction` as the heartbeat response to all tasktrackers, and they send the jobtracker their local filters. The jobtracker merges all the local filters to build the global filters using bitwise OR operations, and sends the `ReceiveGlobalFilterAction` with the global filters in the heartbeat response to all tasktrackers.

Table 3.1: Filter merging time

Number of tasktrackers	3	5	7	10
Elapsed time	22.137	26.735	25.012	28.887

If the number of reduce tasks or tasktrackers is large, the global filter construction is expected to take a long time. Table 3.1 shows the time elapsed for filter merging in the experimental environment described in Section 3.3.1, varying the number of tasktrackers. The time for merging filters tends to increase as the number of nodes becomes larger, although it depends on the difference between the time that the first node finishes its assigned map tasks for the build input and the time that the last node finishes its corresponding map tasks. This overhead could be distributed by merging local filters hierarchically, although this has not yet been implemented—this issue is left for future work.

3.1.4 Filtering Techniques Applicable to MFR-Join

Efficient filtering techniques for joins depend on the distribution of the join keys and the number of records joined. For this reason, MFR-Join has been designed to apply various types of filters. Any filtering techniques can be plugged into MFR-Join as long as they support the following operations:

- `insert(key)`: insertion of the specified `key` into the filter.
- `contains(key)`: returning whether it contains the specified `key` or not.
- `merge(filter)`: merge with another `filter` of the same type.

The `insert` and `contains` operations are the basic operations for testing (approximate) membership. MFR-Join additionally require the `merge` operation for the filters.

Some filtering techniques, such as Bloom filter [11] and its variants [49], the interval filter [44], and the quotient filter [8], support the operation. The Bloom filter and the interval filter can be merged by a bitwise OR operation, and the quotient filter can be merged in a similar way to merge sort in DBMS. The details of the example filters is described in Section 2.5.

3.1.5 API and Parameters

Hadoop provides a library class called `MultipleInputs` to support MapReduce jobs that have multiple input paths with a different `InputFormat` and `Mapper` for each path. This

Table 3.2: User parameters for MFR-Join

Parameter	Description	Type	Default value
mapred.filter.use	whether to use filters for join processing	boolean	true
mapred.filter.class	class name of the filter	String	org.apache.hadoop.mapred. lib.MapBloomFilter
mapred.filter.param.types	parameter types for the filter class	String	int,int
mapred.filter.param.values	parameter values for the filter class	String	4194304,2

class is convenient, as it allows users to specify which jobs should perform the join operation. MFR-Join provides a similar library class called `JoinInputs`. Users can specify jobs to be joined with filters using the following API.

```
JoinInputs.addInputPath(conf, path, dsid, inputformat, mapper)
```

Compared to `MultipleInputs`, it requires `dsid` to specify the dataset identifier, and takes a subclass of `DataSetInputFormat` as the `inputformat` argument, described in 3.1.2.

Several parameters have been added to configure MFR-Join, as shown in Table 3.2. Users can define these in Hadoop configuration files, or specify them as runtime parameters.

3.2 Cost Analysis

This section presents the cost model for MFR-Join by adjusting Herodotou's Hadoop performance model [23], which describes in detail the execution of a MapReduce job on Hadoop, including both I/O and CPU costs. Becauuse it does not include features for filtering, a slight adjustment is needed. Using this cost model, the query optimizer can choose the strategy

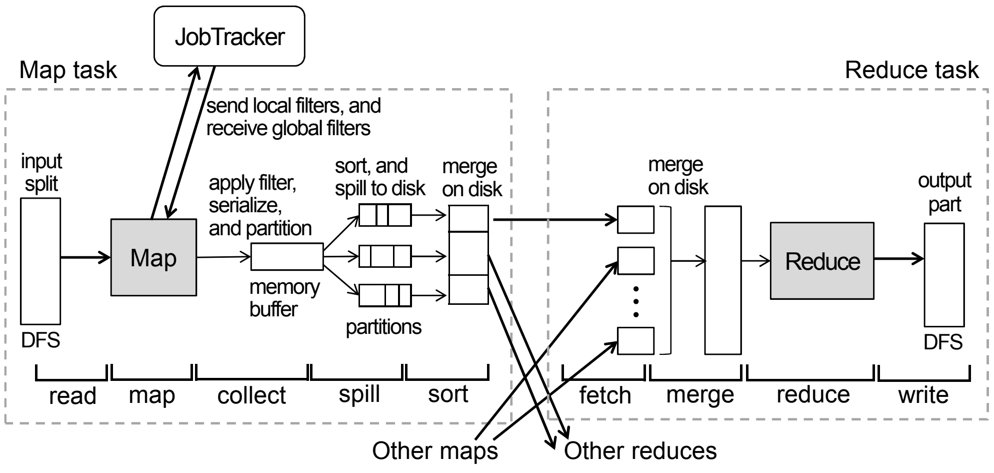


Figure 3.3: Shuffle phase in MapReduce

that minimizes the total cost, including the processing order of input datasets, whether to use filters.

3.2.1 Cost Model

Since the changes that have made in MFR-Join affect the cost of dealing with intermediate records, we need to know more about the *shuffle phase*, in which Hadoop sorts and transfers the intermediate records. Note that the term *shuffle phase* is used for the whole process, from the point where map tasks produce intermediate records to the point where reduce tasks consume them, as in White's book [53].

Figure 3.3 shows the process of execution of a map and reduce task in MFR-Join, and illustrates the shuffle phase in detail. This process is virtually the same as that in the original Hadoop, except for the following two points: (1) Filter application operations are also carried out in the collect stage in map tasks. (2) Filters are merged locally or globally after the map task for the first input dataset is complete. This division cannot include the filter merge processes because it is from the point of view of a single map task. However, it is included when the execution of all the map tasks is considered.

Let T_{job} be the total cost of a MapReduce job, T_{map} be the cost of all map tasks for the job, T_{filter} be the cost of merging filters locally and globally, and T_{reduce} be the cost of all reduce tasks for the job. Consequently, the total cost of a MapReduce job T_{job} can be expressed as $T_{map} + T_{filter} + T_{reduce}$. When execution costs in MFR-Join and the original Hadoop are different from each other, The superscripts f and h is used to denote join processing with filters and without filters in the original Hadoop, respectively. Accordingly, the total cost of a MapReduce job in both of them can be expressed as follows:

$$\begin{aligned} T^f_{job} &= T^f_{map} + T_{filter} + T^f_{reduce} \\ T^h_{job} &= T^h_{map} + T^h_{reduce} \end{aligned} \tag{3.1}$$

Each of the costs will be examined in the following subsections. Table 3.3 shows the parameters used in the cost analysis. Consider a MapReduce job for a join on datasets R and S, assuming the information in Table 3.3 is given. For simplicity, this dissertation assumes that the combiner and compression features are not used. Since Herodotou's cost model [23] considers the features, the proposed cost model can be extended as in the original.

Map Task

The execution of a map task can be divided into five stages, as shown in Figure 3.3. The total cost of all map tasks T_{map} can be computed as the sum of the costs during the five stages.

$$T_{map} = C_{read} + C_{map} + C_{collect} + C_{spill} + C_{sort} \tag{3.2}$$

Read and Map stages. Each map task reads the corresponding input split and converts each record to a key/value pair. It then executes the map function and produces intermediate

Table 3.3: Cost parameters

Parameter	Definition
b_r, b_s	Size of input datasets r and s, respectively, in bytes
l_r, l_s	Length of a record in input datasets r and s, respectively
n_r, n_s	Number of records in input datasets r and s, respectively (i.e., $n_r = b_r/l_r$, $n_s = b_s/l_s$)
b_f	Size of a filter in bytes
c_{r_dfs}	I/O cost of reading from distributed file system per byte
c_{w_dfs}	I/O cost of writing to distributed file system per byte
c_{r_loc}	I/O cost of reading from local disk per byte
c_{w_loc}	I/O cost of writing to local disk per byte
c_{tr}	Network cost of transferring data per byte
c_{c_map}	CPU cost of executing map function per record
c_{c_red}	CPU cost of executing reduce function per record
c_{c_part}	CPU cost of partitioning and (de)serializing per record
c_{c_sort}	CPU cost of sorting per record
c_{c_merge}	CPU cost of merging sorted data per record
c_{c_fltr}	CPU cost of inserting an element into filters or checking that an element is in the filters per record
c_{c_union}	CPU cost of merging a filter
$\#_{nodes}$	Number of tasktracker nodes
$\#_{map_r}$	Number of map tasks for input dataset r
$\#_{map_s}$	Number of map tasks for input dataset s
$\#_{map}$	Number of map tasks for a job (i.e., $\#_{map} = \#_{map_r} + \#_{map_s}$)
$\#_{reduce}$	Number of reduce tasks for a job
$\#_{map/node}$	Maximum number of map tasks that can be simultaneously run by a tasktracker

map output records.

The costs during read and map stages are the same in both join processing with and without filters, and can be computed as follows:

$$\begin{aligned} C_{read} &= (b_r + b_s) \cdot c_{r\_dfs} \\ C_{map} &= (n_r + n_s) \cdot c_{c\_map} \end{aligned} \tag{3.3}$$

The generated records may include redundant intermediate records that did not participate in the join.

Collect stage. All the records are partitioned and processed with an insert or contains operation for filtering. In MFR-Join, the map output records from the first input dataset, say R, are inserted into local filters; or, those from the second input dataset, say S, are checked to determine whether they are in the global filters. The map output records that are not filtered out are partitioned and collected into a memory buffer.

The costs of the collect stage in join processing with and without filters are

$$\begin{aligned} C^{f}_{collect} &= (n_r + n_s) \cdot (c_{c\_part} + c_{c\_fltr}) \\ C^{h}_{collect} &= (n_r + n_s) \cdot c_{c\_part} \end{aligned} \tag{3.4}$$

It is clear that the additional CPU cost of executing filter operations results from applying the filters.

The number of intermediate map output records n_{inter} can be computed as follows:

$$\begin{aligned} n^{f}_{inter} &= n_r + n_s \cdot \sigma_{s\_r} + n_s \cdot (1 - \sigma_{s\_r}) \cdot p \\ n^{h}_{inter} &= n_r + n_s \end{aligned} \tag{3.5}$$

where σ_{s_r} is the ratio of the joined records of S with R, and p is the false positive rate of the global filters. n_r signifies the number of first input dataset records that are not filtered out and used to create filters. $n_s \cdot \sigma_{s\_r}$ signifies the number of second input dataset records that are joined, and $n_s \cdot (1 - \sigma_{s\_r}) \cdot$ signifies the number of second input dataset records that are not joined but passed to reducers as false positives. Without filters, n_{inter} is equal to $n_r + n_s$ because input records are not filtered out at all.

Spill stage. When the data size in the buffer reaches a given threshold, the data partitions are sorted and written to local disk. The cost of the spill stage depends on the number of spills, and is determined according to the size of the spill buffer. The buffer size is configured via Hadoop parameters such as `io.sort.mb`, `io.sort.spill.percent`, and `io.sort.record.percent`. Let b_{buf} be the buffer size in bytes. Further, let the length of an intermediate record l_{inter} be $(l_r + l_s)$, assuming both the records from R and S are joined without projection. Accordingly, the number of records that can be included in buffer n_{buf} can be expressed as b_{buf}/l_{inter}. Thus, the number of spills that are performed in all map tasks $\#_{spill}$ can be estimated as follows:

$$\#_{spill} = \left\lceil \frac{n_{inter} \cdot l_{inter}}{b_{buf}} \right\rceil = \left\lceil \frac{n_{inter}}{n_{buf}} \right\rceil \tag{3.6}$$

In practice, the number of spills will be slightly larger than the estimated value because map tasks may spill any partition whose size is smaller than the unit size.

With the estimate, the cost of the spill stage is computed as follows:

$$C_{spill} = \#_{spill} \cdot (n_{buf} \cdot \log_2 (\frac{n_{buf}}{\#_{reduce}}) \cdot c_{c\_sort} + b_{buf} \cdot c_{w\_loc}) \tag{3.7}$$

Although Equation 3.7 can be used to compute the costs of the spill stage in both join processing with and without filters, the costs may vary because the numbers of intermediate records n_{inter} in them may be different.

Sort stage. In the sort stage, the spilled partitions are merged into a single file. This stage performs an external merge sort similar to the merge stage in a reduce task. The merging process may be performed in multiple merge passes according to the number of spills. The total number of merge passes depends on the number of spills and the number of spills to merge at once, which is configured by `io.sort.factor` and denoted as nf. The number of spills in all map tasks $\#_{spill}$ is computed in the spill stage, but each map task only merges its own spills here. Assuming that each map task merges the same number of spills, the number of spills that are merged in each map task can be calculated using $\#_{spill}/\#_{map}$, and the number of merge passes $\#_{merge}$ can be expressed as $\lceil log_{nf}(\#_{spill}/\#_{map})\rceil$. Let us suppose that Hadoop reads all the spills and writes their intermediate merge output in each merge pass, although Hadoop does not always need to merge all spills. Then, the cost of the sort stage can be computed as follows:

$$\begin{aligned} C_{sort} &= \#_{merge} \cdot \#_{map} \cdot (\frac{\#_{spill}}{\#_{map}} \cdot b_{buf} \cdot (c_{r\_loc} + c_{w\_loc}) + \frac{\#_{spill}}{\#_{map}} \cdot n_{buf} \cdot c_{c\_merge}) \\ &= \#_{merge} \cdot \#_{spill} \cdot (b_{buf} \cdot (c_{r\_loc} + c_{w\_loc}) + n_{buf} \cdot c_{c\_merge}) \end{aligned} \tag{3.8}$$

As in the spill stage, the cost of the sort stage in join processing with and without filters may vary according to the number of intermediate records in each.

Filter Merging

A filter merging process, which incurs an additional cost, is needed to apply filters. When tasktrackers process a map task for the first input dataset, one filter is created per partition, that is, per reducer. The number of filters that are created in each map task is equal to the number of reducers $\#_{reduce}$. Each tasktracker locally merges the filters from its own map tasks, so it maintains only $\#_{reduce}$ filters. The filters are merged globally after all map tasks for the first input dataset have been completed. During this process, tasktrackers wait for the merged filters and do not run other map tasks. This loss must also be taken into account.

Consequently, the total cost for merging filters T_{filter} can be divided into three parts: the cost of merging locally in each tasktracker C_{filter_local}, the cost of merging globally in the jobtracker C_{filter_global}, and the cost of waiting for the global filters without tasktrackers running other map tasks C_{filter_wait}. It can be computed as follows:

$$T_{filter} = C_{filter\_local} + C_{filter\_global} + C_{filter\_wait} \tag{3.9}$$

where

$$\begin{aligned} C_{filter\_local} &= \#_{map} \cdot \#_{reduce} \cdot c_{c\_union} \\ C_{filter\_global} &= \#_{nodes} \cdot \#_{reduce} \cdot (2 \cdot c_{tr} + c_{c\_union}) \\ C_{filter\_wait} &= \#_{nodes} \cdot \#_{map/node} \cdot \frac{T_{map}}{\#_{map}} \end{aligned} \tag{3.10}$$

$\#_{reduce}$ signifies the number of filters, and C_{filter_local} signifies the CPU cost for merging filters for all map tasks. $2 \cdot c_{tr}$ signifies the network cost of communicating the filters between a tasktracker and the jobtracker. C_{filter_global} include the network and CPU cost incurred by the jobtracker merging filters from all the tasktrackers. The waiting cost during the global filter merging C_{filter_wait} cannot be measured consistently, because it is affected by straggler nodes, input data skew, node capability, and so on. It is approximated as the cost for running the maximum number of map tasks simultaneously on all tasktrackers, assuming the difference between the finish time of the first node and that of the last node that finishes map tasks is smaller than the time to run a map task.

Reduce Task

The execution of a reduce task can be divided into the four stages shown in Figure 3.3. Note that the first stage of the reduce tasks is renamed to *Fetch*, whose original name was *Shuffle* in Herodotou's paper [23], because the term *shuffle phase* is used in the broad sense of the meaning. The total cost of reduce tasks T_{reduce} can be computed as the sum of the costs during the four stages of the reduce task.

$$T_{reduce} = C_{fetch} + C_{merge} + C_{reduce} + C_{write} \tag{3.11}$$

Fetch stage. In the fetch stage, the intermediate records produced from map tasks are copied from mappers to reducers. Suppose that no merging process occurs in this stage. Then, the total cost of the fetch stage is

$$C_{fetch} = n_{inter} \cdot l_{inter} \cdot c_{tr} \tag{3.12}$$

Merge stage. This stage merges the sorted partitions that are fetched from mappers. It works similar to the sort stage in map task but the number of merge passes $\#_{merge}$ is switched to $\lceil log_f(\#_{map}) \rceil$, because each reduce task merges $\#_{map}$ partitions that came from mappers. Suppose that all the partitions reside on disk. Then, the total cost of the merge phase can be computed as follows:

$$C_{merge} = \#_{merge} \cdot n_{inter} \cdot (l_{inter} \cdot (c_{r\_loc} + c_{w\_loc}) + c_{c\_merge}) \tag{3.13}$$

Reduce stage. The merged data is processed with a given reduce function. The total cost of the reduce stage is:

$$C_{reduce} = n_{inter} \cdot (l_{inter} \cdot c_{r\_loc} + c_{c\_red}) \tag{3.14}$$

It is clear that the costs of the above three stages depend on the number of intermediate records. Therefore, the costs during each stage in join processing with and without filters may be different.

Write stage. The final results of the job are written to the distributed file system. Let b_{out} be the size (in bytes) of the final results. Then, the costs of the reduce and write stages can be computed as follows:

$$C_{write} = b_{out} \cdot c_{w\_dfs} \tag{3.15}$$

Although the size of the final results may be unknown in advance, because it is determined by the join selectivity, the cost of the write stage is the same without regard to join techniques, as long as they produce correct join results. Therefore, the cost of the write stage can be omitted from cost estimation.

3.2.2 Effects of the Filters

The execution cost of a MapReduce job has been defined step by step in Section 3.2.1. The cost depends on how many intermediate records are filtered out. Define the *equilibrium false positive rate* f_{eq} as the false positive rate when the costs of join processing with and without filters are the same. If the real false positive rate is smaller than f_{eq}, it may not benefit from the filters. f_{eq} can be obtained by finding the false positive rate to make the difference of the total cost in the both cases D_{job} to zero. D_{job} can be defined as the sum of the cost differences in each stage.

$$\begin{aligned} D_{job} = T^f_{job} - T^h_{job} &= (T^f_{map} - T^h_{map}) + T_{filter} + (T^f_{reduce} - T^h_{reduce}) \\ &= (D_{collect} + D_{spill} + D_{sort}) + T_{filter} + (D_{fetch} + D_{merge} + D_{reduce}) \end{aligned} \tag{3.16}$$

The cost difference in the collect stage $D_{collect}$ can be simply computed with the given parameters as follows:

$$D_{collect} = C^f_{collect} - C^h_{collect} = (n_r + n_s) \cdot c_{c\_fltr} \tag{3.17}$$

The difference of the intermediate records d_{inter} after the collect phase is given as follows:

$$\begin{aligned} d_{inter} &= n^f_{inter} - n^h_{inter} \\ &= (n_r + n_s \cdot \sigma_{s\_r} + n_s \cdot (1-\sigma_{s\_r}) \cdot p) - (n_r + n_s) \\ &= -n_s(1-\sigma_{s\_r})(1-p) \end{aligned} \tag{3.18}$$

Since $0 \leq \sigma_{s\_r} \leq 1$ and $0 \leq p \leq 1$, unless σ_{s_r} is equal to one, the number of intermediate records n_{inter} may be reduced according to the false positive rate p, at the cost of applying filter operations. Accordingly, the difference in the number of spills $d_{\#_{spill}}$ in the spill and sort stage is

$$\begin{aligned} d_{\#_{spill}} &= \#^f_{spill} - \#^h_{spill} = \left\lceil \frac{n^f_{inter}}{n_{buf}} \right\rceil - \left\lceil \frac{n^h_{inter}}{n_{buf}} \right\rceil \\ &= \left\lceil \frac{(n_r + n_s \cdot \sigma_{s\_r} + n_s \cdot (1-\sigma_{s\_r}) \cdot p)}{n_{buf}} \right\rceil - \left\lceil \frac{n_r + n_s}{n_{buf}} \right\rceil \\ &= \begin{cases} -\left\lceil \frac{n_s(1-\sigma_{s\_r})(1-p)}{n_{buf}} \right\rceil &, \text{ if } n_s(1-\sigma_{s\_r})(1-p) \bmod n_{buf} \\ & \quad > (n_r+n_s) \bmod n_{buf} \\ -\left\lfloor \frac{n_s(1-\sigma_{s\_r})(1-p)}{n_{buf}} \right\rfloor &, \text{ otherwise} \end{cases} \end{aligned} \tag{3.19}$$

Consequently, the cost differences of the spill and sort stage between them can be expressed as follows, assuming that the numbers of merge passes for both join processing with and without filters are the same for simplicity.

$$\begin{aligned} D_{spill} &= d_{\#_{spill}} \cdot (n_{buf} \cdot \log_2(\frac{n_{buf}}{\#_{reduce}}) \cdot c_{c\_sort} + b_{buf} \cdot c_{w\_loc}) \\ D_{sort} &= d_{\#_{spill}} \cdot \#_{merge} \cdot (b_{buf} \cdot (c_{r\_loc} + c_{w\_loc}) + n_{buf} \cdot c_{c\_merge}) \end{aligned} \tag{3.20}$$

The cost depends on the number of intermediate records in both stages. Therefore, it is eventually affected by the ratio of the joined records and the false positive rates of the filters. Even if the numbers of merge passes differ, the cost still depends on those factors.

The cost differences of the fetch, merge, and reduce stage in the reduce tasks are

$$\begin{aligned}
D_{fetch} &= D_{inter} \cdot l_{inter} \cdot c_{tr} \\
D_{merge} &= D_{inter} \cdot \#_{merge} \cdot (l_{inter} \cdot (c_{r\_loc} + c_{w\_loc}) + c_{c\_merge}) \\
D_{reduce} &= D_{inter} \cdot (l_{inter} \cdot c_{r\_loc} + c_{c\_red})
\end{aligned} \tag{3.21}$$

These costs also depend on the number of intermediate records.

Note that all of the cost factors depend on the number of intermediate results. In other words, the total cost is determined by two factors: the ratio of the joined records and the false positive rate of the filters. The other parameters can be regarded as constants.

The equilibrium false positive rate f_{eq} can be pre-computed if the ratio of the joined records is known. It may be maintained in some systems. In Hive [3], for example, some research has been conducted that aims to optimize queries using table and column statistics [51, 20]. Otherwise, this information can be determined from certain parameters. The proposed cost model can help an optimizer to choose the processing order of the input datasets and whether to use filters.

3.3 Evaluation

This section presents the experimental results that were conducted for evaluation. First, the execution times and intermediate result sizes for the test query are compared against existing join methods. Next, those for the query are evaluated according to the types and sizes of filters.

3.3.1 Experimental Setup

All experiments were run on a cluster of 11 machines consisting of 1 jobtracker and 10 tasktrackers. Each machine had a 3.1 GHz quad-core CPU, 4 GB memory, and 2 TB hard disk.

The operating system was 32-bit Ubuntu 10.10, and the Java version used was 1.6.0_26.

Hadoop was configured based on the real-world cluster configuration parameters in the Hadoop official documentation [1]. The HDFS block size was set to 128 MB and the replication factor was set to three. Each tasktracker could simultaneously run three map tasks and three reduce tasks. The I/O buffer was set to 128 KB, and the memory for sorting data was set to 200 MB.

TPC-H benchmark [4] datasets were used for the experiments, varying the scale factor (SF) between 100, 200, and 300. The scale factor describes the entire database size of the dataset in gigabytes. The test query was a join between the two largest tables in the database, `lineitem` and `orders`. The `orderkey` column of the `lineitem` table is a foreign key to the `orderkey` column of the `orders` table. Thus, some selection predicates were added to control the join selectivities. The test query can be expressed in SQL-like syntax as follows:

```
SELECT substr(l.*, 0, $l_{lineitem}$), substr(o.*, 0, $l_{orders}$)
FROM lineitem l, orders o
WHERE l.orderkey = o.orderkey
  AND o.custkey < '?'
```

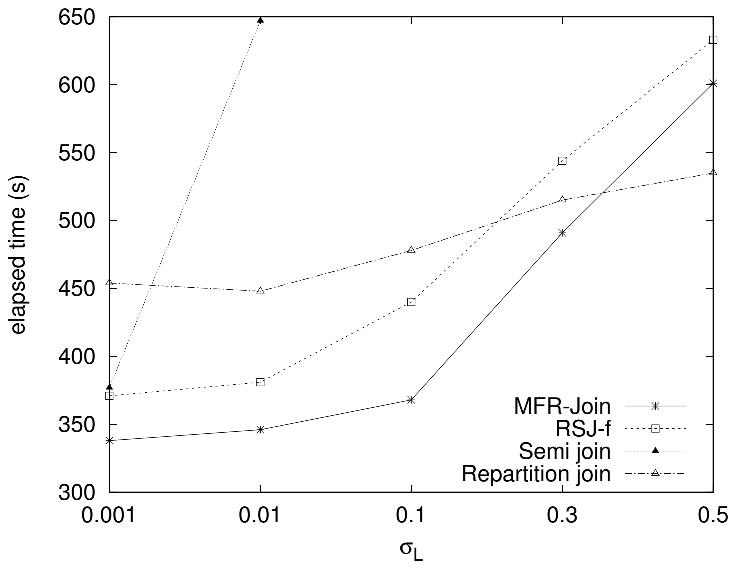
The query was run, changing the '?' in the predicate to set the ratio of joined records of `orders` with `lineitem` (σ_L) to between 0.001 and 0.5. In addition, the query results was set as substrings of the joined records in each table. $l_{lineitem}$ and l_{orders} are the lengths of the substrings, so the length of an intermediate record is $l_{lineitem} + l_{orders}$. As the length increased, the benefits from filtering grow. The lengths for both tables was set to 10, assuming the case of a projection query. The Hadoop program for the test query was hand-coded. `orders` was chosen as the first input, and `lineitem` was the second input.

3.3.2 Experimental Results

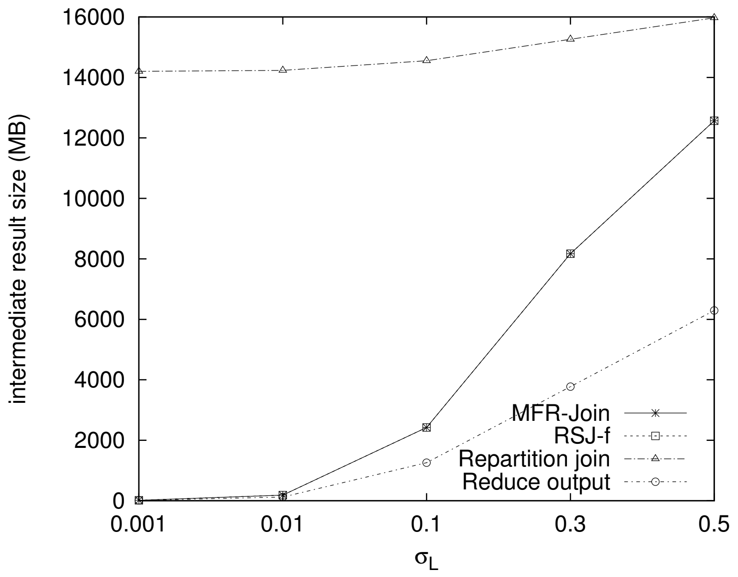
The performance of MFR-Join was compared to that of the existing repartition join [10], semijoin [10], and the reduce-side join with Bloom filters, which are created in an independent MapReduce job, referred to as RSJ-f [40]. Bloom filters were used for MFR-Join when its performance is compared to other join techniques so that the performance of the filters do not differ from RSJ-f. `MurmurHash` implemented in Hadoop was used as the hash function, and set the number of hash function $k = 2$ and the size of a filter to 4 Mb. This configuration was also used for RSJ-f.

Figure 3.4, 3.5, and 3.6 show the execution times and intermediate result sizes of the test queries using each join technique on various sizes of TPC-H datasets. Figure 3.4(a), 3.5(a), and 3.6(a) show similar patterns. The techniques using filters (MFR-Join and RSJ-f) show better performance than the repartition join when σ_L is small. Among these, MFR-Join outperforms RSJ-f, which processes the build input twice and has additional costs to initialize and cleanup an extra MapReduce job. If more records participate in the join, the performance becomes worse because the number of redundant records that can be filtered out is reduced. Semijoin did not finish when σ_L was greater than or equal to 0.1, because it ran out of memory.

Figure 3.4(b), 3.5(b), and 3.6(b) show the intermediate result sizes in each test case. Those of the semijoin were excluded because it uses a map-side join for the second and third jobs. In each case, MFR-Join and RSJ-f have the same size intermediate results. Instead, RSJ-f runs an extra MapReduce job to create the Bloom filters. The repartition join has the largest size, because it emits all probe input records as intermediate results. The intermediate result sizes increase as σ_L increases, and this leads to an increase in execution times. As the size of the filters is fixed, the probability of false positives increases with

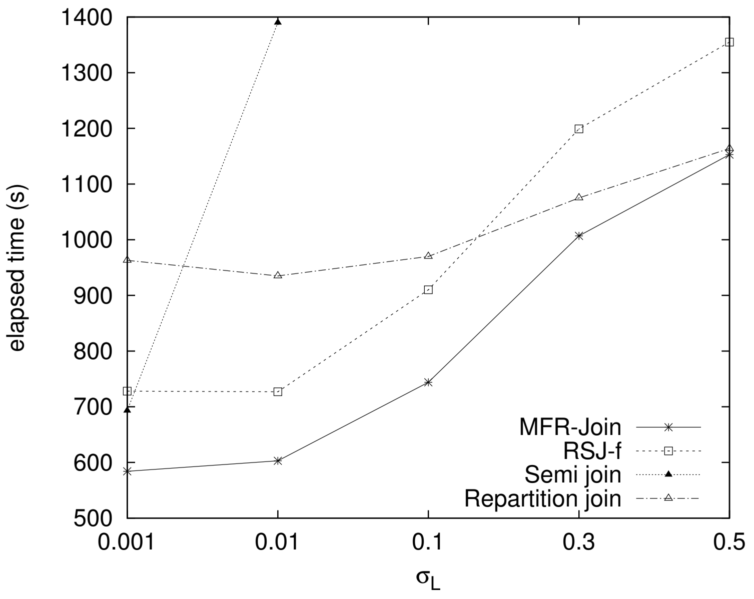


(a) Execution times

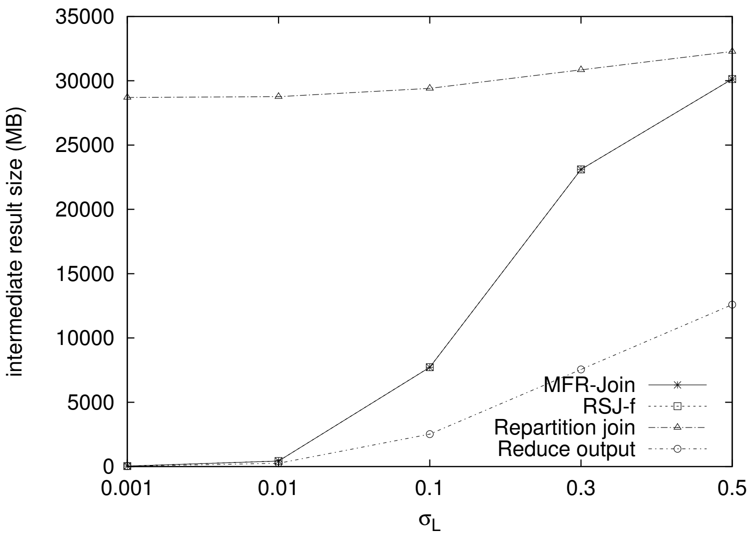


(b) Intermediate result sizes

Figure 3.4: Experimental results: varying the ratio of joined records (SF=100)

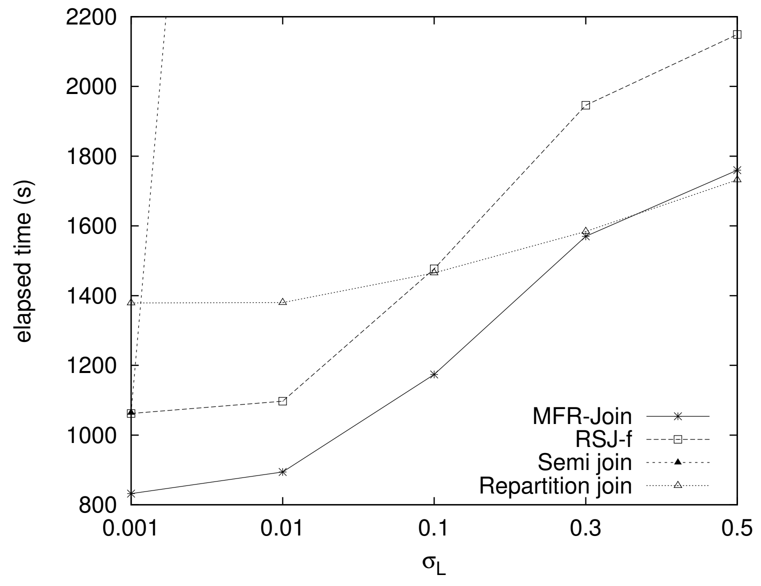


(a) Execution times

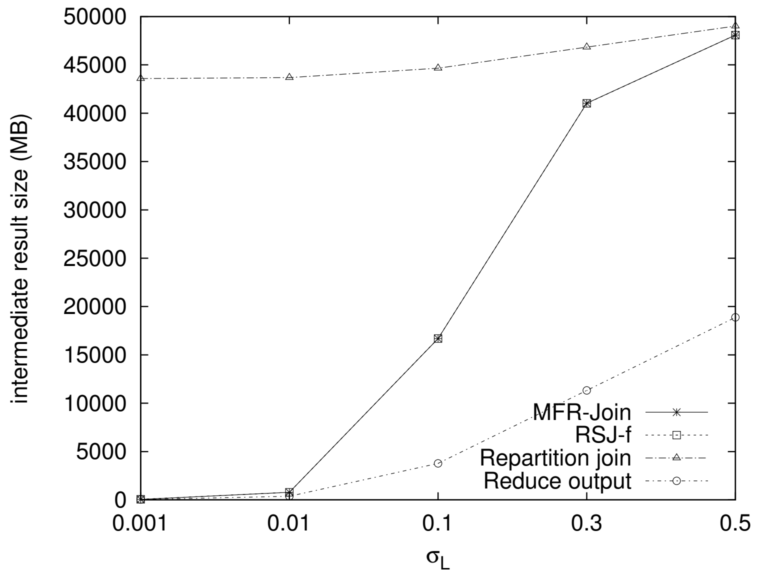


(b) Intermediate result sizes

Figure 3.5: Experimental results: varying the ratio of joined records (SF=200)

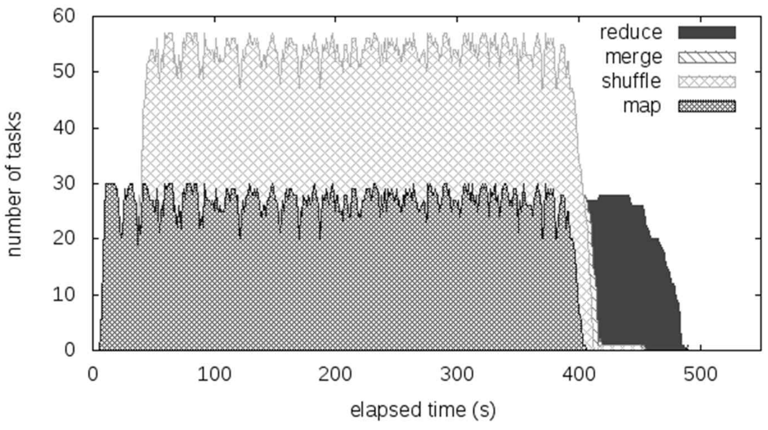


(a) Execution times

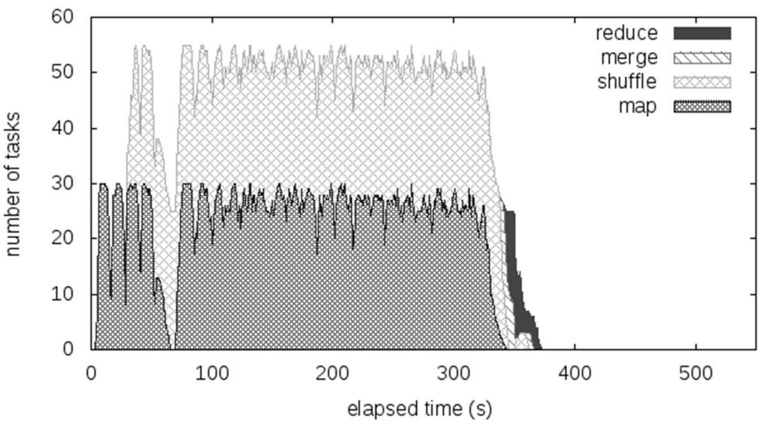


(b) Intermediate result sizes

Figure 3.6: Experimental results: varying the ratio of joined records (SF=300)



(a) MFR-Join



(b) Hadoop

Figure 3.7: Task timelines

the number of join keys that are inserted into the filters. Consequently, the number of false positives, the gap between the intermediate results sizes and reduce output sizes, also increases. This is more obvious in the case of a large scale factor.

Figure 3.7(a) and 3.7(b) show the task timelines of Hadoop and MFR-Join with Bloom filters during the execution of the test query with the scale factor 100 and the ratio of joined

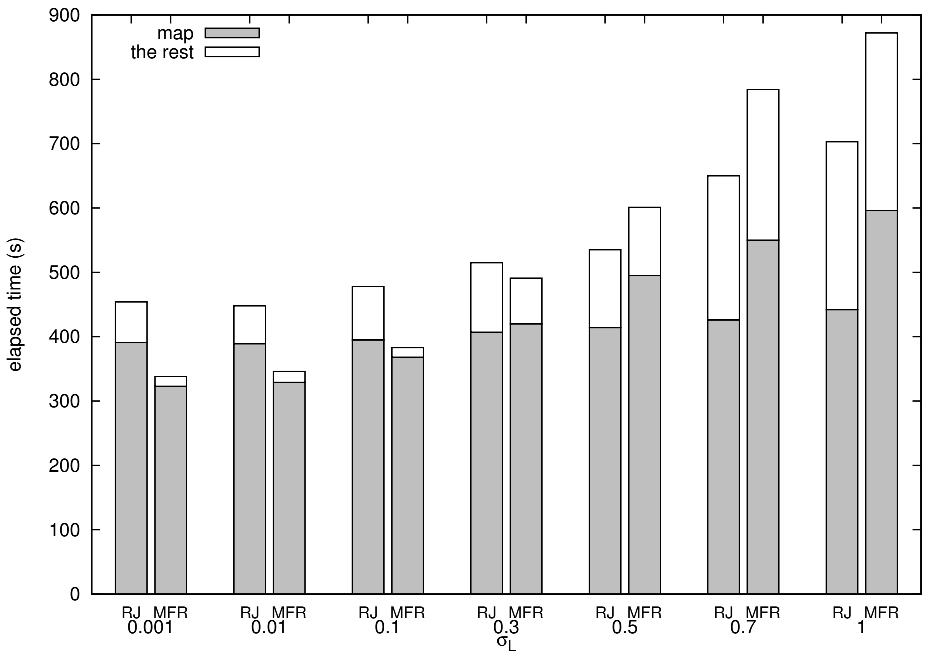


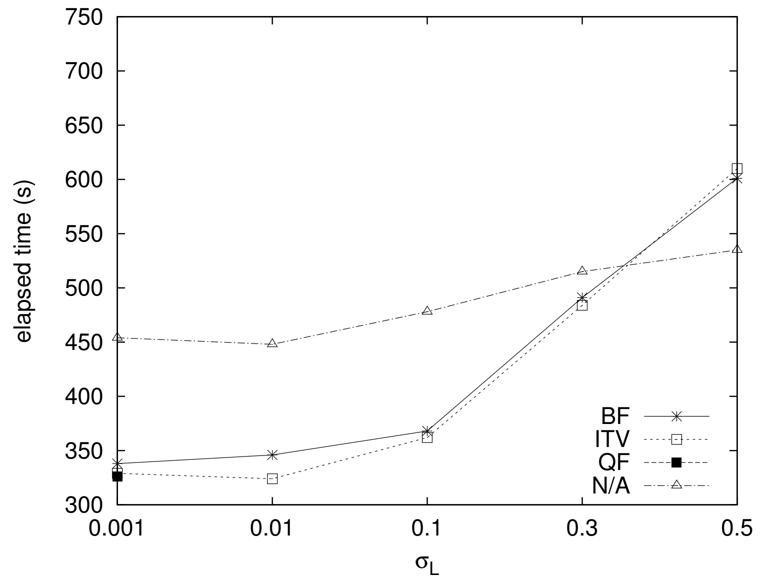
Figure 3.8: Map phase time: varying the ratio of joined records (SF=100)

records 0.1. A key observation is that the number of running map tasks is sharply decreased for a while in Figure 3.7(b). It means that tasktrackers do not run the map tasks for the second input dataset during the global filter construction phase. In spite of the overhead of this period, the execution time of all map tasks and reduce tasks is considerably reduced. Map phase is finished early because the intermediate results are reduced by the filters, and the local I/O and sorting cost are also reduced as a result. Reduce phase is also finished early because the number of intermediate records to be copied from remote map processes is reduced, so the number of input records to process in reduce function is decreased.

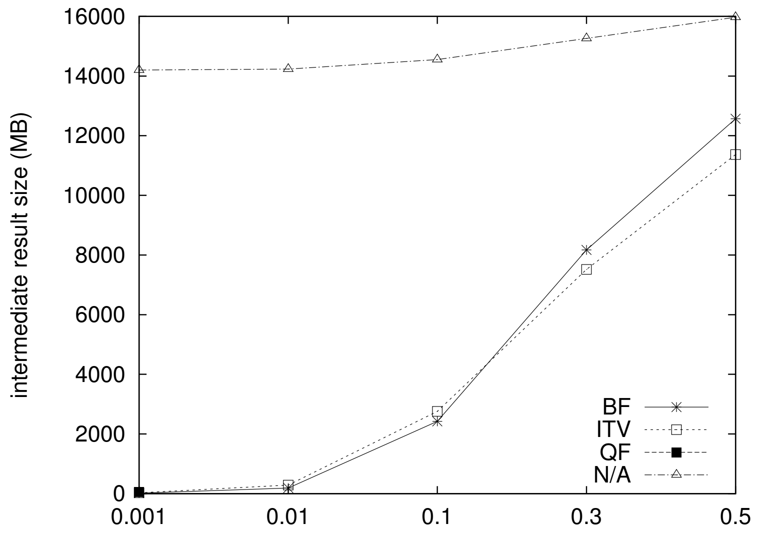
The reason why the execution time of MFR-Join increases as the ratio of joined records increases can be found in Figure 3.8. It can be observed that the map phase time grows more quickly in MFR-Join (MFR) compared to the results of the repartition join (RJ), as the ratio of joined records increases. This is because more time is needed for creating and probing filters as the number of intermediate results increases. MFR-Join is efficient when small portions of input datasets are joined, but otherwise it may be rather inefficient. To avoid such inefficiency, an adaptive join method will be proposed in Chapter 4.

Next, the performance of MFR-Join were measured with three types of filters, the Bloom filter (BF), the interval filter (ITV), and the quotient filter (QF). The parameters for each filter were adjusted to set the size of a filter to 4 Mb. Figure 3.9 shows the execution times and intermediate result sizes of the test query. As shown in Figure 3.9(a), The performance tends to decrease as more records participate in the join, because the number of redundant records that can be filtered out is reduced, regardless of the filtering technique used. When σ_L is 0.5, the repartition join (N/A) rather shows better performance than the others. In Figure 3.9(b), the repartition join is largest in size because it emits all input records as intermediate results. As σ_L increases, the gap of the intermediate result sizes between the repartition join and non-adaptive joins with filters decreases. When σ_L is small, a large number of redundant records are filtered out; whereas, when σ_L is large, the sizes of the intermediate result increase, which leads to the increases in the execution times in Figure 3.9(a). It results from the large false positive rates corresponding to the increased number of join keys that are inserted into the filters. Adaptive join methods, which will be explained in Chapter 4, can save the performance loss in MFR-Join.

Finally, the test query was run with various sizes of the Bloom filter, which is generally used, from 512 Kb to 8 Mb. Figure 3.10(a) and 3.10(b) show the execution times and intermediate result sizes, respectively. It can be observed that the 512 Kb Bloom filter is sufficient when σ_L is 0.001. The intermediate result sizes is barely reduced as the size of the Bloom filter increased. Rather, a large-sized filter increases the execution time. If the

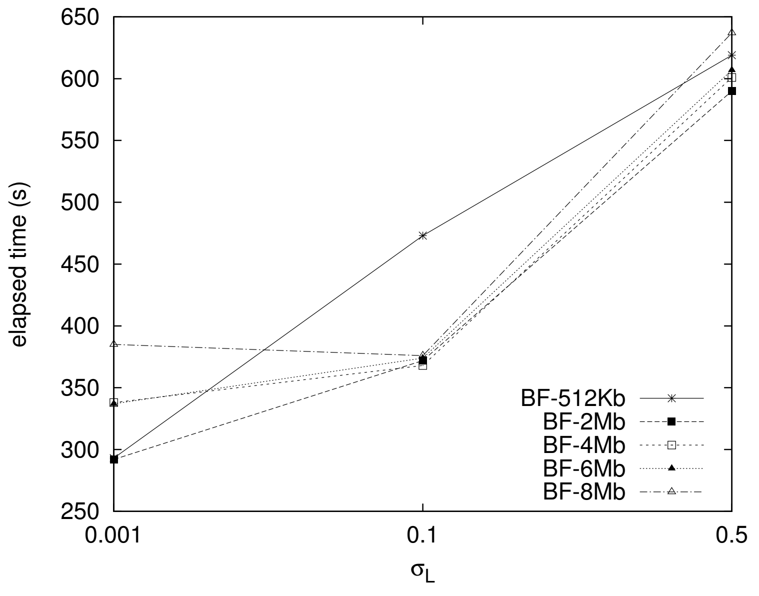


(a) Execution times

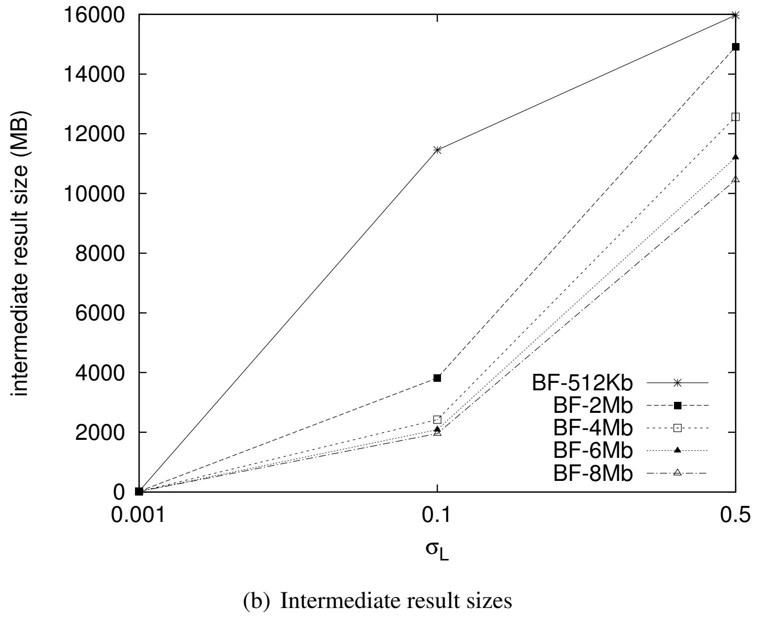


(b) Intermediate result sizes

Figure 3.9: Experimental results with various filtering techniques (SF=100)



(a) Execution times



(b) Intermediate result sizes

Figure 3.10: Experimental results with various Bloom filter sizes (SF=100)

filter size is too large, the overhead for constructing and communicating the filters offsets the benefits of filtering. On the other hand, it can be observed that the 512 Kb Bloom filter is too small when σ_L is 0.5. Only a small number of redundant records is filtered out. This also increases the execution time. If the size of the Bloom filters is too small, redundant records cannot be filtered out. Therefore, it is important to determine the most appropriate size for the filter. This can be determined from statistical information about the input datasets, and this will be addressed in future work.

Chapter 4

Adaptive Join Processing with Filtering Techniques in MFR-Join

In addition to MFR-Join described in Chapter 3, a few researchers have recently attempted to reduce the size of the redundant intermediate results using Bloom filters [11] for join processing in MapReduce [40, 28, 57]. Though these approaches apply Bloom filters in slightly different ways, they involve filtering out the intermediate results that are not joined, so they are efficient when a small portion in an input dataset is joined. However, the performance of these approaches is worse than join processing without filters when large portions of records are joined or the number of distinct keys is large. Further, if statistical information for input datasets is not available or inaccurate, they may be inefficient because filter parameters like the number of bits and the number of hash functions cannot be optimally adjusted. In such cases, applying the filters incurs additional overhead due to the computing of hash values for each record and the merging of the filters. This is because they all apply the filters without regard to their performance.

To handle such cases, an adaptive join processing method with filtering techniques is proposed in this chapter. MFR-Join is enhanced to monitor the performance of filters by means of false positive rates, and to disable filters whose false positive rates are greater than a user-configured threshold. The false positive rates can simply be computed when it merges the filters that are created in all nodes. Because it is advantageous to detect poor filter performance early, it estimates the false positive rates of merged filters with those of individual filters before they are merged. Further, two map task scheduling policies, synchronous and asynchronous scheduling, are defined. In synchronous scheduling, map tasks for the second input dataset are not assigned during the merging phase. In contrast, under asynchronous scheduling, the map tasks are continuously assigned, though some tasks can be processed without the merged filters. Under our scheduling policies, the processing cost is affected by the processing order of the two input datasets. Therefore, a method to choose the processing order based on the estimated cost is presented.

The remainder of this chapter is organized as follows: Section 4.1 explains the design and implementation details. Section 4.2 discusses the effect of FPR threshold value on join costs. Finally, Section 4.3 discusses the experimental results.

4.1 Adaptive join processing in MFR-Join

This section explains an adaptive join processing method with various filtering techniques in MFR-Join. Because this is an extension of MFR-Join that applies filters non-adaptively, which is explained in Section 3.1.1, modified parts for adaptive joins are described in detail, and the other common parts will be briefly described.

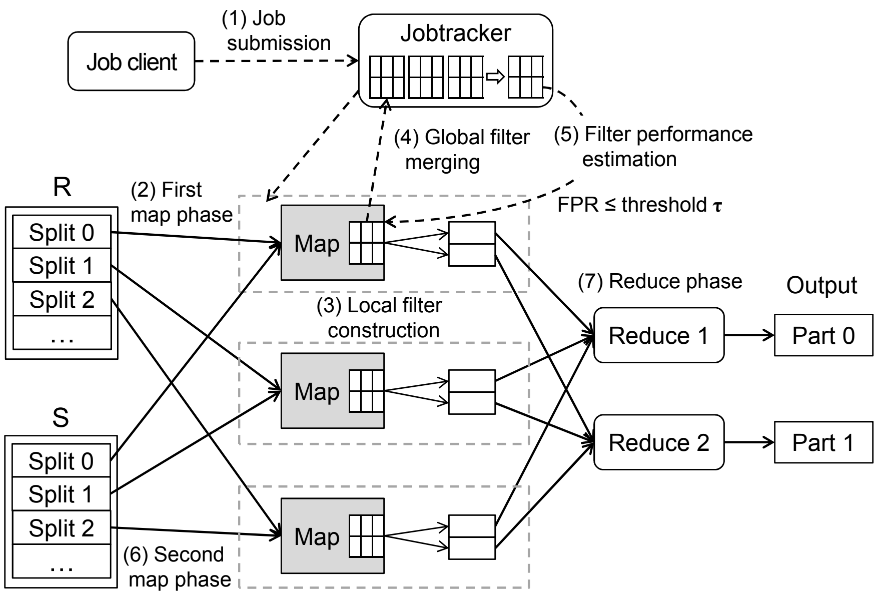


Figure 4.1: Execution overview

4.1.1 Execution Overview

Figure 4.1 depicts an example of an adaptive join between two datasets, R and S, in MFR-Join. As in Section 3.1.1, suppose that R is chosen to be processed first; that is, filters are built on R. When a user runs a MapReduce program, the following sequence of actions is performed.

1-3. **Job submission, first map phase, local filter construction.** These steps are the same as in non-adaptive joins. m_1 map tasks for R, m_2 map tasks for S, and r reduce tasks are created. The jobtracker assigns the m_1 map tasks for R or the reduce tasks to tasktrackers. Mappers execute the map function and produce intermediate pairs. The pairs are divided into r partitions, which are sent to reducers. For each partition, local filters are created.

4. **Global filter merging.** When all m_1 map tasks for the first input dataset R have been completed, the jobtracker signals all tasktrackers to send it their local filters via heartbeat responses. Then, the jobtracker merges all local filters to construct the *global filters* for R. The difference is that the jobtracker may continue to assign map tasks according to the map task scheduling policy. While the merging is in progress, the filter performance is estimated.

5. **Filter performance estimation.** The jobtracker estimates the performance of the merged global filters by estimating their false positive rates. The method used to estimate the false positive rates is set according to the applied filter; this issue is described in Section 4.1.2. Assuming that the false positive rates can be estimated, those filters whose false positive rates exceed the given threshold are disabled. The jobtracker then sends the global filters to all the tasktrackers.

6. **Second map phase.** The jobtracker assigns the m_2 map tasks for S or the remaining reduce tasks to the tasktrackers. According to the map task scheduling policy, tasktrackers may be assigned the map tasks before they receive the global filters. Consequently, some map tasks may be processed without the filters. After the tasktrackers receive the global filters, they run the assigned tasks with the filters.

7. **Reduce phase.** This step is the same as in non-adaptive joins. Each reducer runs the reduce function and produces final output results.

The filter performance estimation and two map task scheduling policies have been added. The following subsections explain the details on the features.

Table 4.1: `Merge` and `estimateFPR` operations for some example filters

	`merge`	`estimateFPR`
Bloom filter	Bitwise OR	$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k}$
Interval filter	Bitwise OR	$\left(1-\left(1-\frac{1}{m}\right)^{n}\right)$
Quotient filter	Similar to merge sort in DBMS	$\left(1-\left(1-\frac{1}{2^{(q+r)}}\right)^{n}\right)$

4.1.2 Additional Filter Operations for Adaptive Joins

To support adaptive joins, in addition to the `merge` operation, the `estimateFPR` operation is required for the filters.

- `estimateFPR()`: computation of its false positive rate.

The filters that do not support `estimateFPR` operation can be applied to MFR-Join, but they cannot be used for adaptive joins.

The example filtering techniques mentioned in Section 3.1.4, which are Bloom filter [11], the interval filter [44], and the quotient filter [8], support the operation. Table 4.1 shows the additional operations for the example filters.

To compute the false positive rates of the example filters, the number of inserted elements n in the merged filters have to be known. However, keeping track of all the distinct values inserted into each local filter in order to compute n is impractical. The number of true bits in the merged filters is available, but not that of inserted elements. Therefore, the number of inserted elements is estimated with the expected number of true bits in the filters after n elements are inserted [41]. Let p_0 be the probability of a bit being *false* after n elements are inserted for a filtering technique. For example, p_0 is $(1-\frac{1}{m})^n$ for the Interval filter and $(1-\frac{1}{m})^{kn}$ for the Bloom filter because it uses k hash functions. Thus, the probability of a bit being *true* is $(1-p_0)$. Therefore, the expected number of true bits can be computed by multiplying it by the total number of bits for a filter.

4.1.3 Early Detection of FPR Threshold Being Exceeded

The execution flow described in Section 4.1.1 estimates the false positive rate after the merging of all local filters into global filters. Before the estimation, all mappers insert the join keys of the first input dataset into the filters, which incurs CPU overhead. Further, during the global filter merging, network I/O cost associated with communicating the filters between the jobtracker and tasktrackers is incurred. If it finds that the FPRs of global filters is greater than the given threshold in advance, the costs resulting from the creation and merging of the filters can be reduced. Further, the earlier the situation is detected, the more costs can be reduced.

Before the merge, only the FPRs of the individual local filters in each tasktracker can be computed. However, if the FPRs of the global filters can be estimated along with them, then the situation could be detected much earlier. Consequently, an optional operation, `estimateUnionFPR`, was added to the filtering techniques. If this operation is supported, the jobtracker can estimate the FPRs of the global filters as follows: In the first map phase, each tasktracker sends the jobtracker only the FPR values of its local filters, not the whole local filters themselves, via heartbeat messages when a map task is complete and its FPRs are changed. The jobtracker then estimates the FPRs of the global filters using the operation.

The FPRs of the global filters can be estimated with those of the individual local filters using methods similar to those used by Michael et al. [37]. Let the FPRs of two local filters be $P(A)$ and $P(B)$. The FPR of the merged filter $P(A \cup B)$ can be computed as $P(A) + P(B) - P(A \cap B)$. Assuming that the distribution of data is independent, $P(A \cap B) = P(A) \cdot P(B)$, $P(A \cap B)$ can be estimated with individual FPRs $P(A)$ and $P(B)$.

The signature of the operation was designed as `estimateUnionFPR(FPRs`$_{global}$`, FPRs`$_{local\_prev}$`,FPRs`$_{local\_cur}$`)`. Note that both the previous and current FPRs of the local filters are required. The jobtracker should repeatedly merge the local filters from a tasktracker, with their changed FPRs. However, estimating the FPRs of the merged filters may not be idempotent. If it merges the same local filters multiple times, its FPRs will be changed. In this case, the `estimateUnionFPR` operation must first compute the FPRs of the global filters, with the exception of the previous FPRs of the local filters, and then estimate the new FPRs of the global filters by merging the current FPRs of the local filters. This is possible if the operation is commutative. In our operations, the order in which the filters are merged does not affect the estimation results.

4.1.4 Map Task Scheduling Policies

Int his section, two scheduling policies of synchronous and asynchronous scheduling are defined. These are similar in that they assign map tasks in the order of the input datasets, but their behavior is different during the global filter merging phase.

Synchronous Scheduling

Under the synchronous scheduling policy, our task scheduler does not assign the map tasks for the probe input during the global filter merging phase. Instead, the assignment is deferred until the global filters are constructed and sent to the tasktrackers. Then, every probe input split can be processed with the filters, so more redundant intermediate results can be filtered out.

However, under this policy, all tasktrackers cannot run the map tasks for the probe input, and should wait until the global filter construction is finished. (Of course, they can run the copy operations of reduce tasks or the tasks of other MapReduce jobs.) The waiting time could be long, especially if straggler nodes exist. Then, the gain from the filtering is offset by the loss from such waiting. Hadoop has a feature known as speculative execution, in which multiple copies of the same task are run on different tasktrackers when the job is close to completion. The waiting time can be reduced using speculative execution during the global filter merging phase.

Asynchronous Scheduling

Even if the waiting time during the global filter merging phase is not long, the loss from the waiting may be large depending on the size of the records that are filtered out. Under asynchronous scheduling, the task scheduler continues to assign the map tasks for the probe input without such waiting. Tasktrackers can run these tasks without filtering until they receive the global filters.

In this policy, tasktrackers do not need to wait during the global filter merging phase. Instead, the size of the intermediate results may be increased. There is a tradeoff between synchronous and asynchronous scheduling, and it depends on the waiting time and the filter performance.

4.1.5 Additional Parameters for Adaptive Joins

Several parameters were added to configure MFR-Join for adaptive joins, as shown in Table 4.2. Users can define these in Hadoop configuration files, or specify them as runtime parameters.

Table 4.2: Additional parameters for adaptive joins

Parameter	Description	Type	Default value
mapred.filter.adaptive	whether to use filters adaptively	boolean	false
mapred.filter.adaptive.in-progress	whether to use early detection	boolean	false
mapred.filter.fpr.threshold	FPR threshold	float	0.5
mapred.filter.async	asynchronous/synchronous map task scheduling policy	boolean	false

4.2 Join Cost and FPR Threshold Analysis

The aim of the proposed adaptive join method is to guarantee join performance that is close to or better than join processing with and without filters by disabling those filters whose estimated FPRs are greater than the user-configured threshold. Thus, it is important for users to appropriately set the threshold value. This section outlines how to choose the FPR threshold value. Based on the cost model described in Section 3.2, the cost of adaptive join and the effects of the FPR threshold on the cost will be discussed.

4.2.1 Cost of Adaptive Join

MFR-Join disables filters whose estimated FPRs exceed the FPR threshold value τ, which is set to a value between zero and one as a parameter. The execution cost of adaptive join depends on how many filters and when the filters are disabled. Although each filter is created per partition and can be individually disabled, suppose that all filters have the same FPR with a uniform distribution of join keys, for simplicity. Let n_m be the number of map tasks that are run before the filters are disabled, and p be the actual FPR value of the filters for the join. Then, the total cost of adaptive join T^a_{job} can be expressed as follows:

$$T^a_{job} = \begin{cases} T^f_{map} + T_{filter} + T^f_{reduce} \text{ , if } p \leq \tau \\ \\ \frac{\#_{map_r}}{\#_{map}} \cdot T^f_{map} + \frac{\#_{map_s}}{\#_{map}} \cdot T^h_{map} + T_{filter} + T^h_{reduce} \text{ , if } p > \tau \text{ is detected} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in global filter merging} \\ \frac{n_m}{\#_{map}} \cdot T^f_{map} + \frac{\#_{map-n_m}}{\#_{map}} \cdot T^h_{map} + T^h_{reduce} \text{ , if } p > \tau \text{ is detected} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{before global filter merging} \end{cases} \tag{4.1}$$

Since the total cost is affected by the FPR threshold value τ, it is important for users to set the threshold value appropriately. If τ is too small, it may not benefit from the filtering of redundant intermediate records. On the other hand, if τ is too large, it may incur unnecessary filtering costs.

4.2.2 Effects of FPR Threshold

In Section 3.2.2, the *equilibrium false positive rate* f_{eq} has been defined as the false positive rate when the costs of join processing with and without filters are the same. If the threshold value τ is larger than f_{eq}, it may result in the filters being used in cases where intermediate records are not filtered out enough to improve the execution time. Conversely, if τ is smaller than f_{eq}, it may result in the filters not being used in cases where the intermediate records are filtered out enough.

Unfortunately, if the ratio of the joined records is not known in advance, f_{eq} cannot be computed before the join execution. Under such situation, performance degradation have to be reduced. Instead of f_{eq}, the FPR threshold can be configured as the FPR value computed with the maximum ratio of the joined records, which is chosen by users or estimated by a query optimizer module. As the maximum ratio decreases, the corresponding FPR threshold increases because the number of intermediate records that are not joined and can be filtered out increases, while the number of intermediate records that cause D_{job} to zero with the join ratio is fixed. Thus, the opportunity to apply filters increases, but it may be rather inefficient.

It is suggested to defensively choose the maximum ratio of the joined records as a large enough value, in case the filters are not beneficial. The FPR threshold value, which corresponds to the ratio, would then be small enough. If the FPR threshold is small, the opportunity to improve performance with filters is reduced. For example, filters may not be applied if the join ratio is much smaller than the maximum ratio and the actual FPRs are slightly larger than the threshold value. Despite this, it should be safely used as an option to ensure stable join performance.

4.2.3 Effects of Map Task Scheduling Policy

The map task scheduling policies, which is described in 4.1.4, affect the number of the intermediate map output records n_{inter}^{f} in Equation 3.5. Let σ_{s_r} be the ratio of the joined records of S with R. Assuming that R is the build input, n_{inter}^{f} can be estimated using the false positive rate p for each scheduling policy as follows:

$$n_{inter}^{f} = \begin{cases} n_r + n_s \cdot \sigma_{s\_r} + n_s \cdot (1 - \sigma_{s\_r}) \cdot p & \text{, under synchronous scheduling} \\ n_r + n_{s\_a} + n_{s\_f} \cdot \sigma_{s\_r} + n_{s\_f} \cdot (1 - \sigma_{s\_r}) \cdot p & \text{, under asynchronous scheduling} \end{cases} \tag{4.2}$$

where $n_{s\_a} = min(b \cdot n_m, |R|)$, $n_{s\_f} = n_s - n_{s\_a}$.

In asynchronous scheduling, n_{s_a} gives the size of the probe input splits that are processed without global filters, and n_{s_f} gives the size of the probe input splits that are processed with the filters. As some probe input splits are processed without the filters, the number of the intermediate results in asynchronous scheduling may be larger than that under synchronous scheduling. Instead, the asynchronous scheduling eliminates the waiting cost C_{filter_wait} in Equation 3.10, because tasktrackers do not need to wait global filters. While synchronous scheduling produces the smaller number of the intermediate results, it suffers from the waiting time during the global filter merging phase.

4.3 Evaluation

This section presents the experimental results that were conducted for evaluation. First, the execution times and intermediate result sizes of adaptive joins are compared against those of non-adaptive joins. Next, the execution times are evaluated according to various FPR threshold values. Finally, the execution times and intermediate result sizes under the synchronous and asynchronous scheduling policy are evaluated.

4.3.1 Experimental Setup

The experimental environment is the same as in Section 3.3.1. All the experiments were run on a cluster of 11 machines consisting of one jobtracker and 10 tasktrackers. Each machine comprised a 3.1 GHz quad-core CPU, 4 GB RAM, and a 2 TB hard disk. Hadoop was configured based on the real-world cluster configuration parameters in the Hadoop official documentation [1]. The HDFS block size was set to 128 MB and the replication factor was set to three. Each tasktracker could simultaneously run three map tasks and three reduce tasks. The I/O buffer was set to 128 KB, and the memory for sorting data was set to 200 MB.

TPC-H benchmark [4] datasets were used with scale factor 100. The scale factor is the entire database size of the dataset in gigabytes. The test query was a join between the two largest tables in the database, `lineitem` and `orders`, and can be expressed in SQL-like syntax as follows:

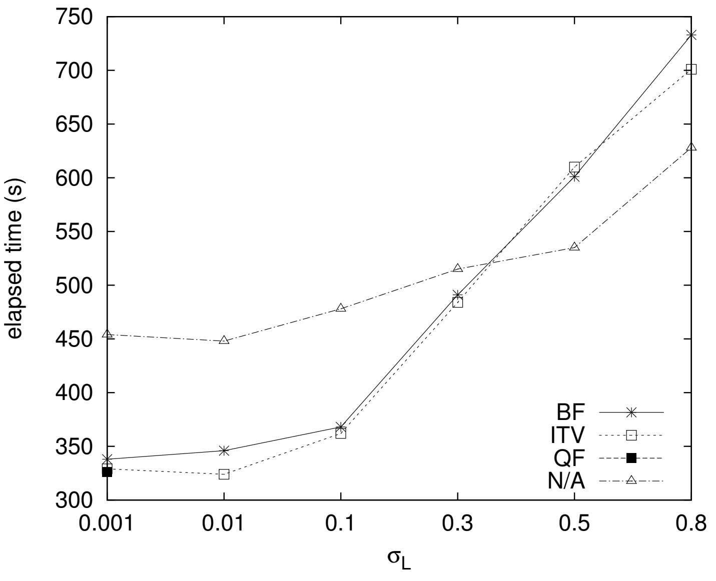
```
SELECT substr(l.*, 0, $l_{lineitem}$), substr(o.*, 0, $l_{orders}$)
FROM lineitem l, orders o
WHERE l.orderkey = o.orderkey
  AND o.custkey < '?'
```

The query was run, changing the '?' in the predicate to set the ratio of joined records of `orders` with `lineitem` (σ_L) to between 0.001 and 0.8. In addition, the query results was set as substrings of the joined records in each table. $l_{lineitem}$ and l_{orders} are the lengths of the substrings, so the length of an intermediate record is $l_{lineitem} + l_{orders}$. As the length increased, the benefits from filtering grow. The lengths for both tables was set to 10, assuming the case of a projection query. The Hadoop program for the test query was hand-coded. `orders` was chosen as the first input, and `lineitem` was the second input.

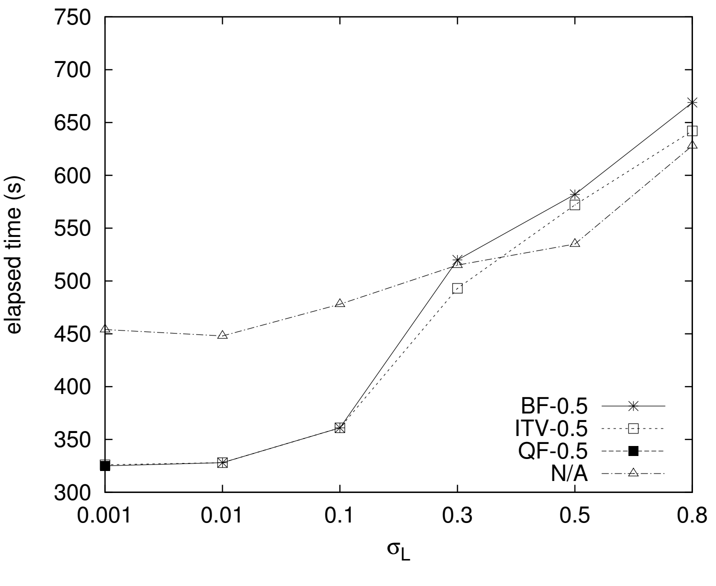
4.3.2 Experimental Results

The performance of our adaptive join method was compared to that of the existing repartition join [10] and non-adaptive join with filters. Three types of filters, the Bloom filter (BF), the interval filter (ITV), and the quotient filter (QF), were applied, and the parameters for each filter were adjusted to set the size of a filter to 4 Mb.

First, the test query was run, varying the ratio of the joined records. The FPR threshold for adaptive joins was set to 0.5. Figure 4.2 shows the execution times of the test query using each join technique. As shown in Figure 4.2(a), The performance of non-adaptive joins decreases as more records participate in the join, because the number of redundant records that can be filtered out is reduced. On the other hand, in Figure 4.2(b), our adaptive joins exhibits performance close to that of the repartition join (N/A) when σ_L is large, regardless of the filtering technique used. Note that joins with Quotient filters did not finish when σ_L was greater than or equal to 0.01, because it cannot contain more than its size of elements

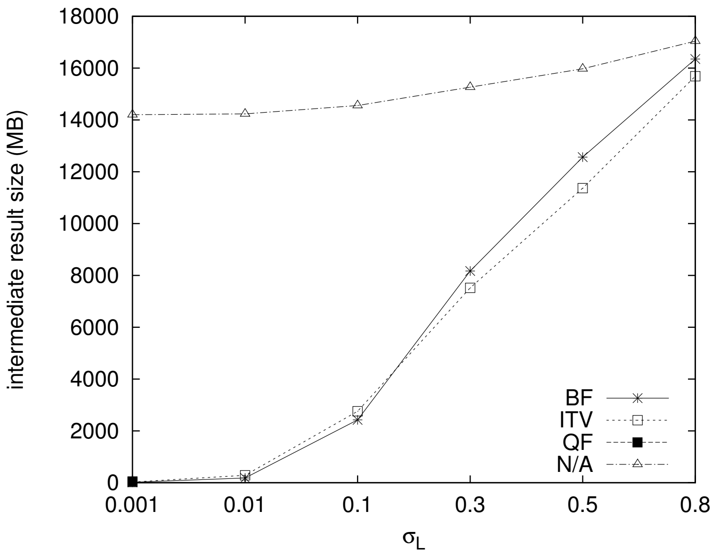


(a) Non-adaptive join

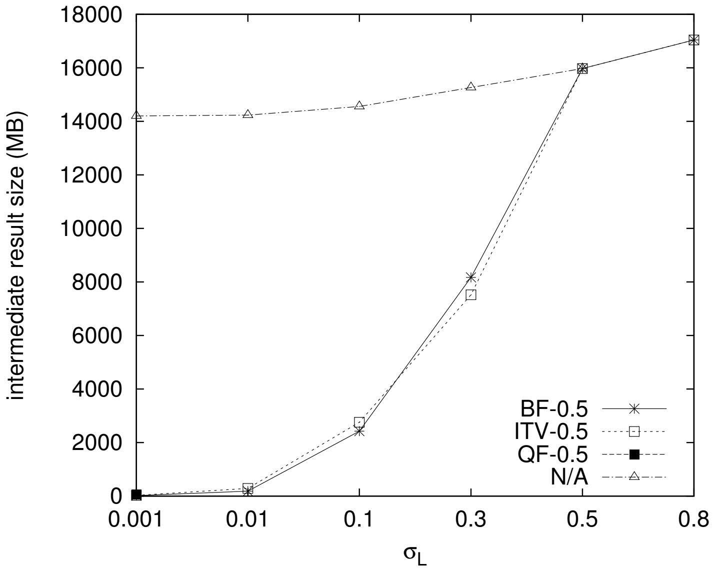


(b) Adaptive join ($\tau = 0.5$)

Figure 4.2: Execution times with various filtering techniques



(a) Non-adaptive join



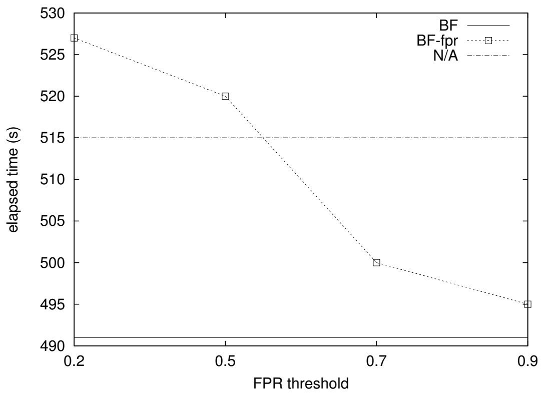
(b) Adaptive join ($\tau = 0.5$)

Figure 4.3: Intermediate results sizes with various filtering techniques

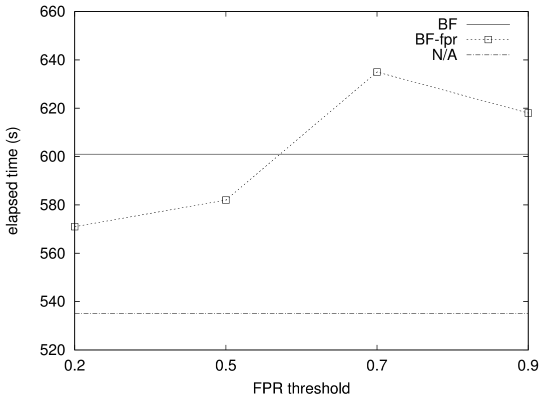
and its merge operations have a time complexity on the order of the number of inserted elements.

Figure 4.3 shows the intermediate result sizes in each case. In Figure 4.3(a), The repartition join is largest in size because it emits all input records as intermediate results. As σ_L increases, the gap of the intermediate result sizes between the repartition join and non-adaptive joins with filters decreases. When σ_L is small, a large number of redundant records are filtered out; whereas, when σ_L is large, the sizes of the intermediate result increase, which leads to the increases in the execution times in Figure 4.2(a). It results from the large FPRs corresponding to the increased number of join keys that are inserted into the filters. In our adaptive joins, as shown in Figure 4.3(b), the intermediate result size rather increases when σ_L is greater than or equal to 0.5. As σ_L is larger than the given FPR threshold, the filters are disabled and the join is executed like the repartition join. Although the size of the intermediate results increases, the adaptive joins exhibit better performance than non-adaptive join with filters because the costs of creating, merging, and checking the filters are saved.

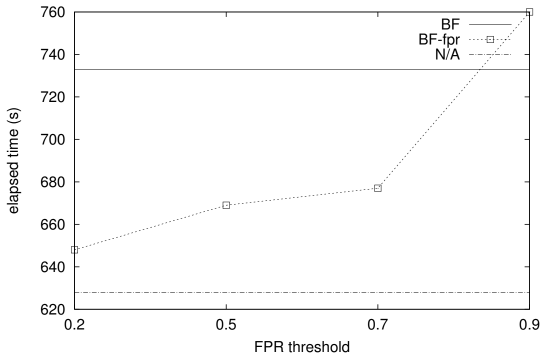
Figure 4.4 and 4.5 show the execution times with the Bloom filters and the interval filters respectively, obtained while varying the FPR threshold. It can be seen that join processing with filters performs better when σ_L is 0.3, but the repartition join performs better when σ_L is 0.5 or 0.8. As shown in Figure 4.4(a) and 4.5(a), if the FPR threshold is too small, the filters may be disabled despite the fact that they are efficient. Conversely, if the FPR threshold is too large, the filters may not be disabled despite the fact that they are inefficient, as shown in Figure 4.4(c) and 4.5(c). The optimal threshold value varies according to the query and data. Therefore, it is important to set the appropriate FPR threshold for each query, as stated in Section 4.2.2, so that it exhibits performance that is close to that exhibited by the better



(a) σ_L=0.3

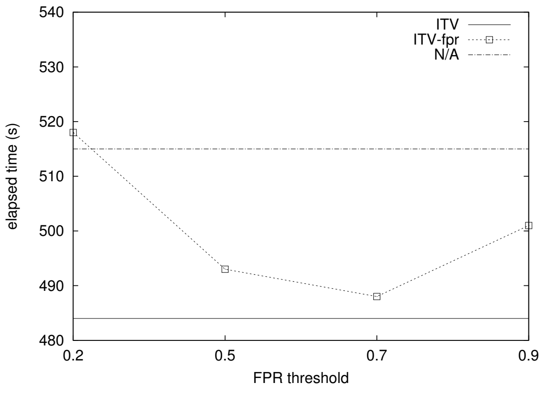


(b) σ_L=0.5

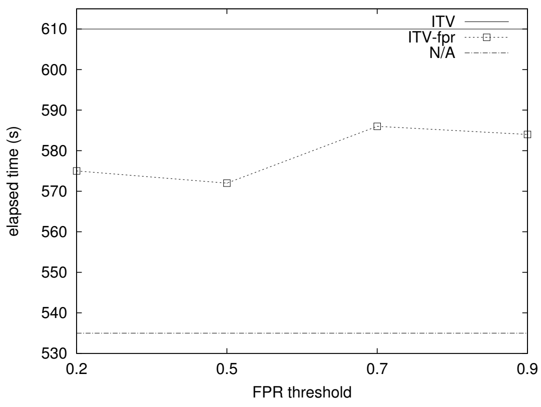


(c) σ_L=0.8

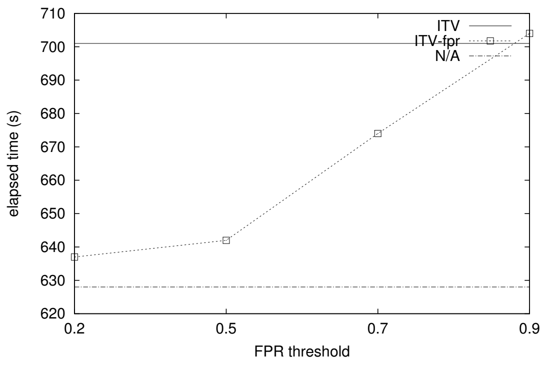
Figure 4.4: Execution times with Bloom filters varying FPR thresholds



(a) σ_L=0.3



(b) σ_L=0.5



(c) σ_L=0.8

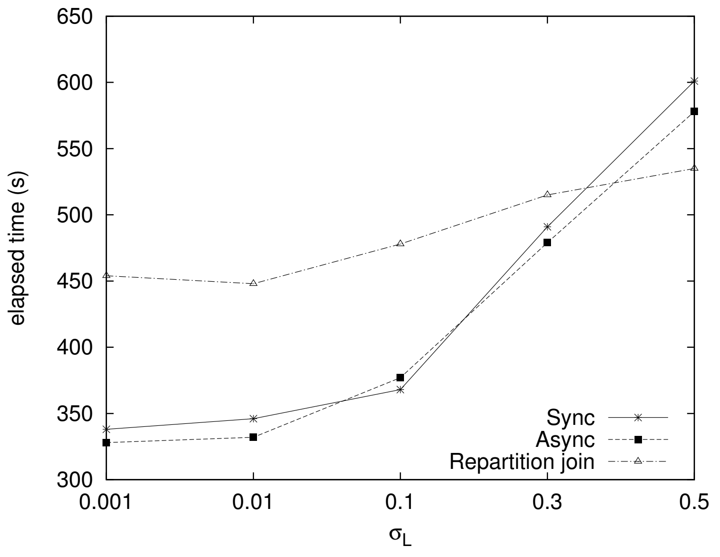
Figure 4.5: Execution times with interval filters varying FPR thresholds

of the repartition join and non-adaptive join with filtering techniques.

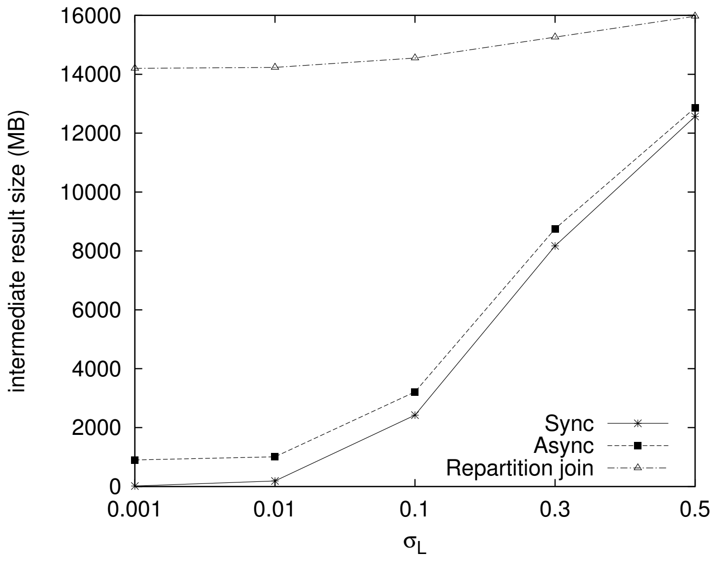
Next, the test query was run with synchronous and asynchronous scheduling. These will be referred to as “Sync” and “Async”, respectively. Figure 4.6(a) shows the execution times of the test queries. Bloom filters were applied and the scale factor was 100. The techniques using filters show better performance than the repartition join when σ_L is small. If more records participate in the join, the performance becomes worse because the number of redundant records that can be filtered out is reduced.

Figure 4.6(b) shows the intermediate result sizes in each test case. Those of asynchronous scheduling vary each time, so the average size was presented in the results. In each case, Async has a slightly larger size than Sync because some map tasks are processed without filtering in the global filter merging phase. Instead, Sync has the additional cost of waiting during the merging phase. The repartition join has the largest size, because it emits all probe input records as intermediate results. The intermediate result sizes increase as σ_L increases, and this leads to an increase in execution times. As the size of the Bloom filters is fixed, the probability of false positives increases with the number of join keys that are inserted into the filters. This is more obvious in the case of a large scale factor.

With the configurations in Figure 4.6(a) and 4.6(b), the asynchronous scheduling shows better performance than the synchronous scheduling, although this is dependent on the size of the intermediate results that are produced in the global filter merging phase. Figure 4.7 shows the execution times with various HDFS block sizes. Likewise, Bloom filters were applied and the scale factor was 100. As the block size is the maximum size of an input split in the map phase, the asynchronous scheduling processes more input records without filtering as the block size increases. In addition, the lengths of the substrings for both tables were set to 30, to clarify the impact of the increase in intermediate result size. In this case,



(a) Execution times

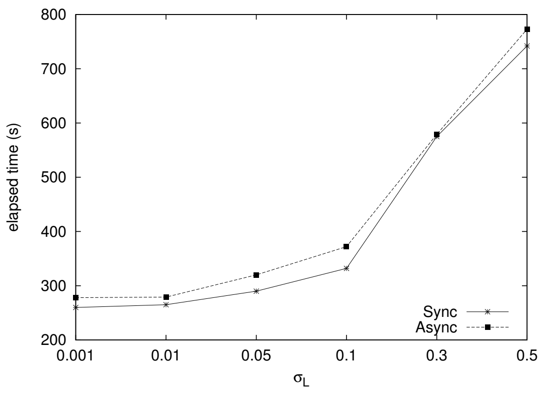


(b) Intermediate result sizes

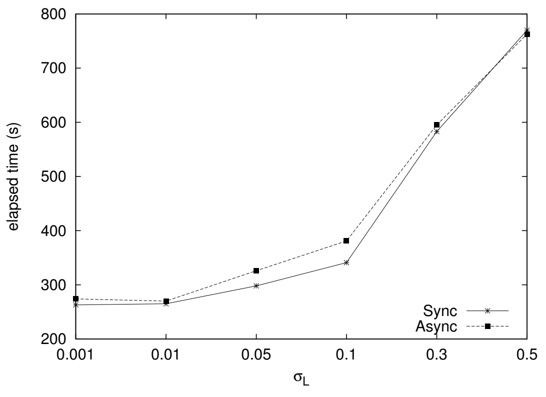
Figure 4.6: Execution times and intermediate result sizes with synchronous and asynchronous scheduling

the synchronous scheduling is better than the asynchronous scheduling when σ_L is small.

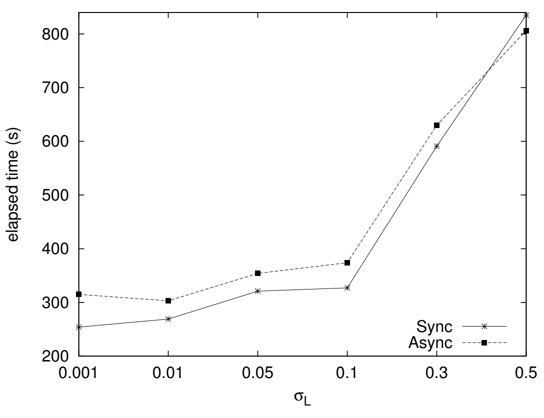
Figure 4.8(a), 4.8(b), and 4.8(c) show the intermediate result sizes corresponding to the cases in Figure 4.7(a), 4.7(b), and 4.7(c). It can be observed that the difference between the intermediate result sizes of both scheduling policies is generally large when σ_L is small, and the gap gets larger as the block size gets larger. Therefore, it can be confirmed that the performance of synchronous and asynchronous scheduling is affected by the intermediate result size, and is dependent on several factors such as the performance of the filters, the input split size, and the maximum number of map tasks, as described in Section 3.2 and 4.2.



(a) 256MB

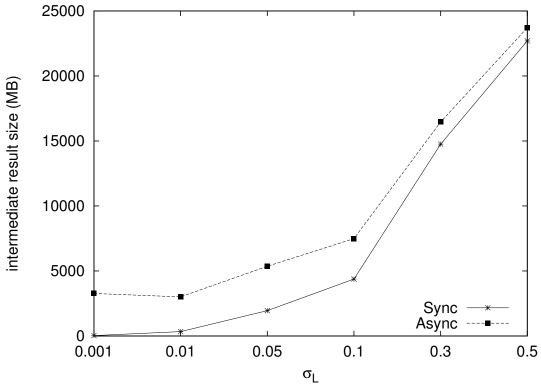


(b) 384MB

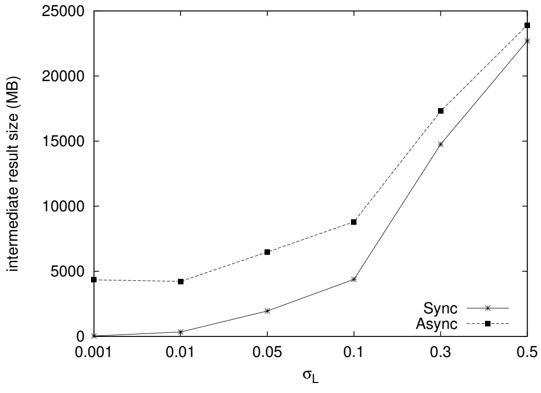


(c) 512MB

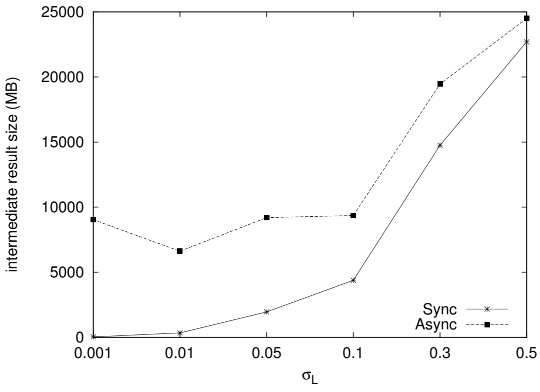
Figure 4.7: Execution times with various HDFS block sizes



(a) 256MB



(b) 384MB



(c) 512MB

Figure 4.8: Intermediate result sizes with various HDFS block sizes

Chapter 5

Multi-way Join Processing in MFR-Join

Joining multiple datasets in MapReduce has been a challenging problem because it may amplify disk and network overhead. Multiple datasets can be joined in the following two ways: (1) using a cascade of two-way (or smaller multi-way) joins, and (2) with a single multi-way join. However, both methods have some drawbacks. A cascade of two-way joins has to write the intermediate join results to the underlying distributed file system, which generally replicates multiple records to ensure high availability and fault tolerance. To process multi-way joins in a single MapReduce job, the map output records have to be replicated multiple times, instead of writing only the final join results to the distributed file system.

As mentioned in Section 2.3.2, previous studies have attempted to improve join performance using filtering techniques [40, 28, 57, 33], including our previous study [31, 32]. These studies focused on reducing the number of map output records that are not joined. This may be more beneficial with multi-way joins. The map output records are replicated multiple times, so filtering out redundant records removes multiple copies of the record in multi-way joins.

An example of basic multi-way join processing in MapReduce has been shown in Figure 2.6, Section 2.4. In this example, three input datasets, i.e., R(a,b), S(b,c), and T(c,d), are joined with two attributes b and c. To join the three datasets simultaneously, some datasets need to be replicated, i.e., R and T in this example. Replication may degrade the join performance, so it is important to reduce the number of redundant records, which are marked with strikethroughs in Figure 2.6. Recent studies have attempted to minimize the number of input records replicated in multi-way joins [6, 25]. They reduce the number by optimizing the replication factor for each join attribute in input datasets. Nevertheless, it still produces a lot of redundant map output records that are replicated.

Multi-way joins can be classified into two types: *common attribute joins* and *distinct attribute joins* [29]. A common attribute join combines datasets based on one or more shared attributes, whereas some relations do not have join attributes in a distinct attribute join. The example shown in Figure 2.6 illustrates a distinct attribute join, because R does not have the attribute c and T does not have the attribute b. In this chapter, the concept of filtering techniques is extended to multi-way joins.

5.1 Applying filters to multi-way joins

This section presents the basic methods used to create and apply filters to common and distinct multi-way joins. To simplify the discussion, joins between three datasets are considered in the following subsections. We then consider the processing of general joins between multiple datasets.

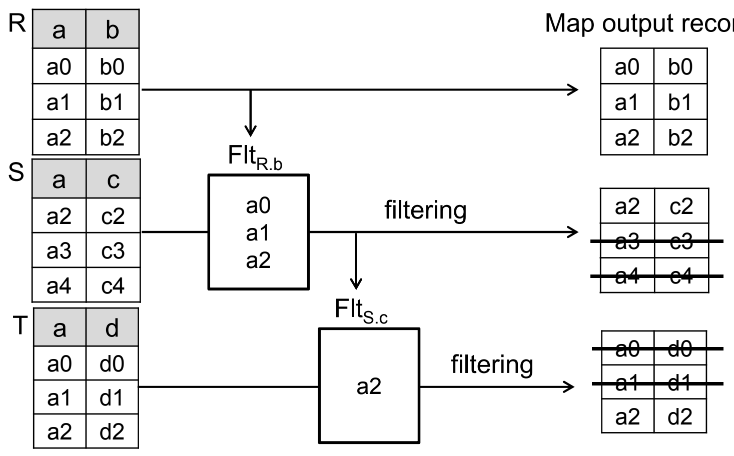


Figure 5.1: Common attribute join

5.1.1 Common Attribute Joins

In common attribute joins, all of the input datasets share join attributes. In these cases, the input records do not need to be replicated and they can be processed in a similar manner to two-way joins. A set of filters is created and probed in turn, depending on the processing order of the input datasets.

Figure 5.1 shows an example of a common attribute join between three input datasets, i.e., R(a,b), S(a,c), and T(a,d), based on the attribute a. Similar to two-way joins, the input records of the first dataset, i.e., R in the figure, are not filtered out and they are used to create a set of filters for the next dataset. The input records of the second dataset, i.e., S in the figure, are processed using the filters and some redundant records can be filtered out. In addition, another set of filters is created using the map output records from S for the next dataset. The map output records are contained in the first set of filters, which means that the second set of filters is automatically the same as the intersection of the filters that are created independently using the first and second dataset. Finally, the third input dataset, i.e., T in the figure, is processed with the second set of filters. Another set of filters does not need to be created because this is the final input dataset for the join attribute.

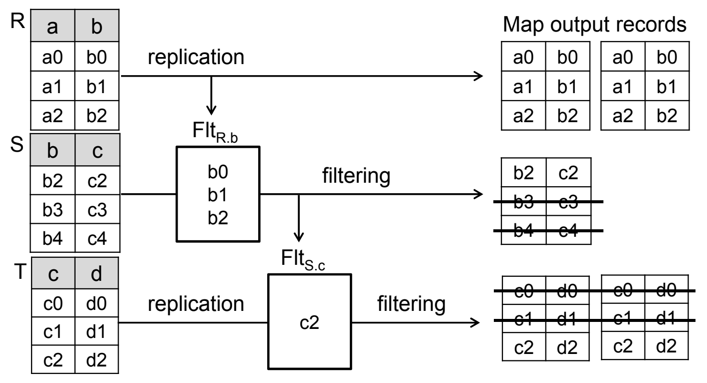
The input datasets are processed in the order of R, S, and T in this example, but any order can be processed in the same way. The join cost depends on the number of input records, the ratio of the joined records, and the false positive rate of the filters, as described in Section 3.2.

5.1.2 Distinct Attribute Joins

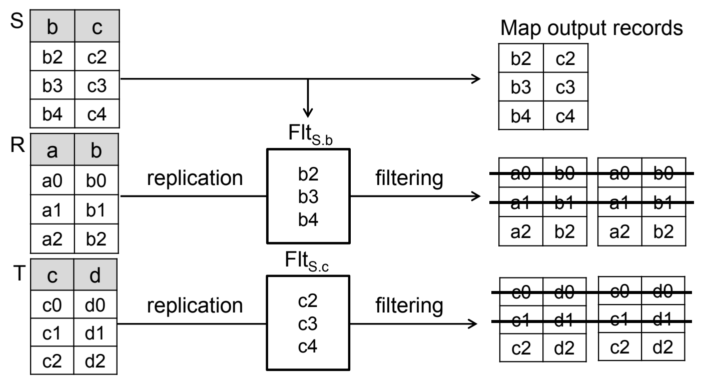
In distinct attribute joins, the input datasets may not have some join attributes. Thus, some of the datasets with missing attributes need to be replicated because their records may be joined to the input records of other datasets with any values of the missing attributes. Let us consider the join example shown in the Figure 2.6, which is a join between three input datasets, i.e., R(a,b), S(b,c), and T(c,d), based on two attributes, i.e., b and c. Assume that R and T are replicated by two reducers in this example. The example join can be processed in 3! = 6 different orders of the input datasets. Depending on the processing order, the filters can be applied in three patterns: *chain*, *star-fact*, and *star-dim*.

Chain

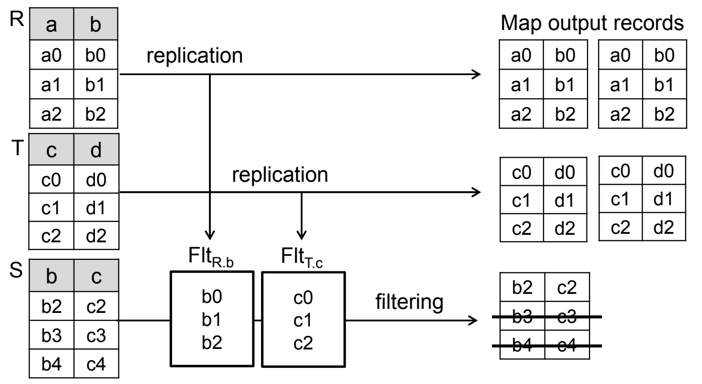
The chain pattern creates and probes filters in turn, in a similar manner to common attribute joins, except that each set of filters is created for a different join attribute. This is analogous to the indirect partitioning method proposed by Kemper et al. [26] Two processing orders correspond to this pattern, i.e., R-S-T and T-S-R. Figure 5.2(a) illustrates an example of a distinct attribute join with the chain pattern. The first dataset R is replicated by reducers and a set of filters is created with the values of the join attribute, b in R. The second dataset S is processed using the filters and some redundant records may be filtered out. Meanwhile,



(a) *Chain* pattern



(b) *Star-fact* pattern



(c) *Star-dim* pattern

Figure 5.2: Distinct attribute joins

another set of filters is created using the map output records from S based on the other join attribute c. Next, the third dataset in the figure, T, is replicated and processed with the second set of filters.

Star-fact

The star-fact pattern creates filters using the dataset with both join attributes and uses the filters to process the other datasets. In database terms, a fact table in a star join is used to create the filters. Two processing orders correspond to this pattern, i.e., S-R-T and S-T-R. Figure 5.2(b) shows an example of a distinct attribute join with the star-fact pattern. The first dataset S, which has both join attributes, is processed and two sets of filters for each join attribute, i.e., b and c, are created. Next, the other datasets, i.e., R and T, are replicated and processed using the set of filters that correspond to the join attribute.

Star-dim

The star-dim pattern creates filters using the datasets with missing join attributes and uses the filters to process the other dataset. In database terms, the dimension tables in a star join are used to create the filters. The remaining two cases, i.e., R-T-S and T-R-S, correspond to this pattern. Figure 5.2(c) shows an example of a distinct attribute join with the star-dim pattern. The first and second datasets, i.e., R and T, are replicated and processed without filters. Each set of filters for the join attributes is created using their join attribute values. Next, the third dataset S is processed using both filters and some redundant records are filtered out. The star-dim pattern appears to be inefficient in this example, but it is efficient if the number of records in the third dataset is much larger than those in the other datasets.

5.1.3 General Multi-way Joins

The basic filtering patterns are presented in Section 5.1.1 and 5.1.2. Multi-way joins of more than three datasets can be processed by combining these patterns. In general, multiple datasets can be joined in any processing order for input datasets using the following rules. For each join attribute,

- Create a set of filters if the dataset is not the last one with the attribute.
- Probe the existing set of filters if the dataset is not the first one with the attribute.

All combinations of these patterns can be summarized using these rules. The filters can be applied in any processing order, but the processing order must be selected carefully because it affects the join cost.

5.1.4 Cost Analysis

The number of intermediate map output records is the most important factor that influences the overall cost. For multi-way joins, the number is affected by the replication factors for each join attribute. Let n_i be the number of records in the i-th input dataset and f_i be the replication factor for the i-th join key. Next, let σ_{i_j} be the ratio of joined records in the i-th dataset relative to a previously processed j-th dataset, and let p_{i_j} be the false positive probability of the previously created filters for the j-th join attributes when the i-th dataset is processed. In addition, let $c(i,j)$ be a binary value function that returns whether the i-th dataset contains the j-th join attribute as follows:

$$c(i,j) = \begin{cases} 1 \text{ , if } i\text{-th dataset contains the } j\text{-th join attribute} \\ 0 \text{ , otherwise} \end{cases} \tag{5.1}$$

Assuming that the attribute values of the input datasets are independent, the number of intermediate map output records n^f_{inter} in a multi-way join between n datasets based on k join attributes can be expressed as follows:

$$n^f_{inter} = \sum_{i=1}^{k} (f_i \cdot n_i \cdot S_i + f_i \cdot n_i \cdot (1 - S_i) \cdot \prod_{j=1}^{k} (p_{i\_j} \cdot c(i,j))) \tag{5.2}$$

where

$$S_i = \begin{cases} 1 & \text{, if } i = 1 \\ \prod_{j=1}^{i-1} \sigma_{i\_j} & \text{, otherwise} \end{cases} \quad \text{and} \quad \prod_{i=1}^{k} f_i = (\#\text{ of reducers})$$

We need to find the replication factors and processing order for input datasets that minimizes the number of intermediate map output records. Note that the factors f_i, S_i, and p_{i_j} depend on the processing order of the input datasets. These equations can be used to select the processing order and to estimate the join cost, but there may be a large search space if the numbers of reducers and the input datasets are large. In these cases, the factors have to be selected using heuristics. For example, the replication factors can be computed using the method proposed by Afrati and Ullman [6], which does not consider filters, or they can be pre-defined by users, or determined by optimizer modules. The brute force approach was used in the experiments conducted with ten reducers in the present study.

5.2 Implementation Details

The two major issues when processing multi-way joins simultaneously in MFR-Join are replicating the records for the corresponding reducers and processing multiple join attributes for filtering. The following subsections describe the specific implementation details that address these issues.

5.2.1 Partition Assignment

For each input dataset, its corresponding reducers for replication are determined as shown in Figure 2.7. The replication of input records for their corresponding reducers can be implemented in a similar manner to the data partitioning method described by Zhang et al. [55]. Algorithm 1 demonstrates how to find the target reducers that correspond to an input record. Depending on the number of join attributes *n*, we may assume that there is an *n*-dimensional space with integer coordinates, where each dimension represents each join attribute and a position represents a reducer. Note that *n* is the number of join attributes, rather than the number of input datasets, which was the case in a previous study [55] that aimed to process theta joins. Then, the corresponding positions of an input record can be obtained by partitioning the join attribute values of the record in a range from zero to (the corresponding replication factor - 1). A coordinate for a missing join attribute can be expressed using a special character, i.e., '*'. This indicates that the record corresponds to the reducers with all possible values for the coordinate. Next, the positions are converted into integer identifiers of the reducer by adding up the values of each position, which are multiplied by the replication factors for the preceding dimensions. Algorithm 2 and 3 show the pseudo-codes for the conversion process.

Now, we consider an example of three-way joins between R(a,b), S(b,c), and T(c,d) using the two join attributes shown in Figure 2.7. Assume that the number of reducers is nine and that there are three replication factors for both R and T. The positions that correspond to each record of R, S, and T in the figure are <1,*>, <1,2>, and <*,2>, respectively. <1,*>represents the positions <1,0>, <1,1>, and <1,2>. If each reducer is assigned with an integer identifier from zero to eight incrementally, starting from the top-left cell in a vertical direction, the identifiers of the reducers that correspond to the records for R are

Algorithm 1 Finding target reducers

```
    ▷ $n$: the number of join attributes
    ▷ $joinAttr$: an array of join attribute values (some values may be missing)
    ▷ $repl$: an array of replication factors for each join attribute
 1: procedure FINDTARGETREDUCERS($joinAttr[1..n], repl[1..n]$)
 2:     $reducerList \leftarrow \emptyset$
 3:     $coord \leftarrow \emptyset$
 4:     for $i = 1$ to $n$ do
 5:         if $joinAttr[i]$ is not null then
                        ▷ $GetPartition()$: returns a number in the range [0 and (repl[i]-1)]
 6:             $coord[i] \leftarrow GetPartition(joinAttr[i], repl[i])$
 7:         else
 8:             $coord[i] \leftarrow 0$
 9:         end if
10:     end for
11:     while $true$ do
12:         $rid \leftarrow CoordToReducer(coord, repl)$
13:         $reducerList \leftarrow reducerList \cup rid$
14:         if $IncrCoord(coord, joinAttr, repl) = false$ then
15:             break
16:         end if
17:     end while
18:     return $reducerList$
19: end procedure
```

Algorithm 2 Converting a coordinate to a reducer id

```
1: procedure CoordToReducer($coord[1..n], repl[1..n]$)
2:     $reducer \leftarrow 0$
3:     $base \leftarrow 1$
4:     for $i = 1$ to $n$ do
5:         $reducer \leftarrow reducer + base * coord[i]$
6:         $base \leftarrow base * repl[i]$
7:     end for
8:     return $reducer$
9: end procedure
```

Algorithm 3 Increasing a coordinate

```
1: procedure IncrCoord($coord[1..n], joinAttr[1..n], repl[1..n]$)
2:     for $i = 1$ to $n$ do
3:         if $joinAttr[i]$ is not null then
4:             continue
5:         end if
6:         $coord[i] \leftarrow coord[i] + 1$
7:         if $coord[i] < repl[i]$ then
8:             return $true$
9:         end if
10:        $coord[i] \leftarrow 0$
11:    end for
12:    return $false$
13: end procedure
```

1, 4, and 7. Similarly, because <*,2>represents the positions <0,2>, <1,2>, and <2,2>, the identifiers of the reducers that correspond to the records of T are 4, 5, 6, and 7. the identifiers of the reducers that correspond to the records of T are 6, 7, and 8. The position of the record of S is <1,2>, so the identifier of its corresponding reducers is 7. Thus, these records are gathered and joined by the reducer with the identifier 7.

5.2.2 MapReduce Functions

A prototype MFR-Join framework has been implemented to create and probe filters using the keys of map output records. To process distinct attribute multi-way joins with multiple join attributes, the keys need to be separated with a delimiter, which is configured using the additional parameter `mapred.filter.key.delimiter`. The target reducers for a record can be found using Algorithm 1, which is described in Section 5.2.1. Algorithm 4 is the pseudo-code for the map function used in multi-way joins.

Algorithm 4 Map function

```
   ▷ value: an input record from the i-th dataset
   ▷ repl: replication factors that are pre-computed or pre-defined in the init phase
1: procedure MAP(key, value)
2:     extract the join attribute values joinAttr[1..n] by parsing the input record value
                                                    ▷ ||: concatenation, #: delimiter
3:     joinAttrKey ← joinAttr[1] || # || .. || # || joinAttr[n]
4:     reducerList ← FindTargetReducers(joinAttr, repl)
5:     for each reducer in reducerList do
                                                    ▷ tag: the dataset id of the record
6:         Emit((joinAttrKey, reducer), (value, tag))
7:     end for
8: end procedure
```

The records generated by the map function are then processed by the MFR-Join framework, as explained in Section 5.1. Some redundant records will be filtered out, depending on the processing order of the input datasets. The map output records that passed the filters have been gathered in the corresponding reducers by their reducer identifiers. Using the reduce function, the records are classified based on the tag representing their original dataset and they are joined with traditional join algorithms. Algorithm 5 is the pseudo-code for the reduce function in multi-way joins.

Algorithm 5 Reduce function

```
   ▷ $values$: intermediate records $(record, datasetId)$ with the same reducer id
1: procedure REDUCE($key, values$)
   ▷ $recordList$: lists for buffering the intermediate records based on their dataset id
2:     $recordList \leftarrow \emptyset$
3:     for each $value$ in $values$ do
4:         $dsid \leftarrow value.tag$
5:         $recordList[dsid] \leftarrow recordList[dsid] \cup value.record$
6:     end for
       ▷ $Join(recordList)$: returns the join results between the records in each record list
7:     $Emit(Join(recordList))$
8: end procedure
```

5.3 Evaluation

This section presents the experimental results for common and distinct attribute joins. The experimental environment is almost same as in Section 3.3.1 and 4.3.1, except the number of reduce tasks that are run simulateneously in tasktrackers. All of the experiments were run on a cluster of 11 machines, which comprised one jobtracker and 10 tasktrackers. Each machine had a 3.1 GHz quad-core CPU, 4 GB RAM, and a 2 TB hard disk. The operating system was 32-bit Ubuntu 10.10 and the Java version used was 1.6.0_26. The Hadoop distributed file system (HDFS) was set to use 128 MB blocks and to replicate them three times. Each tasktracker could run three map tasks and one reduce task simultaneously. The I/O buffer was set to 128 KB and the memory used to sort the data was set to 200 MB.

5.3.1 Common Attribute Joins

For common attribute joins, TPC-H benchmark [4] datasets was used with a scale factor of 100. The scale factor was the size of the entire database in gigabytes. A join was performed between three tables in the database, i.e., `part`, `partsupp`, and `lineitem`, which had a common attribute, i.e., `partkey`. The sizes of the datasets are shown in Table 5.1.

Table 5.1: Test datasets for common attribute joins

Table	# of records	Size	# of records satisfying selection predicate
part	20 M	2.3 GB	1.08 M
partsupp	80 M	12 GB	80 M * σ_{ps}
lineitem	600 M	75 GB	600 M (no predicate)

The test query was extracted from TPC-H Q9 and it could be expressed in SQL-like syntax as follows:

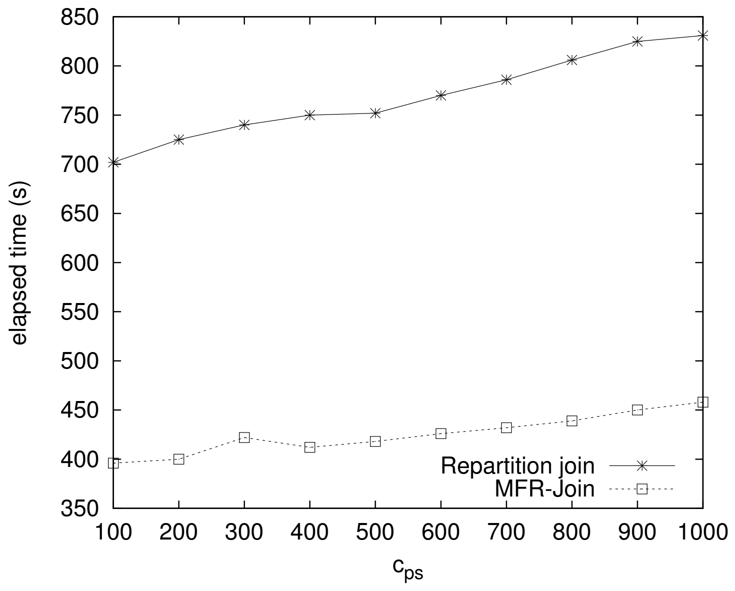
```
SELECT l.orderkey, l.partkey,
       l.suppkey, ps.suppkey,
       l.extendedprice * (1 - l.discount)
       - ps.supplycost * l.quantity as profit
FROM part p, partsupp ps, lineitem l
WHERE p.partkey = l.partkey
  AND ps.partkey = l.partkey
  AND p.name like ‘%green%’
  AND ps.supplycost < c_ps
```

To control the amount of joined records, the selection predicate `ps.supplycost < ` c_{ps} was added to the query. The attribute `ps.supplycost` had a decimal value in the range of 1.0 to 1000.0. The predicate value c_{ps} in the predicate with increments of one hundred. Thus, the ratio of records that satisfied the predicate in `partsupp` σ_{ps} was changed by about 10 %. The performance of the proposed method was compared with a repartition join without filters, because common attribute joins can be processed without replication in a single MapReduce job. In MFR-Join, simple hash filters with a size of 8 Mb were used and the input datasets were processed in the order: `part`, `partsupp`, and `lineitem`.

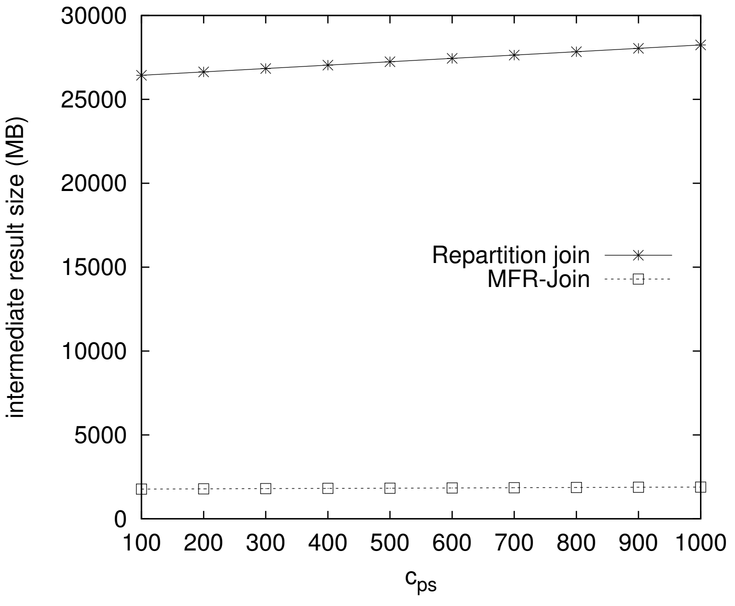
Figure 5.3 shows the execution times and the sizes of the intermediate results for the test queries. The results showed that the proposed method significantly outperformed the repartition join for all of the test cases in Figure 5.3(a). This is because large numbers of redundant intermediate results from the `lineitem` dataset are filtered out, as shown in Figure 5.3(b). The `lineitem` dataset has no selection predicate, so the repartition join has to generate the entire dataset as intermediate results.

5.3.2 Distinct attribute joins

For distinct attribute joins, the TPC-H benchmark [4] datasets were also used, but with a scale factor of 300. The following join query, which was extracted from TPC-H Q2, was performed between the following five tables: `nation`, `region`, `supplier`, `part`, and `partsupp`, where the sizes are shown in Table 5.2. The two tables, `nation` and `region`, contained just a few records, so the joins of the tables were treated as in-memory hashing.



(a) Execution time



(b) Intermediate result sizes

Figure 5.3: Performance of common attribute joins

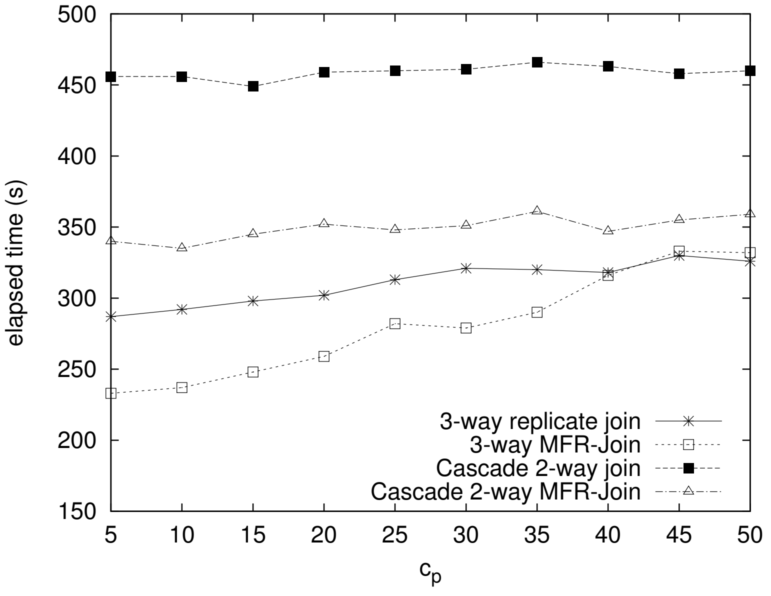
Table 5.2: Test datasets for distinct attribute joins

Table	# of records	Size	# of records satisfying selection predicate
supplier	3 M	0.4 GB	0.6 M
part	60 M	6.9 GB	60 M * 0.2 * σ_p
partsupp	240 M	35 GB	240 M (no predicate)

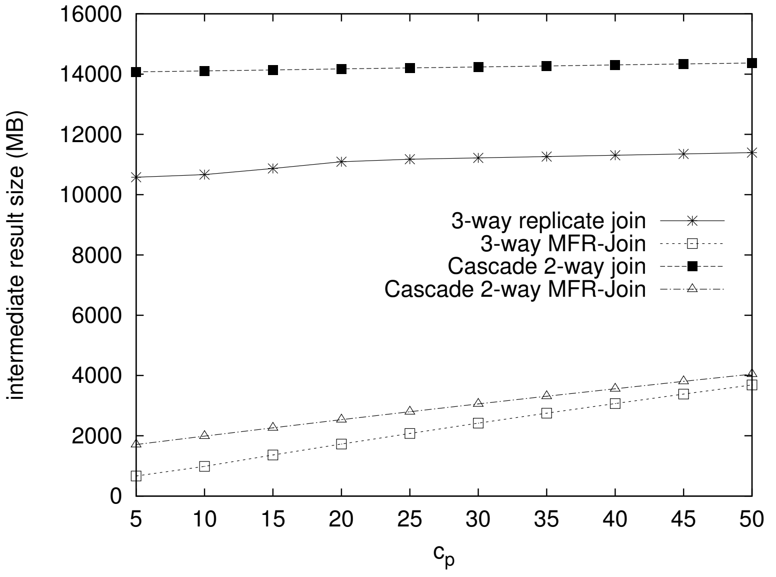
```
SELECT s.acctbal, s.name, n.name,
       p.partkey, ps.supplycost, p.mfgr,
       s.address, s.phone, s.comment
FROM nation n, region r, supplier s,
     part p, partsupp ps
WHERE n.regionkey = r.regionkey
  AND r.name = 'EUROPE'
  AND s.nationkey = n.nationkey
  AND s.suppkey = ps.suppkey
  AND p.partkey = ps.partkey
  AND p.type like '%BRASS'
  AND p.size <= c_p
```

Similar to the test query for common attribute joins, A selection predicate, `p.size <=` c_p, was added to control the amount of joined records. The attribute `p.size` had an integer value in the range of 1 to 50 and the predicate value c_p was changed with increments of five. Thus, the ratio of records that satisfied the predicate in `part` σ_p was changed by about 10 %. Distinct attribute joins required the replication of some input datasets, so the best from the results using all possible combinations of the replication factors was selected.

The performance of the proposed multi-way join method (denoted as 3-way MFR-Join) was compared with that of the basic multi-way join (denoted as 3-way replicate join), and with that of the cascade of two-way joins with and without filters (denoted as Cascade 2-way



(a) Execution time

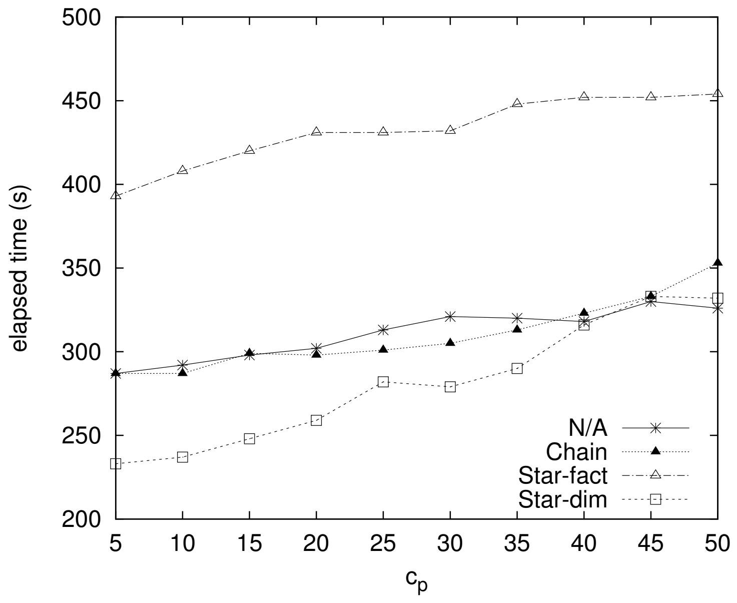


(b) Intermediate result sizes

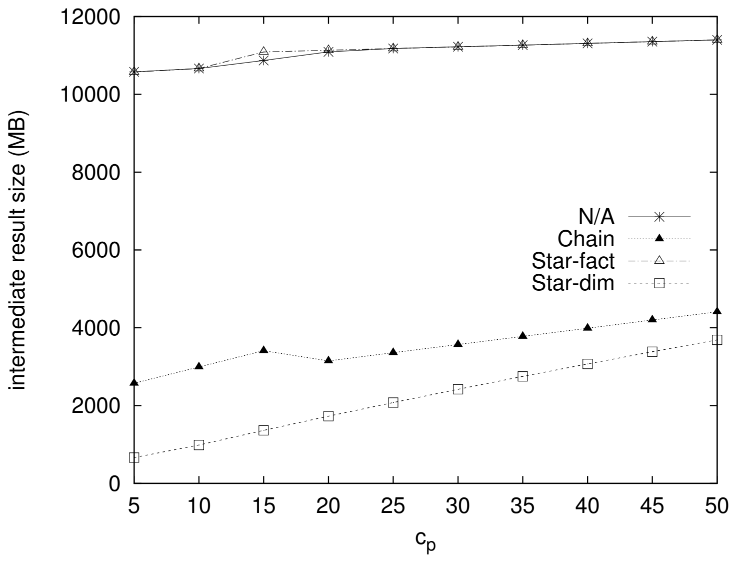
Figure 5.4: Performance of distinct attribute joins

MFR-Join and Cascade 2-way join, respectively). Simple hash filters with a size of 8 Mb were also used for the multi-way join and the cascade of two-way MFR-Join. In the two-way joins, `supplier` and `partsupp` were joined first, before the intermediate join results and `part` were joined. Figure 5.4 shows the execution times and intermediate result sizes for the test queries. The results of the three-way MFR-Join with the star-dim filtering pattern had the best performance with the queries. The multi-way joins outperformed two-way joins for the test queries shown in Figure 5.4(a). The cascade of two-way joins processes the join queries in two MapReduce jobs, which means that they must write the intermediate results of the first join to HDFS, before reading them from HDFS. Furthermore, there are additional costs of initializing and cleaning up a job. In two-way and multi-way joins, the MFR-Join methods with filters delivered better performance than the basic join methods without filters. This was because large numbers of redundant intermediate results from the `partsupp` dataset, which had no selection predicate, were filtered out by the MFR-Join, as shown in Figure 5.4(b).

Figure 5.5 shows the experimental results obtained with the three-way MFR-Join using the filtering pattern. In Figure 5.5(a), the star-dim pattern delivered the best performance compared with the others using the test queries. This was because the fact table, `partsupp`, was much larger than the dimension tables, `supplier` and `part`, in the test datasets. As shown in Figure 5.5(b), the number of intermediate results decreased most with the star-dim pattern. In particular, it should be noted that the star-fact pattern did not decrease the number of intermediate results at all. The join attributes in the queries were the foreign keys in the databases. Furthermore, no selection predicate was specified for the `partsupp` dataset in the test queries, so the records in `partsupp` did not play a role in filtering. The increase in the amount with a c_p value of 15 was caused by the difference in the replication factors



(a) Execution time



(b) Intermediate result sizes

Figure 5.5: Performance of distinct attribute joins with the filtering pattern

with the best execution time. Thus, the execution times were increased slightly by creating, merging, and probing the filters needlessly. It is considered that each filtering pattern may be effective in different cases, depending on the sizes of the input datasets and the ratios of joined records. Therefore, it is important to apply the filters using an advantageous pattern. If the proposed methods are combined with upper-layer data warehouse systems, such as Hive [51], this could be determined using its optimizer module based on statistical information related to the stored tables. This task will be addressed in future work.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Issues on join processing in the MapReduce framework were covered in this dissertation. The proposed approach focused on reducing the number of redundant intermediate results using filtering techniques. In particular, the concept of filtering techniques was adapted within a single MapReduce job to improve the performance of equi-join queries.

MFR-Join was described in Chapter 3, which is a general join framework with filtering techniques in MapReduce. MFR-Join improves the join performance by filtering out redundant intermediate results using various filtering techniques, which can be plugged in MFR-Join so long as they support specific operations. MFR-Join produced two design changes compared to the original Hadoop. First, map tasks are assigned based on the order of the dataset. Second, filters are constructed in a distributed manner. In addition, a cost model for estimation of the total join cost was described. In the experiments that are performed with various sizes of TPC-H dataset on an 11-node cluster, it was shown that MFR-Join significantly improved the query execution time, when small portions of input datasets are joined.

An adaptive join processing method with filtering techniques was presented in Chapter 4. It estimates the performance of filters in terms of the false positive rates, and disables filters with false positive rates greater than a user-configured threshold. To restrain the negative influence from the waiting time during the filter merging phase, two scheduling policies were also presented: synchronous and asynchronous scheduling. Experimental results showed that the proposed methods provided stable performance, which was similar to or better than that of the repartition join and the non-adaptive join.

A method for applying filters to multi-way joins was proposed in Chapter 5. A set of filters is created and applied in turn to achieve common attribute joins and multiple sets of filters are used in various patterns, which depend on the processing order of input datasets, thereby producing distinct attribute joins. Specific details for assigning reducers and writing map/reduce functions were also described. Experimental results showed that the proposed approach significantly improved the execution time by reducing the amounts of intermediate results, compared to basic multi-way joins and the cascade of two-way joins.

6.2 Future Work

There remains a range of issues to be addressed by future research. Here, a few potential issues are briefly pointed out.

6.2.1 Integration with Data Warehouse Systems

There were some limitations to apply filters effectively in the distributed processing framework layer, which simply processes raw data, because the statistical information available is insufficient. To complement this defect, adaptive join methods were proposed in Chapter 4. However, filtering techniques can be applied in more effective ways if it is used to process accumulated data in data warehouse systems based on additional statistics available. For instance, in version 0.9.0 of Hive, table and partition statistics are maintained, such as the number of rows, the number of files, and the size in bytes. The size of intermediate join results is determined using those statistics in the query plan generation. Moreover, further research is being carried out to exploit column level statistics, such as the number of distinct values and histograms. Cost models for estimation were presented in each chapter, so optimizer modules for the data warehouse systems can be developed to determine several parameters for filtering based on estimated cost, such as whether to use filters, the type and size of filters, and the processing order of input datasets. Adaptive joins will then play a role as an insurance in case of incorrect estimation.

6.2.2 Join-based Applications

Since the join operation is an essential operation for data analysis, MFR-Join can be used in various application systems. For example, graph pattern matching queries can be processed based on joins [38, 24]. Figure 6.1 shows an MapReduce execution plan to process the following graph query, which is expressed in SPARQL.

```
SELECT ?X ?Y
WHERE {
  ?X :type Professor .
  ?X :worksFor CS .
  ?Y :advisor ?X .
}
```

To find subgraphs that match a given query pattern from very large graph data, intermediate results for each edge in the query need to be joined. It is expected that MFR-Join will improve the performance for some types of graph query patterns, depending on the distribution

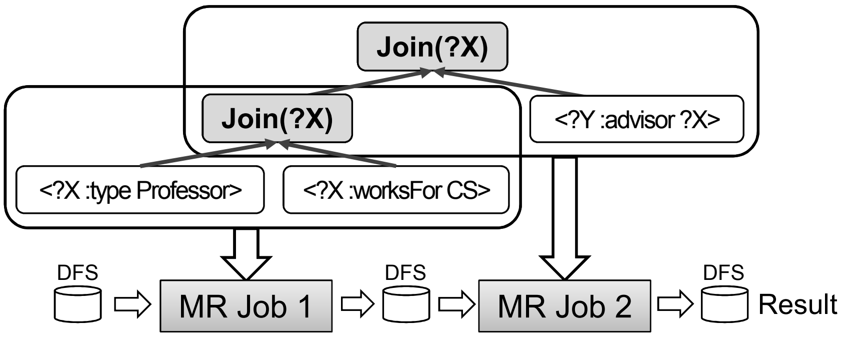


Figure 6.1: Join-based graph pattern matching

of nodes and edges in the graph data.

Another example is Semantic Web reasoning. Joins can be used to infer new knowledge by finding propositions that satisfy specific inference rules [52]. Besides, it is believed that various applications can be improved by MFR-Join. It would be desirable to ascertain if MFR-Join is useful for many other applications.

6.2.3 Improving Scalability

On the experimental side, this study was conducted on a cluster with 11 nodes, which are regarded as a small number of nodes. It is necessary to improve the proposed approach to run on a cluster with hundreds or thousands nodes. A large number of filters will be used on very large clusters, but the overhead of merging the filters tends to increase as the number of nodes increases, as described in Section 3.1.3. Therefore, the filters have to be merged in an efficient way such as hierarchical merging. The merging time can be increased if straggler nodes do not finish their map tasks on a build input. It will be useful to launch backup tasks for each input dataset in its map phase. In addition, the memory size that is occupied by the filters have to be controlled because all of the filters cannot be loaded due to out of memory error.

It is to be hoped that this study will help identify current issues and encourage additional research on analysis of large-scale heterogeneous data.

References

[1] Cluster setup. `http://hadoop.apache.org/docs/stable/cluster_setup.html`.

[2] Hadoop. `http://hadoop.apache.org/`.

[3] Hive. `http://hive.apache.org/`.

[4] TPC-H benchmark. `http://www.tpc.org/tpch/`.

[5] Teradata: Dbc/1012 data base computer concepts & facilities. Teradata Corp. Document No. C02-0001-00, 1983.

[6] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT'10, pages 99–110, 2010.

[7] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. volume 23, pages 1282–1298, 2011.

[8] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012.

[9] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM*, 28(1):25–40, 1981.

[10] 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. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD'10, pages 975–986, 2010.

[11] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[12] B. Brown, M. Chui, and J. Manyika. Are you ready for the era of 'big data'? MacKinsey Quarterly, 2011.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 137–150, 2004.

[14] J. Dean and S. Ghemawat. System and method for efficient large-scale data processing. Patent, US 7650331, 2010.

[15] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD'84, pages 1–8, 1984.

[16] C. Eaton, D. Deroos, T. Deutsch, G. Lapis, and P. Zikopoulos. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. The McGraw-Hill Companies, 2012.

[17] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, SIGMOD'78, pages 169–180, 1978.

[18] A. Espinosa, P. Hernandez, J. C. Moure, J. Protasio, and A. Ripoll. Analysis and improvement of map-reduce data distribution in read mapping applications. *The Journal of Supercomputing*, 62(3):1305–1317, 2012.

[19] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[20] A. Gruenheid, E. Omiecinski, and L. Mark. Query optimization using column statistics in hive. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, IDEAS'11, pages 97–105, 2011.

[21] R. Gupta, H. Gupta, U. Nambiar, and M. Mohania. Efficiently querying archived data using hadoop. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM'10, pages 1301–1304, 2010.

[22] F. Hao, M. Kodialam, and T. V.Lakshman. Building high accuracy bloom filters using partitioned hashing. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 277–288, 2007.

[23] H. Herodotou. Hadoop performance models. Technical Report CS-2011-05, Duke University. `http://www.cs.duke.edu/starfish/files/hadoop-models.pdf`, 2011.

[24] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1312–1327, 2011.

[25] D. Jiang, A. K. H. Tung, and G. Chen. Map-join-reduce: Toward scalable and efficient data analysis on large clusters. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1299–1311, 2011.

[26] A. Kemper, D. Kossmann, and C. Wiesner. Generalized hash teams for join and group-by. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB'99, pages 30–41, 1999.

[27] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[28] P. Koutris. Bloom filters in distributed query execution. University of Washington. `http://www.cs.washington.edu/education/courses/cse544/11wi/projects/koutris.pdf`, 2011.

[29] R. Lawrence. Using slice join for efficient evaluation of multi-way joins. *Data & Knowledge Engineering*, 67(1):118–139, 2008.

[30] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: A survey. *ACM SIGMOD Record*, 40(4):11–20, 2011.

[31] T. Lee, K. Kim, and H.-J. Kim. Join processing using bloom filter in mapreduce. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, RACS'12, pages 100–105, 2012.

[32] T. Lee, K. Kim, and H.-J. Kim. Exploiting bloom filters for efficient joins in mapreduce. *Information — An International Interdisciplinary Journal*, 16(8(A)):5869–5885, 2013.

[33] W. Li, K. Huang, D. Zhang, and Z. Qin. Accurate counting bloom filters for large-scale data processing. *Mathematical Problems in Engineering*, 2013(Article ID 516298):11, 2013.

[34] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB'86, pages 149–159, 1986.

[35] Market Analysis. Hadoop-mapreduce market forecast 2013-2018. `http://www.marketanalysis.com/?p=279`, 2012.

[36] Markets and Markets. Big data market by types (hardware; software; services; bdaas - haas; analytics; visualization as service); by software (hadoop, big data analytics and databases, system software (imdb, imc): Worldwide forecasts & analysis (2013 - 2018). `http://www.marketsandmarkets.com/Market-Reports/big-data-market-1068.html`, 2013.

[37] L. Michael, W. Nejdl, O. Papapetrou, and W. Siberski. Improving distributed join efficiency with extended bloom filter operations. In *Proceedings of the 21st International Conference on Advanced Networking and Applications*, AINA'07, pages 187–194, 2007.

[38] J. Myung, J. Yeon, and S. goo Lee. Sparql basic graph pattern processing with iterative mapreduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC '10, pages 6:1–6:6, 2010.

[39] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD'11, pages 949–960, 2011.

[40] K. Palla. A comparative analysis of join algorithms using the hadoop map/reduce framework. Master's thesis, University of Edinburgh, 2009.

[41] O. Papapetrou, W. Siberski, and W. Nejdl. Cardinality estimation and dynamic length adaptation for bloom filters. *Distributed and Parallel Databases*, 28(2-3):119–156, 2010.

[42] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD'09, pages 165–178, 2009.

[43] Y. Qiao, T. Li, and S. Chen. One memory access bloom filters and their generalization. In *Proceedings of the 2011 IEEE INFOCOM*, pages 1745–1753, 2011.

[44] R. Quislant, E. Gutierrez, O. Plata, and E. L. Zapata. Interval filter: A locality-aware alternative to bloom filters for hardware membership queries by interval classification. In *Proceedings of the 11th International Conference on Intelligent Data Engineering and Automated Learning*, IDEAL'10, pages 162–169, 2010.

[45] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD'89, pages 110–121, 1989.

[46] A. Segev. Optimization of join operations in horizontally partitioned database systems. *ACM Transactions on Database Systems*, 11(1):48–80, 1986.

[47] D. Shasha and T.-L. Wang. Optimizing equijoin queries in distributed databases where relations are hash partitioned. *ACM Transactions on Database Systems*, 16(2):279–308, 1991.

[48] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, sixth edition, 2010.

[49] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys and Tutorials*, 14(1):131–155, 2012.

[50] A. Thusoo, S. Antony, N. Jain, R. Murthy, Z. Shao, D. Borthakur, J. S. Sarma, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD'10, pages 1013–1020, 2010.

[51] 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. In *Proceedings of the 26th IEEE International Conference on Data Engineering*, ICDE'10, pages 996–1005, 2010.

[52] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable distributed reasoning using mapreduce. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 634–649, 2009.

[53] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., second edition, 2011.

[54] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD'07, pages 1029–1040, 2007.

[55] C. Zhang, J. Li, L. Wu, M. Lin, and W. Liu. Sej: An even approach to multiway theta-joins using mapreduce. In *2012 Second International Conference on Cloud and Green Computing (CGC)*, pages 73–80, 2012.

[56] C. Zhang, L. Wu, and J. Li. Optimizing distributed joins with bloom filters using mapreduce. In *Computer Applications for Graphics, Grid Computing, and Industrial Environment*, pages 88–95, 2012.

[57] C. Zhang, L. Wu, and J. Li. Efficient processing distributed joins with bloomfilter using mapreduce. *International Journal of Grid and Distributed Computing*, 6(3):43–58, 2013.

초 록

조인 연산은 데이터 분석을 위한 필수 연산 중 하나이다. 각기 다른 곳에서 수집된 이질적인 데이터를 분석하려면 큰 데이터셋을 조인해야 하기 때문이다. 맵리듀스(MapReduce)는 대규모 데이터 분석에 매우 유용한 프레임워크이지만, 여러 데이터셋을 조인하는 데에는 적합하지 않다. 조인되는 레코드의 크기에 상관 없이 많은 수의 불필요한 중간 결과를 생산하는 탓이다. 조인 성능을 향상하기 위해 기존의 몇몇 방법을 이용해 왔으나 특정 상황에서만 사용 가능하거나 여러 맵리듀스 잡(job)을 요한다. 이러한 문제를 경감하기 위해 본 학위 논문에서는 맵리듀스에서 필터링 기법을 사용하여 동등 조인(equi-join)을 처리하는 일반적인 조인 프레임워크인 MFR-Join을 제안한다. MFR-Join은 단일 맵리듀스 잡에서 맵 단계에서 필터를 적용하여 불필요한 중간 결과를 걸러 낸다. 이를 이루기 위해 맵리듀스 프레임워크를 두 가지 수정한다. 첫째, 맵 태스크를 입력 데이터셋의 처리 순서에 따라 스케쥴한다. 둘째, 필터를 동적으로 데이터셋들의 조인 키를 가지고 분산 방식으로 생성한다. 필요한 특정 연산을 지원하는 다양한 필터링 기법을 MFR-Join에 플러그인 하여 사용할 수 있다. 필터를 사용할 때의 조인 처리 성능이 사용하지 않을 때보다 더 나빠질 경우를 대비하여 적응적 조인 처리 방법도 제안한다. 양성 오류율의 측면에서 필터의 성능을 예측하여 그에 따라 필터를 적용한다. 이와 더불어 동기 및 비동기 스케쥴링의 두 가지 맵 태스크 스케쥴링 방식을 제시한다. 필터링 기법을 이용하는 착상을 다중 조인(multi-way join)까지 확장하여 공통 속성 조인(common attribute join)과 차별 속성 조인(distinct attribute join)의 두 가지 형태의 다중 조인에 대한 필터 적용 방법을 제안

한다. 실험 결과를 통해 제안한 방법이 입력 데이터셋에서 일부만 조인될 때 기존의 조인 알고리즘들보다 나은 성능을 보이고 중간 결과의 크기를 줄인다는 점을 밝힌다.

주요어: *조인 처리, 적응적 조인, 조인 필터, 맵리듀스, 하둡*

학번: 2004-21603