Research and optimization of the Bloom filter algorithm in Hadoop

Bing Dong
Abstract

Research and optimization of the Bloom filter algorithm in Hadoop

Bing Dong

An increasing number of enterprises have the need of transferring data from a traditional database to a cloud-computing system. Big data in Teradata (a data warehouse) often needs to be transferred to Hadoop, a distributed system, for further computing and analysis. However, if data stored in Teradata is not synced with Hadoop, e.g. due to data loss during the communication, sync and copy process, it will cause the data to disaccord. A survey shows that except for the algorithm provided by Hadoop, the Bloom filter algorithm can be a good choice for data reconciliation. MD5 hash technology is applied to reduce the amount of data transmission. In the experiments, data from both sides was compared using a Bloom filter. If there was any data loss during the process, different primary keys could be found. The result can be used to track the change of the original data.

During this thesis project, an experimental system using the Mapreduce mode of Hadoop was implemented. For the implementation, real data was used and the parameters were adjustable to analyze different schemes (Basic join, CBF, SBF and IBF).

Basic knowledge and the key technology of the Bloom filter algorithm are introduced initially. Then the thesis systematically expounds the existing Bloom filter algorithms and the pros and cons of each. It also introduces the principle of the Mapreduce program in Hadoop. In the next part, three schemes, all in concordance with the requirements are introduced in detail. Then in the 4th phase, the implementation of schemes in Hadoop as well as the design and implementation of the testing system are introduced. In the 5th phase, testing and analysis of each scheme is carried out. The feasibility of the schemes is analyzed with respect to performance and cost using experimental data. Finally, conclusions and ideas for further improvement of the Bloom filter are presented.
Acknowledgement

First and foremost, I would like to thank my supervisor Dr. Liu, Qin, I appreciated your guidance to my thesis and the comments to my report. In addition, I very much appreciated the help of my reviewer Ivan Christoff, thank you for being my reviewer and your advice.

I would like to thank my family, it is because of their financial and psychological support that I was able to study in Sweden. And I would like to thank my friends and all the people who are involved in the Sino-Swedish master program, Yang, Xiaowen, Anders Berglund and all the others both in Sweden and in China, for providing us such an opportunity to see the world, to experience different culture.

I would like to thank my friends and tutors in Tongji University, thank you for providing the opportunity to live and learn from you. The three years of my master study is one of the best time in my life.
Contents

1 Introduction .............................................................................................................. 7
   1.1 Background .............................................................................................................. 7
   1.2 Study objective ........................................................................................................ 8
   1.3 Content arrangement .............................................................................................. 8

2 Related Technology about Bloom filter ................................................................. 11
   2.1 Principle of Bloom filter algorithm ........................................................................... 11
   2.2 Introduction on extensions of Bloom Filter ........................................................... 12
       2.2.1 Counting Bloom Filter ..................................................................................... 12
       2.2.2 Scalable Bloom Filter ...................................................................................... 13
       2.2.3 Invertible Bloom Filter ..................................................................................... 13
   2.3 Principle of Mapreduce ......................................................................................... 14
   2.4 Chapter Conclusion ............................................................................................... 15

3 Scheme design .......................................................................................................... 17
   3.1 Proposal ................................................................................................................. 17
       3.1.1 Comparison of schemes ................................................................................. 18
       3.1.2 Reduce size of dataset .................................................................................... 19
       3.1.3 Comparison on dataset consistency .................................................................. 19

4 The Realization of Experimental Program .............................................................. 27
   4.1 Framework ............................................................................................................. 27
   4.2 The architecture design of Bloom Filter module ................................................... 28
       4.2.1 Basic framework ............................................................................................. 28
       4.2.2 Realize the program switch with Java reflection mechanism ......................... 28
       4.2.3 The realization of main functions ................................................................... 29
   4.3 MapReduce workflow ........................................................................................... 31
4.3.1 BasicJoin program ........................................................................................................ 32
4.3.2 BF program .................................................................................................................. 32
4.3.3 The BF program and IBF integrated program .............................................................. 36

5 Evaluation and Results........................................................................................................ 41
  5.1 Experiment setting ....................................................................................................... 41
    5.1.1 Experiment environment ...................................................................................... 41
    5.1.2 Generation of experiment data ............................................................................. 41
    5.1.3 Algorithm parameter table .................................................................................. 42
    5.1.4 Evaluating index and experiment execution ......................................................... 43
  5.2 Experiment result ........................................................................................................ 44
    5.2.1 Reduce size of dataset ......................................................................................... 45
    5.2.2 Accuracy ............................................................................................................... 45
    5.2.3 Relationship between number of data and runtime ............................................ 48
    5.2.4 Diagram of relationship between Accuracy and Runtime ................................... 51

6 Conclusion........................................................................................................................... 55

7 Bibliography........................................................................................................................ 57
Chapter 1 Introduction

1.1 Background

As a new concept proposed at the end of 2007, cloud computing was a great revolution in the IT field. It means that computing can be used as a commodity, like coal gas or electricity. The only difference between them is cloud computing is transported by the computer networks. Organizations may use Hadoop to run "big data" business intelligence-type operations. Practical applications for using Hadoop include supply-chain management, sales analytics, call-center record analysis, web event analysis and financial reporting. Up to now, Google, Microsoft, Alibaba, Amazon and some other famous cooperation have proposed their cloud computing applications and take cloud computing as one of the vital strategies of the future. Under this circumstance, data immigration and data reconciliation between database and distributed system are quite common. How to find a better algorithm which can help to speed up the reconciliation process and ensure the result is what enterprises concern about.

eBay has the largest e-commerce platform on earth, which accepts magnitude millions of trading activity orders each day. eBay has poured money and resources in data analysis platform and is committed to efficient data analysis in order to better grasp users' behavior. eBay not only applies traditional data warehouse (TeraData) but also estimates applying the latest Hadoop technology platform. They have conducted large amount of research and inputs on the efficiency of data transfer between the data warehouse and Hadoop platform as well as consistency maintenance.

Data in a data warehouse (TeraData, short as TD) need to be migrated to the Hadoop file system for OLAP (On-Line Analytical Processing) calculation and analysis, with time lapse, data in TD has changed while the replica in Hadoop does not synchronize updates in time, or replica is destroyed due to network transmission. Therefore, comparison needs to be conducted to identify differences between datasets in TD and datasets in the Hadoop file system.

Following issues are covered in this thesis: policy of data reconciliation, guarantee of data quality (accuracy), policy and performance of data transfer.
1.2 Study objective

The problem can be defined as Figure 1.1. First, in database, data is stored as records, and each record can be considered as a tuple of \(<\text{PK}, \text{Value}>\). Then, considering two data sets DSA (data in TD) and DSB (data in Hadoop), the problem can be defined as finding records in part A (the set of records in DSA but not in DSB) and part C (the set of records in DSB but not in DSA). Part D (inconsistency) means the set of records is in DSB and DSA but they are not the same. Data reconciliation in this thesis is to get the PK of each record in part A and part C.

Normally, DSA is considered as original data, while DSB is a duplicated of DSA. Inconsistent data can be caused by following operations:

1. Insert, One record is added to A while an insert operation is conducted in DSA;

2. Delete, One record is added to C while a delete operation is conducted in DSA;

3. Update, when an update operation is conducted, PK remains the same while value changes and one record is added in D.
Performance objective: The requirement is 10M records at maximum while the error rate is under 0.0001%. Runtime are tested based on schemes.

1.3 Content arrangement

This thesis primarily presents three typical algorithms: standard Bloom filter (BF), Counting Bloom filter (CBF) and Invertible Bloom filter (IBF). With the purpose of evaluating their performance, principally on runtime and accuracy rate, a series of experiments was designed and applied to both a small-scale and a large-scale distributed system. These algorithms are compared based on one traditional query method Basic Join (BJ). And the result shows: under the MapReduce computing framework, Basic Join, followed by BF closely, has the best performance; large-scale distributed system can evidently improve the performance on dealing with large-scale data.

The main study content is:

1) Study on Bloom filter algorithms, especially on efficiency.

2) Study latest BF algorithm and understand their principles, pros and cons, especially on standard BF and IBF. Their disadvantages can be analyzed, and an optimized plan can be proposed.

3) Understand the structure and the code of distributed system and transplant these BF algorithms to the distributed system. Adapt these algorithms to distributed system and complete experiments and analyze the result.

4) Compare the data reconciliation algorithm with transplanted BF algorithm and analyze the pros and cons.

According to content, the paper structure is as followed:

Chapter 1 is introduction. Difficulty of data reconciliation on the distributed system is introduced. A brief introduction of the selected distributed system - Hadoop is carried out. And the study objective of the paper forms the last part.

Chapter 2 describes the basic principles of Bloom filter algorithms and advantages, focusing on the detailed presentation of search methods and design ideas of several classic algorithms, analysis of the advantages and disadvantages about performance.

In Chapter 3, in order to verify the merits of Bloom filter algorithms on distributed platforms mentioned previously. Series of experiments are designed to prove improved algorithms.
The fourth chapter describes the architecture and implementation of the experimental system.

The fifth chapter analyses accuracy and efficiency of improved algorithms.

The final chapter is a summary and prospects for the future work.
Chapter 2 Related Technology about Bloom filter

Bloom filter query algorithm based on theoretical research in recent years has been concerned. Published at home and abroad from the current literature, there have been a number of standard bloom filters query algorithm algorithms, including: counting bloom filter query algorithm, compressed bloom filter query algorithm, Spectral bloom filters query algorithm, split bloom filter query algorithm, dynamic bloom filters query algorithms, scalable bloom filter query algorithm, multidimensional filter query algorithm, basket Bloom filter query algorithms etc. The Bloom filter expansion algorithms have made considerable contribution to bloom filter query algorithm research and development, but owing to lack of early concern to the field of algorithms, the algorithm is still in rising phase while there are a lot of interesting theoretical issues yet to be resolved.

2.1 Principle of Bloom filter algorithm

A Bloom filter, conceived by Burton Howard Bloom in 1970[1], is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. Bloom filter is a space-saving, efficient data representation and the structure of query. This algorithm can simply use bit arrays to represent a set, and to determine whether an element is in the set with a high probability. Therefore, this data structure is suitable for application in low-error-rates-tolerance occasions.
It is known from Figure 2.1, the nature of Bloom Filter structure is to map elements in one set to bit string vector via $k$ hash function, and for each element it just needs to save several bits. Bloom filter, as a query-supported data structure, is efficient, and simple in representing set. It may exist false-positive misjudgment (possibilities of an element does not belong to one data set and is refers to be judged to belong to the data set), and does not exist false-negative misjudgment (possibilities of an element does belong to one data set and is refers to be judged not belong to the data set).

Due to the false-positive misjudgement feature of Bloom Filter, in practice, the misjudgment rate must be evaluated so as to design appropriate Bloom filters and decrease the impact on applications.

2.2 Introduction on extensions of Bloom Filter

2.2.1 Counting Bloom Filter
Deleting elements from a Bloom filter cannot simply be done by changing ones back to zero, since a single bit may correspond to multiple elements. A counting Bloom filter uses an array of n counters instead of bits to support for deletion while there is a dynamic change in a set, the counters track the number of elements currently hashed to that location. It uses a counter instead of v vectors for each bit, and completes the insertion and deletion of elements by operating addition, subtraction on k Counter corresponds to elements. When using counting Bloom filter, the insert and delete operations cause corresponding k Counter in bloom filters to plus 1 or minus 1. And when determining whether an element is in the collection, it just needs to check all k Counter counters are greater than 1.

2.2.2 Scalable Bloom Filter

Because with a certain misjudgment f0, the standard Bloom filters can only accommodate a certain number of data n0, scalable Bloom filter is a better choice when Bloom filter needs to hold more data.

The initial conditions is a standard Bloom filter, when the number of data records n inserted in V0 (length is m0) is more than upper level of n0, another V1 is assigned with length m1=2m0. With a fixed f0, number of data V1 can accommodate is n1=2n0, and so on, with i times extensions, data number accommodated by scalable Bloom filter is (2I-1) n0, the length of vector Vi mi=2i x m0, Figure 2.2 shows this situation:

![Figure 2.2 Scalable Bloom Filter](image)

2.2.3 Invertible Bloom Filter

Invertible Bloom filter extends the Bloom filter data structure to key-value pairs and the ability to list out the pairs it contains with high probability, as long the
number of key-value pairs is below a designed threshold. Instead of bits in standard Bloom filter, IBF uses Cells, structured as Figure 2.3.

<table>
<thead>
<tr>
<th>counter</th>
<th>KeySum</th>
<th>hashKeySum</th>
<th>valSum</th>
<th>hashValSum</th>
</tr>
</thead>
</table>

Figure 2.3 Cell structure

Among them,
(1) counter stores records stored in the unit <key, value>;
(2) keySum stores sum of all key in the unit of <key, value>;
(3) hashKeySum records the unit of all <key, value> key
(4) valSum records the unit of all <key, value> value
(5) hashValSum records the unit of all <key, value> value

IBF in addition to supporting operations such as querying, inserting, deleting, and also support the listing element set operations, IBF can be an existing, <key, value> all, has it that the IBF reduced the entire data set into several cells. Compared with the process of BF, IBF just need to generate one filter and only do data reconciliation one time, due to IBF’s characteristic.

Though those algorithms perform very well in terms of communication cost and computation complexity, however, little research is done on large-scale distributed system, where distributed computation, the bandwidth cost and other factors may influence the performance.

2.3 Principle of Mapreduce

MapReduce can take advantage of locality of data, processing data on or near the storage assets to decrease transmission of data. It is a framework for processing parallelizable problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogenous hardware). Computational processing can occur on data stored either in a filesystem (unstructured) or in a database (structured).

Mapreduce divides the whole data computation process into two parts: “Map” and “Reduce” [5]. Map-Reduce[6][7] framework is based on <key,value> which
means data input is a number of <key,value> and the result is also a number of <key,value>, but sometimes their types are different.

"Map" step: The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node.

"Reduce" step: The master node then collects the answers to all the sub-problems and combines them in some way to form the output – the answer to the problem it was originally trying to solve.

Work flow was shown in the following Figure 2.4:

![Figure 2.4 Principle of Mapreduce](image)

2.4 Chapter Conclusion

The chapter is like its name suggested, several common BF algorithms and how Mapreduce work is introduced here.
Chapter 3 Scheme design

Detailed solutions to requirements and technologies mentioned above were carried out in this chapter. Before this chapter will be based on demand and technology described before specific solutions were made. Framework, solutions (including Basic Join, BF and IBF) were described in details.

In the selection of Hash function, according to the compression experiment (An experiment to test on compressing 100 billion record finished by our lab), LCH and LDH aka. Bernesten and BKDRHash were selected. Experiment data was illustrated in table 3.1.

<table>
<thead>
<tr>
<th>Hash function name</th>
<th>Number of conflict</th>
<th>Time spent on compressing 100 billion records</th>
</tr>
</thead>
<tbody>
<tr>
<td>APHash</td>
<td>134</td>
<td>296ms</td>
</tr>
<tr>
<td>Bernstein(LCH)</td>
<td>223</td>
<td>186ms</td>
</tr>
<tr>
<td>BKDRHash(LDH)</td>
<td>224</td>
<td>181ms</td>
</tr>
<tr>
<td>DJBHash</td>
<td>228</td>
<td>179ms</td>
</tr>
<tr>
<td>RSHash</td>
<td>231</td>
<td>186ms</td>
</tr>
<tr>
<td>DEKHash</td>
<td>236</td>
<td>189ms</td>
</tr>
<tr>
<td>rotatingHash</td>
<td>244</td>
<td>200ms</td>
</tr>
<tr>
<td>FNVHash</td>
<td>245</td>
<td>188ms</td>
</tr>
<tr>
<td>SDBMHash</td>
<td>250</td>
<td>235ms</td>
</tr>
</tbody>
</table>

Table 3.1 Comparison on Hash function performance

3.1 Proposal

Before raising the schemes, issues need to be solved must be clarified: The first issue is to reduce the number of data. Considering the size of data in Teradata and Hadoop is really huge, which may cause the transfer, access difficult and inefficient. Data is compressed to 128bits length by MD5 algorithms so as to reduce the size of data.
And the second issue is data reconciliation. Using the specific of BF algorithms, the schemes are designed to realize the compare of reduced dataset. Get a higher efficiency with some loss of accuracy.

There are various BF algorithms candidates, therefore, a rational structure to arrange these data structure is important. How to design the appropriate data type and port is the key.

3.1.1 Comparison of schemes

Experiments were designed to compare the participle plan, analyze their pros and cons in the space cost, runtime, accuracy so as to select the best scheme. The stage contained design and realization of the whole system.

According to issues mentioned above, experimental design contains two parts. The first part aims at reducing the size of dataset while the second part aims at compare data. There are 3 routines of schemes and all of them is realized on Mapreduce platform. The whole picture was illustrated in figure 3.1:
3.1.2 Reduce size of dataset

MD5 is a hash encryption algorithm that generates a fixed-length (128bit) string of bytes, you can use the MD5 algorithm to encrypt the original data, get 128bit byte strings can relieve the pressure on networks, is more conducive to subsequent data to work. MD5 collision rate is very low which makes it feasible for this project.

3.1.3 Comparison on dataset consistency

First, in database, data is stored as records, and each record can be considered as a tuple of <key, value>. The problem can be defined as, assume T={((key,value)|((key,value) in TD), and H={((key,value)|((key,value) in Hadoop). Data reconciliation[8] in this thesis is to each record in part A and part C and part D in figure1.1. (Since e-Bay does not care about correct records, part B is not taken into account.)

In the test, according to basic research done before, Basic Join(aka BJ), Bloom filter (aka BF) as well as Inversed Bloom filter (IBF) were chosen. They would be described in details below:

a) Basic Join (BJ)

The basic idea of this method was to view through records in one dataset one-by-one to check whether one record exists in another dataset and the area one record belongs can be found directly by comparing. Basic operation was as follows:

Take t ∈ T,
1. If there is no h ∈ H which enables t.key=h.key, then t ∈ A
2. If there is h ∈ H which enables t.key=h.key AND t.Val ≠ h.val, then t ∈ D
Take h ∈ h,
3. If there is no t ∈ T which enables t.key=h.key then h ∈ C

Analysis: the basic algorithm was to use a nested loop, traversal H for every record in T. The complexity of this algorithm is O (n^2). But since this scheme was implemented by Mapreduce on Hadoop, advantages of Mapreduce's grouping mechanisms can be taken to optimize this algorithm.

Advantage of Mapreduce's nature can be used completely to easily performs Intersection Operation (Join Operation) on two sets which was string comparison.
At the same time, sorting method of Mapreduce is close to TerraSort Benchmark standards which means the efficiency of this scheme is considerable and justified. Work flow was shown in the following figure:

![Figure 3.2: Basic Join Scheme](image)

b) Bloom Filter Scheme (BF)

The basic idea of this scheme was to build corresponding bloom filters for original data on both sides, and then record from one side was determined...
whether it exists in another dataset through query operation which avoided loop comparison of each piece of data. Query execution time of bloom filter depends on the \( k \) number of hash functions, and the time complexity of data \( N \) is constant. Algorithm design was as follows:

1. Generate BF1 in \( T \) (each record inserted in \( T \)), generates BF2 in \( H \) (each record inserted in \( H \))
2. take each record \( h \) in \( H \) to conduct query \( h \) BF1, if the query result is that \( h \) was not in BF1, then \( h \) was added to set \( nT \)
3. take each record \( t \) in \( T \) to conduct query \( h \) BF2, if the query result is that \( t \) was not in BF2, then \( h \) was added to set \( H \)
4. so far \( A, C, D \) were obtained by set operations like: \( D = nT \cap nH; A = nH - D; C = nT - D; \)

Analysis: bloom filter is a query-efficient algorithm, its time complexity is \( O(k) \), only depends on efficiency of hash function not on data \( n \) (Time complexity of building bloom filter is \( O(n) \)); since efficiency of query algorithms like traditional tree shaped structure algorithm was related to data quantity \( n \) directly, therefore efficiency of bloom filter has advantages dealing with large data quantity.

Scheme process is shown in the figure 3.3:
c) Inverted Bloom Filter Scheme (IBF)

The scheme used the feature of IBF listing all the elements it represents, using one IBF without having to establish two independent Bloom filter data structures compared to b). In this scenario, data on both sides was inserted into IBF in a reverse manner firstly, specifically insert operations was performed on one side, and delete operation was performed on the other side, after this action finished, same elements were offset on both sides and elements stored in IBF were the different records at both sides. Then all elements in the IBF were listed because the operation was reversed on two sides which generated two sets, one with the
data from insertion and another with data from deletion. Algorithm design was as follows:

1. Generate the initial IBF
2. IBF inserted each record of \( H \)
3. IBF deleted each record of \( T \)
4. List all the elements at this point into two sets \( nH \) and \( nT \)
5. The same with scheme b), find the set \( A,C,D \).

Analysis: the scheme was based on the Bloom filter algorithm, while IBF was simple in process since it need just one filter comparing to BF need two filter to build data structure, but since the algorithm need to store redundant information which made a larger space cost. Secondly, hash value need to compute repeatedly in the stage of listing data element, at this stage the data record stored in IBF happens to be different on both sides, the efficiency is \( O(d) \). On the other hand, advantage of IBF compared to BF is that IBF does not execute the query operation in the scheme, so there is no false positive, accuracy of algorithms is 100%.

Scheme process is shown in the figure 3.4:
d) Bloom filter parameters

- Determination of vector length

The length of bit vector (filters the required storage space) must be determined while establishing Bloom filter data structures in a program.

The length of bit vector of Standard Bloom filters is determined by the three input parameters: 1. $n_0$: the number of data can accommodate; 2. $f_0$: the maximum rate of misjudgment; 3. $k$: the number of hash functions. And length of Bloom filters bit vectors calculation formula is:

$$m = \frac{kn_0}{\ln \left(1 - (f_0)^{\frac{k}{2}}\right)}$$
\( f \) in fact is an indicator to show the precision of algorithms, and in this thesis, is treated as an input parameter. Space is allocated according to its value to ensure accuracy, which means the accuracy of the program is controllable, requirements about accuracy can be met when space is enough.

**Determination of the number of hash function**

Number of hash function impacts accuracy greatly, from the formula above, in order to ensure accurate rate remain unchanged, the vector length must be extended which would increase the space cost, therefore, take space cost and accuracy into consideration together, the number of hash function \( k \) is not the bigger the better. When \( k \) was between 2~4, accuracy rose greatly, and while \( k \) grew, rise of accuracy was no longer significant. On the other hand, time of Bloom filter query operations was also determined by \( k \) therefore, in order to ensure the efficiency of calculation, \( k \) should not be too large. Therefore, \( k \) was determined as 3 in this thesis.

**Choose of experimental parameters**

On choosing parameters of experiments, size of data and different error ratio was considered. While the minimum of data was 10 thousand and the maximum was 300 million. Error ratios were 2%, 10%, 20% and 50%. Detailed parameters would be showed in Chapter 5.
Chapter 4 The Realization of Experimental Program

Based on the realization task of comparison stage for experimental system dataset proposed in the program design of last chapter, this chapter firstly conducted brief introduction for the overall framework of the experimental system, and subsequently described the realization of different programs in the Mapreduce platform respectively, which included Basic Join program, BF program, and succession programs for BF and IBF.

4.1 Framework

The purpose of this program was to realize a set of experimental system, which included not only the implementation of the 3 programs, but also the comparison of the performance of various programs under the experimental system. The diagram of the overall framework was as follows:

Figure 4.1: The framework diagram of the experimental system

The part on the left of the dotted line was the program put forward by this thesis. After Mapreduce processing (Experiment Solution), TD Data and Hadoop
data would generate Reconciliation Result. Bloom Filter or Basic Join scheme was used in this process; meanwhile, the Performance Statistics Gathering module was applied to conduct statistics for the running time of the program. The part on the right of the dotted line was the control group. The same data was used for the standard program to gain the right result, and then the Accuracy Statistics Gathering module was used to compare the two sets of results, the accurate date of the experimental results, and Statistics could finally be obtained.

The following sections of this chapter would explain the main components of the above mentioned modules: the 4.2 section described the realization of Bloom filter algorithm module; the 4.3 section described the structure and realization of the MapReduce processing. Since the part on the right side of the dotted line and the statistics module were used as references and comparison rather than the actual needs of the program, it would not be further elaborated in this thesis.

4.2 The architecture design of Bloom Filter module

4.2.1 Basic framework

Various Bloom filters applied in this thesis were improved on the basis of standard Bloom filter, but their basic principles were the same: map the data through hash algorithm to the bit vector. Therefore, the various algorithms were in the same logic base class. First, established BloomFilterBase. Such interface for hash algorithm was used by sub-bases, and then IHashes interface needed to be established. As \( k \) choice of hash function was a parameter of the following experiments, different programs of hash combination needed to be implemented. When the BloomFilterHashBase achieved the IHashes interface, different combination programs of hash functions would inherit this base, and different Bloom filter would inherit BloomFilterBase so as to call a unified method for operation.

4.2.2 Realize the program switch with Java reflection mechanism
Both BloomFilterBase and BloomFilterHashBase mentioned in 4.2.1 contained a static method getInstance of Java reflection mechanism, which could specify the instantiated base name (complete path name) by inputting parameters through the program, establish the appropriate bases by the Class.forName method, and then generate specified type of instances by calling the newInstance method of Class instance. In this way, as long as the program parameter with specified sub-base of the Bloom filter was input, the corresponding instance of reduce phase could be obtained to implement the corresponding programs.

The base diagram was shown as below:

Figure 4.2 Class diagram of Bloom filter

4.2.3 The realization of main functions

1. BloomFilterBase
   getMBits: the method calculated the bit vector length by inputting parameters
getInstance: utilized JAVA reflection mechanism, input Bloom filter class, called the static method of Class.forName, obtained a Class object, and then called the newInstance method of the Class object to return an instance of a BloomFilterBase Class.

2. Hash combination
   BloomFilterHashBase also realized the getInstance method and achieved the flexible switch of subclasses. Moreover, it realized the combination of two sets of
hash functions. Selected three kinds from 9 common hash functions for the combination, and it was found that LeastCollisionHash was the combination with the lowest conflict rate, and LeastDurationHash was the combination with the shortest calculation time.

3. Counting BloomFilter
insert: calculated the hash value according to the data records and added 1 at the counter value of the hash value points.
delete: calculated the hash value according to the data records and reduced 1 at counter value of the hash value points.
exists: calculated the hash value according to the data records and read the counter value the hash value points; if all were greater than 0, it would return true, otherwise, it would return false.

4. ScalableBloomFilter
insert: first checked the number of records stored in the current filter: if it was greater than the threshold value $n_0$, it would extend (call extend) or find a latest extended bit vector, and then calculate the hash value according to the data records, and place the hash value at the location of 1.
extend: when at the $i$th expansion, the allocation length of $2^{n_0}$ was the new bit vector and was initialized into 0.
exists: calculated the hash value according to the data records, and read the counter value of that hash value points in the first bit vector: if all were greater than 0, it would return true, otherwise, it would be checked again in the next bit vector. If all bit vectors were checked as false, it would return false.

5. InvertedBloomFilter
insert: calculated the hash value according to the data record <key, val> and conducted the following operations for the cell the hash value points:
The counter added 1,
keySum cumulatively added key,
hashKeySum cumulatively added $g(key)$ ($g(x)$ was an independent hash function),
valSum cumulatively added val,
hashValSum cumulatively added $g(val)$
delete: opposite to the operation of insert, replaced the addition with subtraction
recover: listed all data records in the set represented in the filter. The specific method was:
1. Find the cell with counter value of 1 or -1
2. As for the counter value of 1, <keySum, valSum> could represent a data record. If \( g(keySum) = hashKeySum \) and \( g(valSum) = hashValSum \), it meant this record was from the insertion end, and this record would be output into a queue.
3. Delete this record from the filter (call the delete method)
4. As for the counter value of -1, <-keySum-valSum> could represent a data record, if \( g(keySum) + hashKeySum = 0 \) and \( g(valSum) + hashValSum = 0 \), it indicated that this record was from delete end, this record would be output into another queue.
5. Insert this record to filter (call the insert method)
6. Repeat the above operation until the filter is returned to the initial state

4.3 MapReduce workflow

The system implemented design and development in the Hadoop platform. According to the MapReduce programming model, the input of Mapreduce was a 2-tuple of (Key, Value), the map phase would collate the data and generate the appropriate intermediate results based on the need. After the output of intermediate results, it entered reduce stage, which mainly performed computing processing on the data, and finally output the results. Each Mapreduce operation was divided into the following modules:

Mapper: first called setup method to obtain the program context and configuration files, and then called the map method to execute the operation at the map phase. In this way, users could realize the desired computing through rewriting the map method. For each 2-tuple of (key, value) executed the map method for one time until all the data was processed.

Partitioner: performed grouping operation, and grouped the output results of the map according to certain rules so as to make them with the same Key to inherit such custom actions.

Reducer: performed reduce operations and the result was output to the file system and hereditament such custom operations.

Driver: used for configuring the various parameters and submission of
Mapreduce operations.

4.3.1 BasicJoin program

This program included the four basic modules required in Mapreduce model. The map stage would obtain the raw data, and the intermediate results would be directly output by adding the flag bit. Data obtained in reduce stage was the sorted data and all data with the same key would get a corresponding set of values to determine the situation of these flag bits and whether the value was equal so as to get the final result.

HMapper: read data from Hadoop end, and added flag bit of ‘H’
TMapper: read data from TD end, and added flag bit of ‘T’
JoinReducer: compared if the value of the same key was equal, and finally wrote the obtained final results into the file.

The input and output data flow was as follows:

<table>
<thead>
<tr>
<th>Component</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMapper</td>
<td>Offset: long, row: Text</td>
<td>Key: long, Hashes: String</td>
</tr>
<tr>
<td>TMapper</td>
<td>Offset: long, row: Text</td>
<td>Key: long, Hashes: String</td>
</tr>
<tr>
<td>BFPartitioner</td>
<td>Key: long, Hashes: String</td>
<td>Partition index: int</td>
</tr>
<tr>
<td>BFReducer</td>
<td>Key: long, Hashes: String</td>
<td>Left(A), Middle(C), Right(D)</td>
</tr>
</tbody>
</table>

Table 4.1: data flow of Basic Join

4.3.2 BF program

The main process of BF program and IBF was divided into two stages: the first stage was to generate Bloom Filter, while the second stage was the Data Reconciliation.
Figure 4.3 The stage diagram of program design

◆ The following components need to be realized at the first stage:
The entry program of **BFGenDriver Mapreduce** was used for the preparation and submission of tasks

**BFGenMapper** generated map method of Bloom filter and MD5 set

The **BFGenPartitioner** Grouper: it needed to be noted that the grouping strategies of the two stages must be consistent, as only this strategy could guarantee the Bloom filter to execute query operation when the Bloom filter record was inserted

**BFGenReducer** generated reduce method of Bloom filter

**BFGenerator** generated interface for the generated Bloom filter method

The class diagram was as follows:
Figure 4.4 The stage class diagram of BFGeneration in program design

- The sequence diagram was as follows

Figure 4.5 The stage sequence diagram of BFGeneration in BF program
The output of this stage was the Bloom filter, which output it into a file; the file would be read when doing data reconciliation in the second stage. As here the data structure of Bloom filter needed to be output to a file, it was necessary to realize its serialization. The method was to use Bloom filter class to realize the WritableComparable interface packed by Hadoop. After realizing the read and write methods in this interface, it could support the read and write operations of this class to the disk.

The second stage also needed to realize several classes of Mapreduce basic framework (as it was similar to 2.2.4, hereby it would not be described again)

![Diagram of Reconciliation in BF](image)

Figure 4.6 The stage class diagram of Reconciliation in BF diagram

Here the IBF interface was left out, but because the number and structure of intermediate results (Bloom filter) generated in BF program and IBF program were different, the realization of two sets of programs was needed, for this designed program was not scientific enough.

- The sequence diagram is as follows
After the completion of the above two stages, the nT and nH needed in the
algorithm could be obtained. The third stage was the Join operation. As the output
data was also stored in the distributed file system, and its data scale could not be
estimated, it also adopted Mapreduce to execute Join operation. To complete this
set of program, a total of three Mapreduce tasks must be executed.

4.3.3 The BF program and IBF integrated program

As described in the above 4.3.2, after testing, it could be found that as this design
needed to execute three Mapreduce tasks, data I/O operation occupied a large
proportion of the total running time, while BasicJoin only executed one
Mapreduce task, and the I/O operation time of two sets of programs was not
equal; their operation time was not comparable, as a result, the experimental
results were convincing enough. Therefore, the improvement of the program and
the aim of the new program integrated the two stages mentioned in 2.2.4 into

Figure. 4.7 The stage sequence diagram of Reconciliation in BF diagram
Mapreduce operation.

The advantage of this program was that it solved the large cost of I/ O; on the one hand, it could improve the execution efficiency, on the other hand, it made the program comparable with BasicJoin. The reason for not applying this program before was that originally the intermediate results written in a file must be read and written on the memory, which has a greater requirement on memory space. However, in the actual eBay experimental environment, the memory space and storage space of the distributed system were very large; moreover, the demand also mentioned that eBay concerned more about saving time and improving efficiency rather than save space. Therefore, this strategy of “exchanging space for time” was feasible.

In addition to the four basic modules required in the Mapreduce model, the program also included Bloom filter algorithm interface. The two programs were unified encapsulated in an interface of IBFOperator in order to create an interface for the Bloom filter operation. This interface mainly provided the two methods of create and reconcile. Here the create was used to create the data structure of the Bloom filter; while reconcile was used to execute query operation, and obtain the nH and nT set in the algorithm process.

Interface Join offered the method for the last set operation, and found the final results of A, C, and D with nH and nT as input parameters. The class Operator realized the IBFOperator interface. For the two different programs, it respectively realized the two classes inherited from Operator: BFOpr and IBFOpr, which respectively realized BF program and IBF program. The reason for applying this kind of realization in the experiment was that this program better utilized the common characteristics of BF program and IBF program, simplified the procedures, saved the execution steps and embodied the polymorphism and encapsulation advantages of the object-oriented programming design.

As the program adopted Mapreduce program, the original data needed to be divided into several blocks. As for the Bloom filter generated in each block (the mapping rule was BFKey = recordKey / nRecords, that is, divided the record key by the maximum number the filter could hold). In fact, with the Mapreduce parallel computing features, it reduced the needed storage space, and better played the performance advantage.
The class diagram of this program:

---

![The class diagram of this program](image)

---

Figure. 4.8 The Mapreduce framework diagram of Bloom filter program

The realization of the main functions:

Operator and its sub-classes: the base of Operator utilized JAVA reflection mechanism to realize the getInstance method.

The realization of the two sub-classes of BFOpr and IBFOpr: the former was responsible for the operation of the BF program, while the latter was in charge of the operation of the IBF program. The main functions were described as follows:

**BFOpr:**
- initialize method - creating two initial Bloom filters;
- create method – inserting the data of both ends (determined by the flag bit added at the map stage) into a Bloom filter respectively;
- reconcile methods – implementing the query operation in the Bloom filter generated at another end of the data with the data at the both ends.
IBFOpr:

- initialize method - creating an initial reversible Bloom filter;
- create method - utilizing data at the TD end to execute the delete operation in the filter, and executing insert operation for the data at the Hadoop end;
- reconcile method - calling IBF.recover, listing all data records stored in the IBF and putting them into two output queues.

Mapper:

1. Added different flag bits on the input data of both ends so that they could be processed respectively at the reduce stage.
2. Called IHashes interface to calculate the hash value, and passed it to reduce in the form of character string. Calculating the hash value in the map stage was because the hash value would be used in both create and reconcile stages. Therefore, calculating only once could save computing time. In order not to concentrate the calculating pressure of the operation on the reduce stage, it was executed in the map stage.
3. According to data recorded key, the corresponding key in the Bloom filter (BFKey) was calculated. That is, each data record was mapped to the corresponding Bloom filters.
4. The BFKey was the output Key of the map, the key of flag bit and data record were connected with the hash value as the output Value of the map.
Reducer:

1. Allocated instances for IBFOperator interface in the setup.
2. In order to avoid the re-allocated memory space, the memory must be reused, and the IBFOperator.clear method must first be called before each execution of the data processing operation so as to empty the contents of the allocated Bloom filter data structure for application.
3. Called IBFOperator:create method to establish the required filter data structure
5. Called the calculation method provided by Join interface provides, found the final results and then wrote to the file.

<table>
<thead>
<tr>
<th>Component</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMapper</td>
<td>Offset : long, row : Text</td>
<td>BFKey : long, Hashes : String</td>
</tr>
<tr>
<td>TMapper</td>
<td>Offset : long, row : Text</td>
<td>BFKey : long, Hashes : String</td>
</tr>
<tr>
<td>BFPartitioner</td>
<td>BFKey : long, Hashes : String</td>
<td>Partition index String</td>
</tr>
<tr>
<td>BFReducer</td>
<td>BFKey, Hashes : String</td>
<td>Left (A), Middle (C), Right (D)</td>
</tr>
</tbody>
</table>

Table 4.2 Dataflow of integrated program
Chapter 5 Evaluation and Results

Experiment design and execute process, result and analysis are carried out in this chapter.

5.1 Experiment setting

5.1.1 Experiment environment

1. Small cluster
Small cluster experiment is carried out in a three-node Hadoop cluster, with the following configure:

CPU: Inter(R) Core™ 1.86GHZ
Memory: 8G
OS: Ubuntu 10.04
Hadoop version: Cloudera CDH3u2.
It has one Name node and three Data nodes and executed small-scale data from 10K rows to 10M rows.

2. Large-scale cluster
Large-scale cluster experiment is carried out in a 1000-node Hadoop cluster provided by eBay CDC. The input data size starts from 10M rows, with 10M incremental data rows for each time, till it reaches 300M data rows.

5.1.2 Generation of experimental data

Data stimulated both from TD and Hadoop and got prepared beforehand. All data has been put into binary set and stored in Hadoop.

The scale of output data was determined with requirement proposer and the input parameter was put in table 5.1.
<table>
<thead>
<tr>
<th>Data ID</th>
<th>MD5bits</th>
<th>Number of records</th>
<th>Error rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>G041</td>
<td>128bits</td>
<td>1M</td>
<td>2%</td>
</tr>
<tr>
<td>G051</td>
<td>128bits</td>
<td>5M</td>
<td>2%</td>
</tr>
<tr>
<td>G012</td>
<td>128bits</td>
<td>10K</td>
<td>10%</td>
</tr>
<tr>
<td>G022</td>
<td>128bits</td>
<td>100K</td>
<td>10%</td>
</tr>
<tr>
<td>G032</td>
<td>128bits</td>
<td>500K</td>
<td>10%</td>
</tr>
<tr>
<td>G042</td>
<td>128bits</td>
<td>1M</td>
<td>10%</td>
</tr>
<tr>
<td>G052</td>
<td>128bits</td>
<td>5M</td>
<td>10%</td>
</tr>
<tr>
<td>G062</td>
<td>128bits</td>
<td>10M</td>
<td>10%</td>
</tr>
<tr>
<td>G013</td>
<td>128bits</td>
<td>10K</td>
<td>20%</td>
</tr>
<tr>
<td>G023</td>
<td>128bits</td>
<td>100K</td>
<td>20%</td>
</tr>
<tr>
<td>G033</td>
<td>128bits</td>
<td>500K</td>
<td>20%</td>
</tr>
<tr>
<td>G043</td>
<td>128bits</td>
<td>1M</td>
<td>20%</td>
</tr>
<tr>
<td>G053</td>
<td>128bits</td>
<td>5M</td>
<td>20%</td>
</tr>
<tr>
<td>G063</td>
<td>128bits</td>
<td>10M</td>
<td>20%</td>
</tr>
<tr>
<td>G044</td>
<td>128bits</td>
<td>1M</td>
<td>50%</td>
</tr>
<tr>
<td>G054</td>
<td>128bits</td>
<td>5M</td>
<td>50%</td>
</tr>
</tbody>
</table>

Table 5.1 Input parameter of experiment data

5.1.3 Algorithm parameter table

Parameters blow are input parameters of experiment:

<table>
<thead>
<tr>
<th>Algorithm ID</th>
<th>Solutions</th>
<th>Hash combinations</th>
<th>Error rate allowed by data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BJ</td>
<td>BF</td>
<td>IBF</td>
</tr>
<tr>
<td>A10</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A21</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A31</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A41</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>A51</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>A61</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A71</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>A22</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A32</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>A42</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
5.1.4 Evaluating index and experiment execution

Accuracy evaluation was divided into two indicators, one is cover rate while another is accuracy.

A. Set A is defined as record with difference in the original dataset in Figure 1.1, Set B is record identified by the program (matched with Set A, records with difference), Set C is all of the record identified by the program.

Coverage refers to the ratio that record in Set B divided by record in Set C. In consistency comparative tests, there is likely to find some not correct result records, that is why this indicator need to be measured. The formula is

\[
\text{Cover rate} = \frac{\text{Number of records of all the data in Set B}}{\text{Number of records of all the data in Set C}}
\]

B. Accuracy refers to the ratio that records in Set B divided by record in Set A. In order to get the actual differences, an accurate result done by Basic Join for each dataset is made as a reference, so as to evaluate the accuracy of Bloom filters. The formula is:

\[
\text{Accuracy} = \frac{\text{Number of records of all the data in Set B}}{\text{Number of records of all the data in Set A}}
\]

C.
Memory space is the total memory space of all filters. Total memory space is measured as follows:

1. Establish a global accumulator - ExMem;
2. In the setup phase of Reducer, the program builds and when the Bloom filters are completed, space of the filters can be determined, MemUsage method is called to get the space of this filter;
3. This value is cumulative to the accumulator ExMem;
4. View the value of the accumulator ExMem when the program finished, write down the total memory consumption.

Note: due to Mapreduce parallel mechanisms, total memory space demand is not the maximum memory requirements at the same time, because this value does not have a typical representative meaning, it can only be used as a macro reference. So it is not concerned in this thesis.

What really useful is memory space for Bloom filters allocated by all the parallel Reducer at the same time, but since how many Reducer nodes are running parallelly at the same time cannot be measured, this value is hard to be calculated, so space overhead is not further discussed in this experiment.

D.

Running time is time of Mapreduce start running to end running and is measured as follows:

1. Write down the time before program starts- start;
2. Call the statement execute Mapreduce job-waitforcomplete ();
3. Write down the end time, find the time of executing the job (including I/O time in Mapreduce and computing time), Duration=end-start;
4. Write a statistics program. Use Java to generate a program to statistic results, including program running time, memory required and accuracy. The program used Hadoop HDFS API, to read and write file in Stream way.
5. Write experiment script using bash shell[15], subject was a loop, table 3 and table 4 were used as input parameters for experiment, and all the input parameters were the Cartesian product of two tables. The file also included the setting of input parameters.
5.2 Experiment result

5.2.1 Reduce size of dataset

It was mentioned in the experimental design part that MD5 is used to reduce the size of data in this scheme. MD5 method is to convert each record to a pair of (PK, MD5) key, theoretically, dataset size cannot be reduced using this method if the length of raw data is less than 128 bits. The effect is related to length of data, which means the bigger the raw data, the bigger the scale.

Data sample generated according to real need, the result of this stage is shown below:

<table>
<thead>
<tr>
<th>Number of data(row)</th>
<th>Size of raw data(KB)</th>
<th>Size of data after applying MD5(KB)</th>
<th>Reduction Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>100K</td>
<td>5632</td>
<td>3950</td>
<td>29.8%</td>
</tr>
<tr>
<td>500K</td>
<td>29460</td>
<td>19966</td>
<td>32.2%</td>
</tr>
<tr>
<td>1M</td>
<td>59245</td>
<td>40474</td>
<td>31.6%</td>
</tr>
<tr>
<td>5M</td>
<td>309245</td>
<td>204536</td>
<td>33.8%</td>
</tr>
<tr>
<td>10M</td>
<td>621745</td>
<td>414497</td>
<td>33.3%</td>
</tr>
</tbody>
</table>

Table 5.3 Experiment result of reducing dataset using MD5

It can be regarded that, dataset size is reduced to 70%. The advantage of this method is that all records are mapped into a fixed-size file. Data is standardized while it is reduced.

5.2.2 Accuracy

Part of the result was put below in table 5.4, number of data records was 1M and 5M while error rate was 20% and the accuracy was based on different input parameters.
<table>
<thead>
<tr>
<th>G043</th>
<th>BJ</th>
<th>1M</th>
<th>20%</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>G043</td>
<td>BF</td>
<td>A21</td>
<td>20%</td>
<td>2</td>
<td>1.0000%</td>
<td>1</td>
<td>0.99891</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A31</td>
<td>20%</td>
<td>3</td>
<td>1.0000%</td>
<td>1</td>
<td>0.998526 667</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A41</td>
<td>20%</td>
<td>4</td>
<td>1.0000%</td>
<td>1</td>
<td>0.99891</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A51</td>
<td>20%</td>
<td>5</td>
<td>1.0000%</td>
<td>1</td>
<td>0.998526 667</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A61</td>
<td>20%</td>
<td>6</td>
<td>1.0000%</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A71</td>
<td>20%</td>
<td>7</td>
<td>1.0000%</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A22</td>
<td>20%</td>
<td>2</td>
<td>0.0100%</td>
<td>1</td>
<td>0.999986 667</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A32</td>
<td>20%</td>
<td>3</td>
<td>0.0100%</td>
<td>1</td>
<td>0.99996</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A42</td>
<td>20%</td>
<td>4</td>
<td>0.0100%</td>
<td>1</td>
<td>0.999986 667</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A52</td>
<td>20%</td>
<td>5</td>
<td>0.0100%</td>
<td>1</td>
<td>0.99996</td>
</tr>
<tr>
<td>G043</td>
<td>IBF</td>
<td>A62</td>
<td>20%</td>
<td>6</td>
<td>0.0100%</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G043</td>
<td>IBF</td>
<td>A72</td>
<td>20%</td>
<td>7</td>
<td>0.0100%</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A23</td>
<td>20%</td>
<td>2</td>
<td>0.0001%</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A33</td>
<td>20%</td>
<td>3</td>
<td>0.0001%</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A43</td>
<td>20%</td>
<td>4</td>
<td>0.0001%</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G043</td>
<td>BF</td>
<td>A53</td>
<td>20%</td>
<td>5</td>
<td>0.0001%</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G043</td>
<td>IBF A63</td>
<td>1M</td>
<td>20 %</td>
<td>6</td>
<td>0.0001 %</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G043</td>
<td>IBF A73</td>
<td>1M</td>
<td>20 %</td>
<td>7</td>
<td>0.0001 %</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G053</td>
<td>BJ A10</td>
<td>5M</td>
<td>20 %</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G053</td>
<td>BF A21</td>
<td>5M</td>
<td>20 %</td>
<td>2</td>
<td>1.0000 %</td>
<td>1</td>
<td>0.99898 467</td>
</tr>
<tr>
<td>G053</td>
<td>BF A31</td>
<td>5M</td>
<td>20 %</td>
<td>3</td>
<td>1.0000 %</td>
<td>1</td>
<td>0.99855 533</td>
</tr>
<tr>
<td>G053</td>
<td>BF A41</td>
<td>5M</td>
<td>20 %</td>
<td>4</td>
<td>1.0000 %</td>
<td>1</td>
<td>0.99898 467</td>
</tr>
<tr>
<td>G053</td>
<td>BF A51</td>
<td>5M</td>
<td>20 %</td>
<td>5</td>
<td>1.0000 %</td>
<td>1</td>
<td>0.99855 533</td>
</tr>
<tr>
<td>G053</td>
<td>IBF A61</td>
<td>5M</td>
<td>20 %</td>
<td>6</td>
<td>1.0000 %</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G053</td>
<td>IBF A71</td>
<td>5M</td>
<td>20 %</td>
<td>7</td>
<td>1.0000 %</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G053</td>
<td>BF A22</td>
<td>5M</td>
<td>20 %</td>
<td>2</td>
<td>0.0100 %</td>
<td>1</td>
<td>0.99998 867</td>
</tr>
<tr>
<td>G053</td>
<td>BF A32</td>
<td>5M</td>
<td>20 %</td>
<td>3</td>
<td>0.0100 %</td>
<td>1</td>
<td>0.99996 933</td>
</tr>
<tr>
<td>G053</td>
<td>BF A42</td>
<td>5M</td>
<td>20 %</td>
<td>4</td>
<td>0.0100 %</td>
<td>1</td>
<td>0.99998 867</td>
</tr>
<tr>
<td>G053</td>
<td>BF A52</td>
<td>5M</td>
<td>20 %</td>
<td>5</td>
<td>0.0100 %</td>
<td>1</td>
<td>0.99996 933</td>
</tr>
<tr>
<td>G053</td>
<td>IBF A62</td>
<td>5M</td>
<td>20 %</td>
<td>6</td>
<td>0.0100 %</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G053</td>
<td>IBF A72</td>
<td>5M</td>
<td>20 %</td>
<td>7</td>
<td>0.0100 %</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G053</td>
<td>BF A23</td>
<td>5M</td>
<td>20 %</td>
<td>2</td>
<td>0.0001 %</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G053</td>
<td>BF</td>
<td>5M</td>
<td>20</td>
<td>3</td>
<td>0.0001</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
If Bloom filter predicates a set does not contain an element it cannot make mistake, therefore in the program made in this thesis, eventually results in the does not contains those does not in a, and c, and d in the of records; but due to bloom filter exists positive miscalculation in query, so it may predicate some elements which does not exist in one set that in fact they are in by mistake which caused loss of accuracy.

It is known from table 5.4 above:

The accuracy of BJ is 100%, the same as the scheme guesses.

Considering about BF algorithms (shown in light grey part) when the false rate of one algorithm is high, the accuracy of result is low and they maintains at the same accuracy. (For example, when false rate is 1%, accuracy is higher than 99% which means the misjudgment in result is lower than 1%).) Dark grey in table means, when false rate of one algorithm is lower, accuracy gets higher, nearly 100%.

Accuracy of IBF is 100% based on the specification of the algorithm, the same as scheme assumes.

Similarly the same law appears when the number of data is 5M. In the regard, considering about accuracy, Basic Join and IBF scheme can guarantee a 100% accuracy based on algorithm features. Accuracy in BF schemes is adjustable based on input parameters, specifically is to set the false rate in input parameters based on requirement (for example, when the false rate is below 0.0001%, then the parameter need to be set 0.0001%). Based on the result above, accuracy of all the schemes can meet the requirement of the project.

<table>
<thead>
<tr>
<th></th>
<th>A33</th>
<th>%</th>
<th>%</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>G053</td>
<td>BF A43</td>
<td>5M</td>
<td>20</td>
<td>4</td>
<td>0.0001 %</td>
</tr>
<tr>
<td>G053</td>
<td>BF A53</td>
<td>5M</td>
<td>20</td>
<td>5</td>
<td>0.0001 %</td>
</tr>
<tr>
<td>G053</td>
<td>IBF A63</td>
<td>5M</td>
<td>20</td>
<td>6</td>
<td>0.0001 %</td>
</tr>
<tr>
<td>G053</td>
<td>IBF A73</td>
<td>5M</td>
<td>20</td>
<td>7</td>
<td>0.0001 %</td>
</tr>
</tbody>
</table>

Table 5.4 Experiment result of accuracy
5.2.3 Relationship between number of data and runtime

Six groups of different number of data were tested in this experiment, 10 thousand; 100 thousand; 500 thousand; 1 million; 5 million and 10 million. And related parameters were: error rate: 20%, algorithm false rate: 0.0001. Experiments were conducted with these parameters. Results were shown in the table below, 1 was BJ, 2 was CBF, 4 was SBF and 6 was IBF. Each experiment was conducted twice at least to minimize the accidental error and the time is the average value of all the runtime. This table shows the runtime trend with a growing number of data, x-axis is number of data (row) while y-axis is runtime (second).

<table>
<thead>
<tr>
<th>Data (row)</th>
<th>BF</th>
<th>CBF</th>
<th>SBF</th>
<th>IBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>41.84</td>
<td>41.52</td>
<td>51.35</td>
<td>62.38</td>
</tr>
<tr>
<td>100000</td>
<td>41.69</td>
<td>44.97</td>
<td>54.13</td>
<td>66.23</td>
</tr>
<tr>
<td>500000</td>
<td>39.43</td>
<td>44.78</td>
<td>55.03</td>
<td>64.18</td>
</tr>
<tr>
<td>1000000</td>
<td>55.83</td>
<td>49.62</td>
<td>53.96</td>
<td>71.91</td>
</tr>
<tr>
<td>5000000</td>
<td>181.25</td>
<td>141.67</td>
<td>161.35</td>
<td>161.35</td>
</tr>
<tr>
<td>10000000</td>
<td>352.30</td>
<td>268.81</td>
<td>268.81</td>
<td>268.81</td>
</tr>
</tbody>
</table>

Figure5.1 Diagram of the number of data and runtime

It can be told from the diagram that with a small number of data (less than 1M), the differences are very little. However, with the growing of data, Basic Join showed its superiority. It can be told from the diagram that Basic Join cost more time than others only when number of data was 10 thousand while it cost the least during all other cases. Therefore, BJ would be regarded as reference group in analysis below.

Time cost of BJ minus time cost of other algorithms divided by time cost of BJ will be used to compare differences of other three algorithms with BJ.
<table>
<thead>
<tr>
<th>Number of data</th>
<th>BJ compared to other algorithms</th>
<th>Speed Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CBF</td>
<td>SBF</td>
</tr>
<tr>
<td>10K</td>
<td>-0.15</td>
<td>-2.40</td>
</tr>
<tr>
<td>100K</td>
<td>0.83</td>
<td>0.63</td>
</tr>
<tr>
<td>500K</td>
<td>2.78</td>
<td>3.68</td>
</tr>
<tr>
<td>1M</td>
<td>3.86</td>
<td>1.80</td>
</tr>
<tr>
<td>5M</td>
<td>27.95</td>
<td>7.12</td>
</tr>
<tr>
<td>10M</td>
<td>97.05</td>
<td>10.83</td>
</tr>
</tbody>
</table>

Table 5.5 Runtime comparison between Basic Join and other algorithms

It can be told from the table, when number of data was 10 thousand, BJ was the slowest of them all. 5.7% (2.4sec) slower than SBF and 0.3%(0.15sec) than CBF.

When number of data was 100 thousand, BJ was the fastest among all algorithms, 0.7%(0.32 sec) than IBF, 1.9%(0.83 sec) faster than CBF and 1.4%(0.63 sec) faster than SBF.

When number of data was 500 thousand, BJ was the fastest among all algorithms, 5.1%(2.61 sec) than IBF, 5.4%(2.78 sec) faster than CBF and 7.2%(3.68 sec) faster than SBF.

When the number of data was 100 thousand, BJ was the fastest among all algorithms, 0.7%(0.32 sec) than IBF, 1.9%(0.83 sec) faster than CBF and 1.4%(0.63 sec) faster than SBF.

When the number of data was 5 million, BJ was the fastest among all algorithms, 5.5%(7.12 sec) than IBF, 21.6%(27.95 sec) faster than CBF and 24.9%(32.15 sec) faster than SBF.

When the number of data was 10 million, BJ was the fastest among all algorithms, 4.1% (10.83 sec) than IBF, 36.4%(97.05 sec) faster than CBF and 32.1%(85.7 sec) faster than SBF.

It can be told from the diagram that with a small number of data (10 thousand), the differences was very little. However, with the growing of data, Basic Join showed its superiority. When the number of data was 100 thousand, 500 thousand and 1 million, BJ was faster than other three schemes but the gap was only 7%, not that wide. With a greater number of data like 5 million and 10
million, BJ was about 30% faster than CBF and IBF while the speed of SBF was 5.5% and 4.1%, still quite close. Overall, the greater the number of data was the more obvious the advantage of BJ was. SBF can keep a good performance (about 5% slower than BJ) no matter number of data was big or small, which means change of the number of data did not affect SBF’s performance. CBF and IBF had a bad performance when the number of data was big.

It can be concluded from the experiment that BJ is the most efficient regardless of the size of data which is consistent with the description of paper (in Chapter 3). Analysis of other three algorithms was as follows:

Theoretically, time cost in query phase of CBF and SBF only associated with the execution time of a hash function, not affected by the amount of data, thus the reason of time-consuming increase was inferred due to building Bloom filters in data insert phases since its time complexity was O(n).

On the other hand, with the increases of input data, the memory space assigned to the Bloom filter data structures by program also increased, thus resulted in increased cost of visiting memory; this was also the reason why CBF’s efficiency was not as good as SBF’s since SBF required less memory space than CBF from the analysis of structure data, so SBF’s runtime was the closest to Basic Join. Impact of memory cost on runtime can be further optimized in the follow-up job.

On considering No.6 algorithm-IBF scheme, time to generate the data structure would be affected by the number of data, the runtime to recover was related to number of inconsistent data. The more inconsistent data there was, the more time “recover” need. In the experiment, data error rate stand the same, therefore, with the increasing of data, the number of inconsistent was going to increase which increased the time cost of IBF scheme.

The difference between IBF and BF was that in data structure generating phrase, the former performs on only one traverse while BF have to perform traverses for the source data on both sides in order to generate their own data structures. Due to different data structure, IBF consumes more time on operating metadata, so total time was different based on different input data. And statistical data can be concluded from the first column of the chart (10K) that IBF took the minimum runtime due to rare inconsistent data and in the recovery phase also. And IBF defeated BF in building structure stage as well.
5.2.4 Diagram of the relationship between Accuracy and Runtime

G052 (5 million) was selected as experimental input data, runtime of different algorithm with different accuracy is illustrated below (false rate are 0.01; 0.0001; 0.000001 while Basic Join is set as 0). A10 was Basic Join, A2* and A3* were CBF; A4* and A5* were SBF while A6* and A7* were IBF.

It can be concluded from this diagram that different accuracy for different algorithm leads to different run time. In all the schemes, BJ has the shortest run time meanwhile it is not affected by parameter. A table can be concluded using the same way as table 5.5:

<table>
<thead>
<tr>
<th>No.</th>
<th>Algorithm</th>
<th>Time cost of algorithm</th>
<th>Speed Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>–time cost of BJ(s)</td>
<td></td>
</tr>
<tr>
<td>0.000001</td>
<td>0.0001</td>
<td>0.01</td>
<td>0.000001</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>397.02</td>
<td>35.50</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>409.11</td>
<td>31.65</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>21.10</td>
<td>25.24</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>25.82</td>
<td>17.60</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>187.39</td>
<td>32.10</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>255.16</td>
<td>38.43</td>
</tr>
</tbody>
</table>

Table 5.6 Runtime comparison of algorithms on different accuracy

It can be seen from 1st and 2nd lines that CBF required a very high accuracy (error less than 0.000001) and the difference was very obvious. Its runtime was
approximately 3 times more than Basic Join’s (332.9% and 332.7%), When accuracy was 0.0001, its runtime was a few percentage points more compared to Basic Join. But when accuracy requirement was low (0.01), its runtime was a dozen percentage points more than Basic Join’s. And it can be concluded that when accuracy is 0.0001 CBF has a better performance.

It can be seen from the 4th and 5th line when accuracy changes, change of SBF’s runtime was small compared to Basic Join, both showed a dozen percentage points which means its performance is not affected by accuracy greatly.

It can be seen from the 5th and 6th lines that IBF required an extremely high accuracy. When error was less than 0.000001, its runtime was approximately 1.5~2 times more than Basic Join, the differences were very obvious. And when accuracy was 0.0001 and 0.01, runtime showed little difference, 10% more than Basic Join. With validation in the previous chapter, IBF can guarantee a completely accurate result, so here accuracy was just one incoming parameters which affects memory space allocated to IBF data structures by system, thus precision parameters should not be configured too high but fits the actual situation to reduce the memory space allocated.

When accuracy rate was low (<0.000001), time-consuming of different scenarios was not notable, but when accuracy rate was high, time-consuming of CBF and IBF increased bursting. The reason can be explained as, with the increase of accuracy rate, in order to guarantee accuracy more memory space must be distributed so as to guarantee the low conflict rate of hash algorithm. Therefore it was inferred that this time-consuming increase situation was due to increase of memory need. And while this algorithm was short of memory space, it caused a large amount of page-changing operation which brought interference to experimental results. And since SBF need smaller memory space comparing to CBF and IBF, it was not affected significantly.
Chapter 6 Conclusion

With the advantage of Mapreduce’s parallel computing capabilities, three scenarios were generated using the Mapreduce program model on Hadoop platform. And according to customer needs, this thesis generated mass of data (maximum 10M records) for experiment and validated the feasibility, conducted experiments on the improved scenario. In order to meet the need of the experimental process, the whole experimental system was designed using the object-oriented method; and on program structure, JAVA reflection mechanism was used to achieve a flexible class-switching code which enabled the flexibility so as to experiment with mass data by adjusting parameters.

Last, analysis was made through current experimental results, with accuracy, and runtime. In accuracy, the conclusion was that all three scenarios can provide enough guarantee and can meet customer of needs (errors in 0.0001% within). Basic Join and IBF can guarantee a 100% accurate rate while BF can reach an acceptable accurate rate by adjusting parameters like gaining more space.

In efficiency, when the amount of data was small (10,000), BF was faster than Basic Join and when the amount of data was greater than 10,000, efficiency of Basic Join was the best in three scenarios. On the premise of ensuring accuracy, SBF was close to Basic Join, while CBF and IBF were about 2~3 times slower.

When the need for accuracy was low, CBF, SBF and IBF were almost the same in runtime while Basic Join was slightly better. And when the need for accuracy was high, due to the sharp increase in memory space, which caused a spurt in the runtime of CBF and IBF, and SBF was not sensitive to accuracy adjustment.

According to laboratory test results, on one hand, increase of runtime was found in memory operations during experiments with a higher accuracy which caused a certain amount of interferences on results. On the other hand, if experiments were conducted in an environment with better hardware and bigger memory space, the results may also change. In short, memory operations can be optimized, such as how to reduce the space requirements of CBF. It can be said that the win of Basic Join further validates the efficiency of Mapreduce.
Chapter 7 Bibliography