

University of Windsor

Scholarship at UWindor

Electronic Theses and Dissertations

Theses, Dissertations, and Major Papers

1999

Reduction of collisions in Bloom filters during distributed query optimization.

Yan Liang
University of Windsor

Follow this and additional works at: <https://scholar.uwindsor.ca/etd>

Recommended Citation

Liang, Yan, "Reduction of collisions in Bloom filters during distributed query optimization." (1999).
Electronic Theses and Dissertations. 3291.
<https://scholar.uwindsor.ca/etd/3291>

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Reduction of Collisions in Bloom Filters during Distributed Query Optimization

by
Yan Liang

A Thesis
Submitted to the College of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the
Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

1999



National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

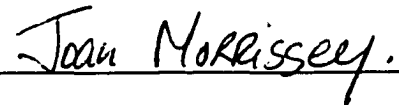
0-612-52744-1

Canada

903695


Yan Liang 1999
© All Rights Reserved

APPROVED BY:

_____

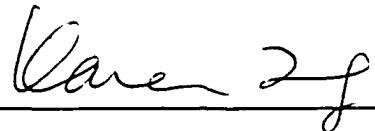
Dr. J. Morrissey, School of Computer Science

Principal Advisor

_____

Dr. C. Ezeife, School of Computer Science

Department Reader

_____

Dr. K. Fung, Department of Mathematics and Statistics

Outside Department Reader

Abstract

The goal of distributed query optimization is to find the optimal strategy for the execution of a given query. The approaches in distributed query processing have mainly focused on the use of joins, semijoins, and filters. Semijoins have the advantage over joins in that there are no increases in data sizes. However, a semijoin needs more local processing such as projection and higher data transmission. To improve the distributed query processing, the filter-based approach is utilized. One of the limitations of this approach is collisions.

We investigate how collisions affect the performance of the algorithm and how performance can be improved given those collisions. Our proposed algorithm utilizes two sets of filters to reduce the collisions, so the performance has been improved when collisions exist.

Our proposed algorithm is evaluated objectively by comparison to a full reducer which is the algorithm that fully reduces all relations involved in a query by eliminating all non-participating tuples from the relations.

The results of the evaluation show that:

1. With a perfect hash function, on average, our algorithm eliminates 97.41% of the unneeded data and fully reduces the relations of over 70% of the queries.
2. Using a single set of filters with specific percentages of collisions, on average, less than half of all queries are fully reduced by the algorithm. Therefore, the collisions substantially affects the performance..
3. Using two sets of filters, On average, our algorithm eliminates 95% of non-contributive tuples and achieves over 60% full reduction.

In conclusion, our improved algorithm utilizes the two sets of filters to reduce the effects of collisions substantially. Therefore, we improve the performance of our algorithm under the assumption of collisions which is the major problem in using Bloom filters during distributed query optimization.

to my mother, my son and my husband...

Acknowledgments

This work could not have been accomplished without the help of so many people.

First, I would like to thank Dr. Morrissey for the ideas of the algorithm, the using of two sets of filters, and a lots of her precious time for the consultation on this work.

Thanks Dr. Ezeife and Dr. Fung for their comments on my thesis.

Secondly, I would like to thank my family for their support. My son, Cheng Zhou, brought me the joy and took good care of himself, so I could work very hard on this thesis. My husband, Lenong Zhou, drove far from home for his work in order to let me stay near the university, so I could easily discuss the problems with my supervisor and do the survey.

Contents

Abstract	iv
Acknowledgments	vi
List of Figure	ix
 Chapter 1 Introduction.....	 1
Chapter 2 Background	3
2.1 Definitions	3
2.2 Approach I: Joins.....	4
2.3 Approach II: Semijoins	9
2.4 Approach III: Filters.....	14
2.5 Conclusions	18
 Chapter 3 The Algorithm	 19
3.1 Details of the Algorithm.....	19
3.2 An Example of Using the Algorithm	21
 Chapter 4 Experiments.....	 28
4.1 Experimental System.....	28
4.1.1 Rationale of the Experiments.....	29
4.1.2 Queries and Relations	29
4.1.3 Full Reducer.....	31
4.1.4 Analysis Programs	33

4.2 Experiments without Collisions-----	34
4.3 Experiments with Collisions Using a Single Set of Filters -----	34
4.3.1 Simulating Collisions-----	35
4.3.2 Problem Caused by Collisions-----	36
4.4 Experiments with Collisions Using Two Sets of Filters-----	39
4.4.1 Two Sets of Filters -----	41
Chapter 5 Results-----	45
5.1 Results without Collisions-----	45
5.2 Results with Collisions Using a Single Set of Filters -----	46
5.3 Results with Collisions Using Two Sets of Filters-----	48
Chapter 6 Conclusions -----	51
Bibliography-----	53
Vita Auctoris -----	58

List of Figures

Figure 2-1. The join of R1 and R2 over attribute B	5
Figure 2-2. The Semijoin of R2 by R1 over attribute B.....	9
Figure 2-3. The reduction of R2 by the reduction filter for attribute b.....	17
Figure 3-1. The five relations of the example.....	21
Figure 3-2. The adjacency matrix	22
Figure 3-3. The adjacency lists	22
Figure 3-4. The inverted lists	23
Figure 3-5. Step 1 of the example	24
Figure 4-1. 'Rel' file for a relation	30
Figure 4-2. Result of joining of five relations.....	32
Figure 4-3. Results of fully reduced relations.....	33
Figure 4-4. Simulating collisions	36
Figure 4-5. Relations for illustration the effect of collisions	37
Figure 4-6. Collisions in Bloom filter B	37
Figure 4-7. Reduced R2 and updated filter B	38
Figure 4-8. Reduced R3 and updated filter B	39
Figure 4-9. Example relations	40
Figure 4-10. Fully reduced relations.....	40
Figure 4-11. Non reduction of R2	41
Figure 4-12 Two sets of filters	43
Figure 4-13 Reduced R2 and updated filters.....	44
Figure 5-1. Results without collisions.....	45
Figure 5-2. Results of average reduction (%) with collisions using a single set of filters.....	47
Figure 5-3. Results of average percentages of full reduction with collisions using a single set of filters.....	47
Figure 5-4. Results of average percentage reduction (%) with collisions for two sets of filters	48
Figure 5-5. Results of full reduction (%) with collisions for the tow sets of filters.....	49

Chapter 1 Introduction

A Distributed Database Management System (DDBMS) is a collection of computers, each at a different site, connected by some communication networks [1].

The database is distributed amongst all the sites which vary in size, function and complexity. Each site maintains a local database but can also participate in a global query. The advantages of a DDBMS is that data can easily be shared between sites, there is local control over local data and data replication ensures increased availability. The major disadvantage is that queries requiring data from more than one site entail higher overheads due to an increase in message passing and data transfers.

The optimization of general queries in a DDBMS is an important area of research. The problem, which must be solved, is to select the best sequence of database operations which will process the query and keep costs to a minimum. Finding the optimal solution is NP-hard [2] and so the approach taken in the literature is to find heuristic algorithms which give efficient but perhaps sub-optimal solutions.

The main approaches include algorithms based on joins, semijoins and Bloom-filters. The approach joins have been discussed in references [5], [31], [32], and [33]. The approach semijoins have been addressed in papers [7], [8], [9], [10], [11], and [35]. The approach based on bloom filters have been discussed in paper: [16], [17], [18], [19], [20], [21], [24], and [26]. Early work in the area concentrated on the use of joins. However, joins often require large amounts of data to be shipped between sites; as an operation, it has high complexity and frequently the use of a join can lead to higher data transmission costs. To solve these problems the semijoin was proposed and in most cases semijoin-based algorithms are very efficient. However, semijoins may entail more local processing and data transmission costs are still very high. Bloom-filters have been proposed as a method of executing semijoins at a much lower cost. In general, the use of Bloom filters greatly improves performance.

However, there is still the problem that, due to collisions in the filters which are built using a hash function, some relations in the query may not be reduced to the full extent possible. This means that data transmission costs are higher than they need be.

In this thesis we investigate the effect of collisions on the performance of a filter-based algorithm. We do so with 3 sets of experiments. First, we conduct some experiments with a simulated perfect hash function, where no collisions are possible, and we evaluate the performance of our algorithm. Then we conduct experiments where we simulate varying collision percentages, from 1% to 60%. The results show the effect of collisions on the performance of the algorithm. We found that for certain types of queries the effect of the collisions was significant. To improve performance for these queries we propose the use of two sets of filters, where each set simulates a different hash function yielding different collisions. In our last set of experiments we investigate the use of the two sets of filters and their effect on the performance of our algorithm.

The objective of our thesis is to investigate the hypothesis that the use of two sets of filters, in a filter-based algorithm, can significantly reduce the impact of collisions and improve the performance of the algorithm.

The thesis is structured as follows: Chapter 2 introduces some basic concepts and summarizes important research in this area. In Chapter 3 the algorithm is described and illustrated with a running example. Chapter 4 describes all of our experiments. The results are presented and discussed in Chapter 5. The thesis concludes with Chapter 6.

Chapter 2 Background

Distributed query optimization involves operations on distributed databases. The goal of distributed query processing is to minimize the cost of a distributed query. In this chapter, some definitions will be given and some representative approaches in the area of distributed query optimization will be presented. The approaches use joins, semijoins, and filters.

2.1 Definitions

Join is the operation which joins two tables together on the basis of common values in a common column. For example, given two relations, R_1 and R_2 , with attribute A which exists in both relations, the join of R_1 and R_2 over A is performed by concatenating tuples of R_1 with those of R_2 where the attribute value for A is equal.

The semijoin is defined as follows: Given two relations, R_1 and R_2 , with attribute A which exists in both relations, the semijoin of R_2 by R_1 over A is performed by projecting R_1 over A to get $R_1[A]$, shipping $R_1[A]$ to the site of R_2 , and joining $R_1[A]$ with R_2 .

In the query processing in relational databases, a filter is a bit array which is encoded with the information contained in a joining attribute. This encoding is usually done by hashing.

Hashing is a technique for providing fast direct access to a specific stored record on the basis of a given value for some field. A collision occurs when two or more distinct records hash to the same address.

The cost model is used to predict the cost of alternative execution plans for a query. There are two popular cost models: the total cost model and response time model [27]. The total cost model includes both the data transmission cost and the local processing cost. The response time cost model calculates the total execution time of the query from beginning to the calculation of the final result.

2.2 Approach I: Joins

Join is one operation used in distributed query processing. It joins two tables together on the basis of common values in a common column.

The features of join:

- The join is an essential operation in a query which involves two or more relations.
- The join operation is easy to understand, so it is simple to write the query using joins.
- A join is a computationally expensive operation.
- A join may increase the size of relation. This means that the relation resulting from a join may be much larger than the relations before the join.

In figure 2-1, we show an example of a join. Given two relations R1 and R2, both containing joining attribute B, the join of R1 and R2 is performed by concatenating tuples of R1 and R2 where the value of attribute B is equal for both tuples. The join of R1 and R2 results in a larger relation in this case.

Efficient processing of joins is important in distributed database systems where network transmission costs determine the efficiency of a particular method. Inefficient processing of distributed queries not only increases task duration, but can also degrade performance of the entire system if network congestion develops. Distributed joins can be expensive operations due to expensive data transmission costs.

Distributed joins are particularly expensive because a joining relation located at a remote site may be transmitted to the join site. For joins in which only a small percentage of tuples at a remote site are needed, transmission of an entire relation is clearly a waste of network resources.

R1	A	B	R2	B	C
	1	3		3	2
	2	3		3	4
	3	5		5	5
	4	5		6	6
	5	5		6	8

R1 ⋈ R2	A	B	C
	1	3	2
	1	3	4
	2	3	2
	2	3	4
	3	5	5
	4	5	5
	5	5	5

Figure 2-1. The join of R1 and R2 over attribute B

The representative algorithms which use joins are R* optimizer and the two-way join.

R* optimizer:

The R* optimizer [32] is an iterative algorithm which uses a dynamic programming approach. The goal of the R* optimizer is to reduce the total cost of a query. Joining strategies of two relations are generated, followed by strategies of three relations, strategies of four relations, and so on. A decision is made by the optimizer on the join order of the relations during its execution. The decision depends on the following factors [32]:

- Access method. A decision is made on whether to perform an index scan or a sequential scan of the relation. The least costly cost of the two alternatives is taken.
- Join method. The optimizer decides whether it is more beneficial to perform a merge join or a nested loop join, and chooses the cheaper of the two join methods.
- Join site. A decision is made by the optimizer as to which relations to ship to perform a join of two relations located at different sites.
- Inner table transfer strategy. Given a join of an outer relation (the relation formed from previous joins) and an inner table (the next relation to be joined), if the inner relation has been chosen to be shipped to the site of the outer relation for joining, the cheapest method for transferring the inner relation must be chosen. The two candidate methods are to fetch individual tuples as necessary or to ship the entire relation to the site of the outer relation.
- Ordered result delivery to query site. Finally, the optimizer must choose between sorting the result or planning the query to produce a sorted result. Also, the optimizer must choose whether to plan a query which ends at the query site or to ship the final result to the query site.

The R* optimizer has exponential complexity, since it performs an enumeration of strategies. The limitations of the performance evaluation are the restriction to two relation joins, the lack of evaluation using very large databases and no consideration of fragmentation.

Two-Way Joins:

The previous algorithm for processing distributed join queries did not consider fragmentation [33]. The two-way join is used for joining two fragmented relations. The two general strategies in processing the two-way join $R_1 \bowtie R_2$ are:

- Union the fragments of each relation, then join them.
- Join all fragments of R_1 with all fragments of R_2 , then union the results.

The qualification of fragment is a formulation of the properties common to all tuples in a fragment. By examining the qualifications of two fragments to be joined, some fragment joins can be eliminated from the execution strategy.

A four phase optimization framework for the two-way join is proposed by Chen and Li. The following summary is from [34]:

1. Join Graph Construction. A bipartite graph is used to represent the two-way join. The qualification of fragments is used to determine and remove empty fragment joins from the graph. The empty fragment joins are eliminated from the graph by removing the edge existing between them.

2. Join analysis Graph Construction. Using the join graph produced above, a join analysis graph is constructed. The weight corresponding to each vertex is a measure of the data transmission cost of the fragment or fragment join represented by the vertex. An algorithm is proposed which maps a join graph to a join analysis graph. When performing the mapping, the algorithm takes into consideration the data stored at the query site (which does not have to be transmitted) and other properties involved with performing fragment joins.

3. Determining a Minimum-Weight Vertex Cover. Given a join analysis graph, determining a minimum-weight vertex cover is analogous to determining a set of fragments and fragment joins such that the cost of data transmission is minimal. The authors prove that the problem of determining a minimum-weight vertex cover is NP-Complete. Using certain properties of a join analysis graph, the authors show how the graph can be reduced in size before the application of an enumeration algorithm for determining the vertex cover. Also, the authors propose a heuristic algorithm for determining the minimum-weight vertex cover. This algorithm eliminates the need for performing enumeration.

4. Final Processing. After determining a minimum-weight vertex cover, an existing copy of each selected fragment is chosen from the network for transmission. The decision as to which copies to use depends on the load at each site. Also, the site of each fragment

join must be chosen. An algorithm is presented to handle this. After the choices of existing fragments and join locations are made, the fragment joints are transmitted to the query site and united to finish the join. It should be noted that an algorithm for performing a fragment-to-fragment semijoin is proposed and utilized to reduce fragments before they are transmitted. Refer to [34] for details.

The authors have proved that most algorithms in that paper have polynomial complexity.

The algorithms proposed by Ahn and Moon select a subset of fragments and fragments join results to ship that reduces the data communication cost. The two algorithms are summarized from [33]:

- Greedy Heuristic. Algorithm GH consists of three phases. In the first phase, the number of fragments to be considered is reduced by applying derived theorems. In the second phase, the fragments to be transmitted are chosen. It is an iterative phase which determines the fragment with the highest net benefit at each iteration. This phase terminates when the net benefit becomes zero. The third and final phase consists of adding to the transmission schedule the shipment of fragment joints which are not already located at the query site.

- Single Path heuristic. Algorithm SPH consists of three phases. In the first phase, the number of fragments to be considered is reduced by applying derived theorems. In the second phase, the fragments to be transmitted are chosen. In algorithm SPH, this is also an iterative process. The difference here is that all fragments are considered in the order given. Any fragments with a net benefit are scheduled for transmission. The third and final phase consists of adding to the transmission schedule the shipment of fragment joints which are not already located at the query site.

2.3 Approach II: Semijoins

Early research in distributed query processing used the join operator query optimization plan. However the join operator involves large relations to be shipped to other sites. To reduce the cost of transmission, the semijoin operation was introduced.

The semijoin can be explained by the following example. Given relations R1 and R2, and a join attributes B that exists in both relations, the semijoin of R2 by R1 over B is illustrated by Figure 2-2 and executed as follows:

1. Project R1 over B to get R1[B]
2. Send R1[B] over to the site of R2
3. Perform $R1[B] \bowtie R2$

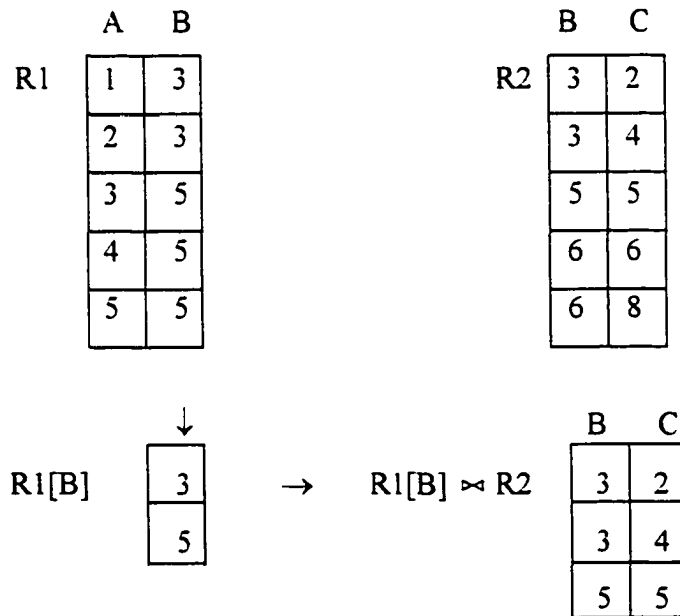


Figure 2-2. The Semijoin of R2 by R1 over attribute B

The advantage of the semijoin is that it will either leave the relation as the same size or it will reduce its size. A join may result in relations which are larger than the original relations.

The disadvantage of the semijoin is that it incurs more local processing costs since a projection must be performed each time a semijoin is executed.

Most work in distributed query processing using semijoins assumes the following three phases[27]:

1. Local processing. All relations involved in the query undergo selections and projections at the site where each resides.
2. Processing with semijoins. Following local processing, the relations undergo size reduction using semijoins.
3. Final processing. Following semijoin processing, the reduced relations are shipped to the query site to produce the final result.

The representative algorithms which use semijoins are the SDD-1 optimizer, the AHY algorithms, algorithm W, the two-way semijoin, and the one-shot algorithm.

SDD-1 Optimizer:

One of the first query optimization algorithms was for the SDD-1 (System for Distributed Database) [36]. The goal of the SDD-1 strategy is to reduce the data transmission cost, which is the dominant cost factor for executing a distributed query.

SDD-1 Algorithm is based on a three phase mechanism to process a distributed query.

1. In the initial processing phase, the appropriate selections and projections at the local sites are performed.
2. During the reduction phase, semijoins are used to reduce the size of the relations that do not satisfy the qualification of the query. The authors use a greedy algorithm which is iterative in nature. In each iteration, the most cost effective semijoin is chosen for execution until all semijoins have been considered. The query site is chosen in such a way that data transmissions are minimized.
3. In the final assembly phase, relations in the qualification component of the query are sent to a query site to produce the final join result.

The main drawback [41] of the SDD-1 algorithm is that it has not recognized the fact that certain relations involved in previous executed semijoins are not needed for further processing and can therefore be discarded.

The AHY Algorithms:

Apers, Hevner and Yao developed a collection of algorithms for minimizing the cost of distributed query processing. These algorithms are based on the semijoin and are referred to as Algorithm AHY [35], [34].

The algorithm AHY GENERAL promises the capability of optimizing general queries which contain more than one join attribute. Basically, Algorithm GENERAL first decomposes a general query into a collection of simple queries which are then processed by either Algorithm SERIAL (total cost version) or Algorithm PARALLEL (response time version). In each algorithm, semijoins are employed to reduce the size of relations by deleting those tuples which will not contribute to the final join result.

The Algorithm PARALLEL begins with the initial feasible solution, which is the parallel transmission of all required relations to the query site without taking reduction into account. The Algorithm PARALLEL searches for cost beneficial transmissions by shipping small relations to larger relations in order to join them.

The Algorithm SERIAL is a strategy which consists of an ordered transmission of each relation to the site of the next relation, where the join or semijoin is performed before the relation is shipped to the next site. The relations are shipped in increasing order of size. This algorithm attempts to minimize data transfers by constructing a schedule of semijoins for each relation such that the cost of reducing and shipping the relation is as low as possible.

There are many assumptions in the algorithm. It does not consider network factors such as line congestion, communication delays, etc. These assumptions are not valid for a real distributed database system.

Algorithm W:

The algorithm W proposed by Morrissey et al [42], [44], [43], [45] attempts to minimize the total amount of data transferred over the network during query processing. For each joining domain participating in the query, a reducer is created and used to reduce all relations. A reducer is a joining attribute constructed from the joining attributes which come from the same joining domain.

There are three steps in Algorithm W [42, 43, 44, 45]:

Step 1: Establish schedules for the construction of reducers.

Step 2: Examine the effects of reducers and review the schedule of those not used.

Step 3: Execute the schedule.

Algorithm W produces semijoin strategies with polynomial complexity of $O(mn)$, where m is the number of relations and n is the number of joining domains.

Two-Way Semijoin:

The two-way semijoin [46], [8] performs not only forward reductions as the semijoin, but also backward reduction which results in the reduction of both relations involved.

In [8], the authors propose a three phase processing of query using two-way semijoin program:

1. Forward Reduction and Local Processing. All forward semijoins are performed in conjunction with the initial local processing. All tuple connectors are generated during this process. A tuple connector is a representation of a relation, which consists of all joining attributes and some tuple identifier. Since tuple connectors only contain the information required for processing a query, they generally are smaller than their corresponding relation.

2. Backward reduction and collecting. The tuple connectors are joined to form the pipeline cache planner which is the relation consisting of tuple identifiers of all tuples required for the query. All joining attributes are projected out during this process.

3. Pipeline Execution. The pipeline cache planner is used for retrieving the necessary tuples and generating the final result.

The two-way semijoin can be more effective than the semijoin in reducing relation sizes in distributed query processing.

One-Shot Algorithm:

In [47, 48], it is argued that traditional sequential semijoins have several drawbacks including the loss of parallelism, processing overhead, loss of global semijoin optimization, and inaccurate semijoin reduction estimation. The authors propose a new algorithm called the one-shot semijoin.

The one-shot semijoin consists of three phases:

1. Projection phase. During this first phase, all joining attributes required for semijoins are obtained from the relations in parallel.
2. Transmission phase. In the second phase, the parallel transmission of all joining attributes occurs.
3. Reduction phase. Finally, all joining attributes are applied to their respective relations in parallel.

The advantages of this strategy include increased parallelism, no propagation of errors, reduced processing overhead, and the opportunity to apply global optimization to semijoins.

The algorithm presented uses hash tables for storing joining attributes, which allows for faster processing of semijoins during the reduction phase. The strategy has the trade-off between hash table size and reduction effectiveness.

2.4 Approach III: Filters

Semijoins are good at reducing relation sizes, but there are some limitations such as extra local processing costs and higher data transmission costs. To resolve these problems, the filter-based approach has been employed. In the distributed query processing, a filter is a bit array which is encoded with the information about the values contained in an attribute. The filters are used to identify tuples which cannot belong to the result relation so that the amount of data transmission is reduced.

Due to the size of the filter, it is cheaper to ship a filter over network lines than to transmit a relation or a semijoin projection.

The disadvantage [27] of using filters is that collisions occur as a result of two or more attribute values hashing to the same address in the array. This results in tuples which are not required for the query being shipped to the final site because the attribute value accidentally passed the filter test.

The term “Bloom filter” comes from Bloom [51]. A Bloom filter [3] is simply an array of bits which functions as a very compact representation of the values of a join attribute. The use of a Bloom filter can achieve the same result as a semijoin but at a much lower cost. The filter is constructed as follows:

1. First, construct an array and set all bits to zero.
2. Then, for each value of the join-attribute use a hash function. Hashing is the procedure of applying the hash function to a key or attribute value to produce an address in a data structure.
3. Finally, for each address produced, set the corresponding bit in the array to one.

The Bloom filter algorithm has lower local processing and data transmission costs. However, the problem is that collisions occur. A collision means that two different attribute values may be hashed into the same bit address. A consequence is that tuples that

should be eliminated from the relation are not and hence we get larger relations and higher transmission costs.

The Bloom filter based algorithms are described as follows.

In [25], Tseng and Chen propose an algorithm based on hash semijoins. The purpose of this work is to minimize the cost of performing a semijoin. The implementation of this hash semijoin is summarized from [25] as follows:

Given two relations R_i and R_j , with common joining attribute A :

1. Calculate the bit array size and initialize to zero.
2. For each join attribute value in $R_i[A]$, use d hash functions to produce d addresses in the bit array and set the corresponding bits to 1.
3. Ship the bit array to the site of R_j .
4. For each join attribute value in $R_j[A]$, produce d addresses in the bit array using the same d hash functions. If all d addresses in the array are set to 1, the tuple is kept; otherwise, reject the tuple.

Morrissey and Ma [18] present Algorithm X which uses Bloom filters to reduce query response time. Algorithm X has been designed to process general queries of arbitrary size, and relies on no other relational operators in the process.

The basic idea of Algorithm X is to apply all filters to all relations, concurrently. The filter sizes are relatively small and therefore the cost is relatively low. Each relation is processed at most two times, once to construct the filters and once when the reducing filters are applied.

The description of Algorithm X from [18] is as follows.

Begin

Send all relevant filters to the relation R which is to be reduced.

Repeat

read a tuple T ;

hash on all join-attribute values in R .

```
        if there is a hit in every filter,  
        then keep tuple as part of the reduced relation R,  
        else discard tuple;  
        read next tuple T;  
    Until all tuples have been processed.  
End
```

Algorithm X clearly has a superior performance due to the fact that the semijoins are replaced with the use of filters which are cheaper to use. The application of all filters concurrently will not increase the response time and it will certainly decrease the local processing cost.

The disadvantage of Algorithm X is the assumption of a perfect hash function. This is an unrealistic assumption.

Morrissey and Osborn [19] propose an algorithm for processing general queries which minimizes the cost of data transmissions. This algorithm uses reduction filters to achieve the same reduction benefits as a semijoin strategy but at a lower cost.

A reduction filter is a bit array which serves to concisely represent a common joining attribute. The reduction filter serves the same purpose as the semijoin, which is to filter out tuples which will not be part of the final result.

The figure 2-3 illustrates a reduction filter. Given two relations, R1 and R2, both containing joining attribute b, the filtering takes place as follows. First, a filter is created for attribute b of R1 and is transmitted to the site of R2. The filter is a bit array. The bits are set to 1 if the attribute values are hashed to the corresponding addresses. In this example, the attribute b has values 3 and 5, so we set the third and fifth positions in filter b to 1. Second, the reduction filter is applied to R2 in the following manner. For each tuple in R2:

1. Hash on the value of attribute b.
2. Test for the presence of a 1 at the address produced.
3. If a 1 bit is found, keep the tuple for the final result.

4. Otherwise, discard the tuple.

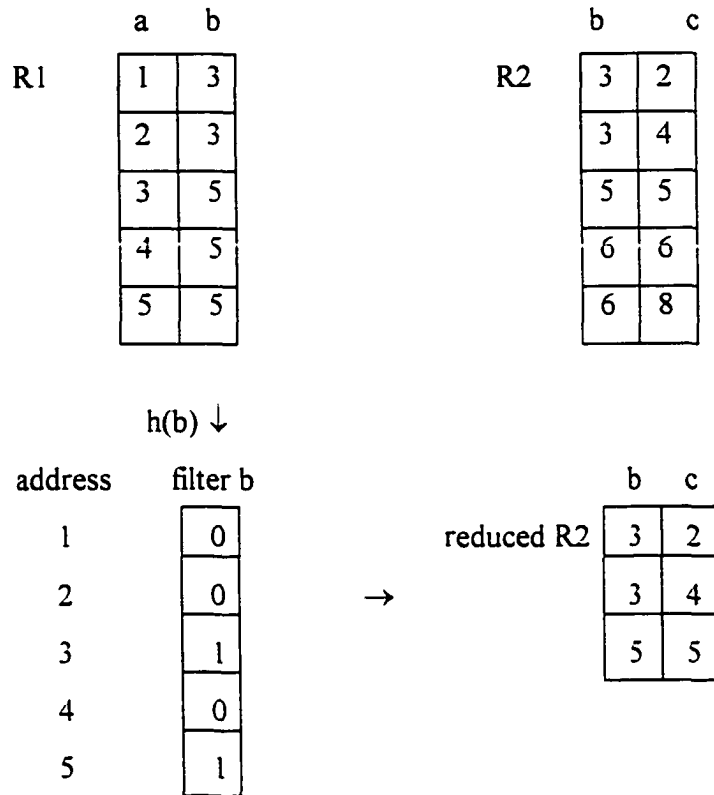


Figure 2-3. The reduction of R2 by the reduction filter for attribute b

Due to their size, reduction filters are very inexpensive to transmit between sites compared to the attribute projections required for semijoins.

The limitation of this algorithm is the collisions which occur in the reduction filters. The results [27] of the experiments show that as the number of collisions increases the percentage of queries that achieve full reduction decreases. Full reduction means that all non-participating tuples of all relations involved in a query are eliminated by the algorithm.

2.5 Conclusions

In this chapter, we discuss the different approaches for distributed query processing. The goal of the algorithms in this area is to minimize the cost of query processing in distributed databases. The previous research focuses on the three categories: joins, semijoins, and filters.

The join operator was utilized in the early work of this area. The join needs to ship large relations to other sites and has high computational complexity. To solve these problems, the semijoin is employed. Semijoins have the advantage over the joins that no increase in data sizes occurs from their use. However, a semijoin needs more local processing such as projection and higher data transmission costs than necessary. To improve the distributed query processing, the filter-based approach is utilized.

The filters have better performance such as relation size reduction and response time than others. Although the filter-based algorithm has lower local processing costs and less data transmission than semijoins, it suffers from collisions. This is one of the reasons why the use of filters for processing distributed queries has not been widely accepted. In our research, we investigate how collisions affect the performance of the algorithm and how performance can be improved given that collisions occur.

Chapter 3 The Algorithm

In this chapter, we present our algorithm. This algorithm uses Bloom filters to achieve the same reduction benefits as a semijoin strategy but at a lower cost.

3.1 Details of the Algorithm

The algorithm for processing each query consists of three phases:

1. The data structures are built according to the query. The data structures include an adjacency matrix, the adjacency list, the inverted list, a relation queue and a filter list.
2. We use the data structures built in phase 1 to construct filters and reduce the relations
3. The relation queue is used to decide which relations must be further processed.

The details of the three phases are given as follows.

Phase 1. The data structures are constructed:

1. We build the adjacency matrix using the name of relations and the names of joining attributes in the query.
2. The adjacency list is constructed from the adjacency matrix. The adjacency list contains the information such as the relation name, the indegree which is number of joining attributes for this relation, and the attribute name.
3. The inverted lists are built from the adjacency matrix. The inverted lists represent the attribute name and relevant relation names. The inverted lists are used to decide which relation should go back on the queue.
4. We build the relation queue which contains the relations which need to be further processed.

5. We construct the filter list which is a simple list of filters already available.

Phase 2. We construct the reduction filters and reduce the relations with the filters using the data structures built in phase 1. The iterative process of this phase is described below:

1. From the adjacency list, we select the relation with the lowest indegree. We will denote this relation as R_i .
2. We determine if reduction filters for any of the joining attributes exist, and apply them to R_i to reduce it further.
3. While processing R_i , construct new reduction filters for all joining attributes contained in R_i .
4. Determine which relations to place back on the queue. The 'filter rule' states that a relation is placed on the queue if:
 - a. The reduction filters for any of its joining attributes have changed after being applied to R_i ,
 - b. it is not R_i ,
 - c. it is not already on the queue, and
 - d. it has been processed already.
5. We reduce the indegree of the relation in the adjacency list and mark this relation as processed.
6. Repeat steps 1 to 5 in this phase for all relations in the query.

Phase 3. The processing of the relation queue is described below:

1. We remove relation R_i from the front of the queue.
2. We apply all reduction filters for all joining attributes contained in R_i to reduce the relation further.
3. Determine which relations to place back on the queue. The same 'filter rule' used for placing relations on the queue in Phase 2 is also used here.

4. Repeat steps 1 to 3 in this phase until the relation queue is empty.

3.2 An Example of Using the Algorithm

We explain how the algorithm works in detail using the following example.

In this example, the query has five relations which must be joined. These five relations are shown in figure 3-1. The algorithm executes the query efficiently by using filters.

R1	<u>A B C</u>	R2	<u>A D E F</u>		
	1 2 4		1 2 4 3		
	2 2 5		2 2 5 5		
	3 3 6		3 3 6 7		
			4 4 7 9		
R3	<u>B D E</u>	R4	<u>C G</u>	R5	<u>F H</u>
	2 2 4		4 2		3 4
	3 3 6		5 3		3 7
	4 7 9		7 4		7 7

Figure 3-1. The five relations of the example

Phase 1. We construct the data structures.

1. We build the adjacency matrix of figure 3-2 using the relation names and attribute names of these five relations. The columns are labeled by the attribute name. The rows are labeled by the relation names. Every cell of the matrix is initialized to 0. The matrix cell is marked 1 if the relation has the joining attribute.

	A	B	C	D	E	F	G	H
R1	1	1	1	0	0	0	0	0
R2	1	0	0	1	1	1	0	0
R3	0	1	0	1	1	0	0	0
R4	0	0	1	0	0	0	1	0
R5	0	0	0	0	0	1	0	1

Figure 3-2. The adjacency matrix

2. We construct the adjacency lists of figure 3-3 for each relation from adjacency matrix. The first node contains the relation name and the indegree of this relation. The rest of the nodes contain the relevant relation name and joining attribute name.

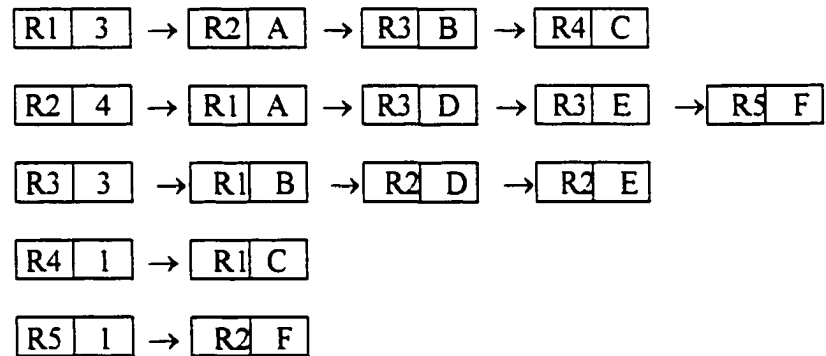


Figure 3-3. The adjacency lists

3. Build inverted lists of figure 3-4 from the adjacency matrix. The first node has the attribute name and the remaining nodes consist of the relation names which have this attribute. The attributes G and H are not joining attributes, so they are not in our inverted lists.

$A \rightarrow R1 \rightarrow R2$
 $B \rightarrow R1 \rightarrow R3$
 $C \rightarrow R1 \rightarrow R4$
 $D \rightarrow R2 \rightarrow R3$
 $E \rightarrow R2 \rightarrow R3$
 $F \rightarrow R2 \rightarrow R5$

Figure 3-4. The inverted lists

4. The relation queue consists of the relations which need further processed according to 'filter rule'. This queue is built by adding a relation to it during query processing.
5. The filter list contains the filters which are already available.

Phase 2. We construct the reduction filters and reduce the relations with the filters using the data structures built in phase 1.

1. In the adjacency lists, R4 and R5 have the lowest indegree which is 1.

We select R4 and construct a filter for attribute C. We simulate a perfect hash function by setting bits 4, 5, and 7 in the filter, representing the corresponding attribute values 4, 5, and 7. This is illustrated by the figure 3-5.

After filter C is constructed, it is placed on the filter list. Nothing goes on the relation queue according to the 'filter rule'.

The adjacency lists are updated. The indegrees of R1 and R4 decrease by 1. The relation R4 is marked as processed.

R4	C	G	h(value) → address	filter C
	4	2	1	0
	5	3	2	0
	7	4	3	0
			4	1
			5	1
			6	0
			7	1
			8	0
			9	0
			10	0

Figure 3-5. Step 1 of the example

2. We select R5 which has lowest indegree now. No filter for attribute F is available. We produce filter F by setting bits 3 and 7. Filter F is placed on the filter list.

Nothing goes on the relation queue according to the 'filter rule'.

The adjacency lists are updated. The indegrees of R5 and R2 decrease by 1. The R5 is marked as processed.

3. We select R1 whose indegree is 2 now. The filter C is available. We reduce R1 with filter C and produce filters for attribute A and B at the same time.

Reduced R1	A	B	C
	1	2	4
	2	2	5

We produce filter A by setting bits 1 and 2 as 1. We build filter B by setting bit 2. We insert filter A and B into filter list. The filter list contains filters C, F, A, and B.

The filter for C has changed. The inverted list and 'filter rule' are used to decide which relations go on the relation queue. R4 goes on the queue.

The adjacency lists are updated. The indegrees of R1, R2 and R3 are decreased by 1. R4 has been processed, so its list is not changed.

R1 is marked as processed.

4. We select R2 and check the filter list. The filter A and F are available, so R2 is reduced using filter A and F.

Reduced R2	A	D	E	F
	1	2	4	3

We build filter D by setting bit 2 and produce filter E by setting bit 4.

The filters D and E are inserted to the filter list.

The filters E and F have changed, so R1 and R5 go on the queue according to the inverted list and 'filter rule'. The relation queue contains R4, R1, and R5.

The indegree of R3 decreases by 1.

R2 is marked as processed.

5. We select R3. All filters exist, so R3 is reduced with filters B, D, and E

Reduced R3	B	D	E
	2	2	4

The filters have not changed, so no relation is added to the queue.

R3 is marked as processed.

The five relations have been marked as processed now.

Phase 3. We process the relation queue which contains R4, R1, and R5.

1. We remove R4 from the queue and process it.

The filter C is available. We reduce R4 with filter C.

Filter C has not changed. Nothing goes on the queue.

2. We remove R1 from the queue and process it.

All filters are available. We reduce R1 with filters A, B, and C.

Reduced R1	A	B	C
	<hr/>		
	1	2	4

Filter C changes. R4 goes back on queue using the inverted list and the 'filter rule'.

3. We remove R5 from the queue and process it.

The filter F exists. We reduce R5 with filter F.

Reduced R5	F	H
	<hr/>	
	3	4
	3	7

The filter F does not change. Nothing goes on the queue.

4. The relation queue only has R4 now.

we reduce R4 with filter C.

Reduced R4	C	G
	<hr/>	
	4	2

The queue is empty now. The process stops.

Chapter 4 Experiments

In this chapter, we present our experimental system and describe the experiments without collisions, experiments with collisions using a single set of filters and the experiments with collisions using two sets of filters for evaluating the algorithm.

4.1 Experimental System

The experimental system includes the query-relation generator, our Bloom filter algorithm, the full reducer, the analysis programs and the execution scripts.

The query-relation generator was programmed by the Database Research Group of the University of Windsor. Some details are given later in this chapter.

The Bloom filter algorithm has been described in chapter 3 and implemented in C++. We simulate a specific percentage of collisions for the values of joining attributes.

The full reducer is an algorithm that fully reduces all relations involved in a query by eliminating all non-participating tuples from the relations.

In our experiments we constructed 15,000 different queries and relations as input. The queries and relations vary in the number of relations, the number of joining attributes, the relation size, the domain size, the selectivity and the connectivity. In order to effectively evaluate our algorithm with such a large number of divers queries, we split the queries into runs, each run contains 100 queries.

The relations are reduced by our algorithm and the full reducer. We do the experiments under the assumption of a perfect hash function and a specific percentage of collisions. We analyze the results under different conditions using analysis programs.

The experiments are carried out using the execution scripts which contain a collection of commands in order to fulfill the tasks.

4.1.1 Rationale of the Experiments

Most previously proposed algorithms have not been objectively evaluated. The performance of the heuristics have been evaluated by comparison with another heuristic. In this way, it can not be determined how close an algorithm achieves full reduction for the relations. Some previously proposed algorithms have been evaluated theoretically by a time complexity analysis. However, a theoretical evaluation does not determine how the algorithm will perform with real queries.

To evaluate our algorithm objectively, we compare the performance of our algorithm with the performance of a full reducer. By this approach, our algorithm will be evaluated to determine how close it achieves full reduction under various conditions. This approach is objective, so it is better.

The major problem when using Bloom filters is collisions. In order to investigate how the collisions affect the performance of the algorithm, we first do some the experiments under the assumption of a perfect hash function. Then we repeat the experiments for the different percentages of collisions using a single set of filters and then using two sets filters. In this way, we can compare the results and make the conclusions.

4.1.2 Queries and Relations

The experimental system takes different queries and relations as input to evaluate our algorithm.

Each query consists of an arbitrary number of relations and an arbitrary number of joining attributes. We evaluate our algorithm by using different query types which are the combination of a relation count and an attribute count. For example, the query type 3-2 represents a query with three relations and two joining attributes.

The relations vary in the size, the attribute domain size, the selectivity and the connectivity.

The relation size is the number of tuples in a relation.

The attribute domain size is the total number of distinct attribute values available.

The selectivity is defined as the ratio of distinct attribute values over the attribute domain size.

The connectivity is an approximate ratio of the number of joining attributes appearing in all relations of the query over the total number of possible join attributes that can appear in the query.

The queries and relations are generated by C programs which are described in [6]. The C programs are `create_query.c` and `relbuilder.c`. We discuss these two C programs which are needed by our experimental system below.

Program `create_query.c`: This program generates a query. The input includes the number of relations and the number of common join attributes. The output of this program consists of the file 'dbstats' which is the database statistics, file 'Rel' which is relation data and file 'domains' which is the domain size.

The file 'dbstats' contains the number of relations, the number of common joining attributes, the relation size, attribute size and the selectivity.

For each relation specified in the query, a 'Rel' file is produced and is used to generate relations. The figure 4-1 is an example of this file.

```
Rel 0
242 2 0 186 222 1 155 218
```

Figure 4-1. 'Rel' file for a relation

There are 8 numbers in this file. The first number (242) is the size of the relation. The second number (2) is the number of join attributes. The third number (0) represents attribute 0. The fourth number (186) is the size of attribute 0. The fifth number (222) is the domain size of attribute 0. Similarly, the attribute 1 has size 155 and domain size 218.

Program relbuilder.c: This program produces a relation based on the data generated in create_query.c. The input to relbuilder.c is a number of relations involved in the query. The output is a relation, which contains the required number of tuples, the number of joining attributes and the joining attribute labels.

4.1.3 Full Reducer

To evaluate the algorithm described in chapter 3 objectively, we need to compare the performance of our algorithm to a full reducer. For this reason, we have developed a full reducer program.

The full reducer program includes two steps.

- Step 1. Join all relations required by the query to get the result. We use a nested loop join.
- Step 2. Obtain the reduced relations by projecting the attributes of each relation from the joining result.

We explain how the full reducer works with following example. In this example we use the five relations given in figure 3-1.

First, we join the five relations.

R1 \bowtie R2:	<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>
	1	2	4	2	4	3
	2	2	5	2	5	5
	3	3	6	3	6	7

R1 ⋈ R2 ⋈ R3:	<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>
	1	2	4	2	4	3
	3	3	6	3	6	7

R1 ⋈ R2 ⋈ R3 ⋈ R4:	<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>	<u>G</u>
	1	2	4	2	4	3	2

R1 ⋈ R2 ⋈ R3 ⋈ R4 ⋈ R5 :

	<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>	<u>G</u>	<u>H</u>
	1	2	4	2	4	3	2	4
	1	2	4	2	4	3	2	7

Figure 4-2. Result of joining of five relations

Second we obtain the fully reduced relations by projecting the attributes of each relation from the result of joining the five relations which is in figure 4-2.

Fully reduced R1:	<u>A</u>	<u>B</u>	<u>C</u>
	1	2	4

Fully reduced R2:	<u>A</u>	<u>D</u>	<u>E</u>	<u>F</u>
	1	2	4	3

Fully reduced R3:	<u>B</u>	<u>D</u>	<u>E</u>
	2	2	4

Fully reduced R4:	<u>C</u>	<u>G</u>
	4	2

Fully reduced R5:	<u>F</u>	<u>H</u>
	3	4
	3	7

Figure 4-3. Results of fully reduced relations

Notice that in each relation we only have contributive tuples, all non-contributive tuples have been eliminated. To evaluate our algorithm, we compare its reduced relations to the fully reduced relations for each query.

4.1.4 Analysis Programs

To evaluate the performance of our algorithm precisely, we have produced the analysis programs to collect results generated by our algorithm and the full reducer. The analysis programs calculate the average percentage of reduction achieved by our algorithm and the percentage of queries that achieve full reduction. The calculation of average reduction is based on the following formula.

$$\text{average reduction (\%)} = [(\text{total size} - \text{reduced size}) / (\text{total size} - \text{full size})] * 100$$

In the formula above, the total size denotes the total size of all relations related to the query before reduction, the reduced size represents the size of the relations after reduction using the algorithm, the full size denotes the size of the relations after being fully reduced by the full reducer.

For example, given the following data:

total size = 36

reduced size = 18

full size = 14

average reduction (%) = $[(36 - 18) / (36 - 14)] * 100 = 81.8\%$

The full reduction (%) represents the number of queries which are fully reduced by the Bloom filter algorithm, this number of queries is out of 100 queries.

4.2 Experiments without Collisions

The experiments without collisions are carried out under the assumption of a perfect hash function. We simulate a perfect hash function by using the attribute values (all are integers) as the address. All attribute values hash to the addresses specified by the values with the perfect hash function. No collisions can occur with this method. For example, an attribute value of 8 will hash to address 8 in the Bloom filter.

The experiments without collisions will determine how well our algorithm performs under the assumption of a perfect hash function.

In the experiments, we split the queries into runs, each run containing 100 queries. Each relation in the query consists of 200 to 600 tuples. The attribute domain contains 150 to 250 distinct values. The minimum selectivity is 0.5. The minimum connectivity is 75%. (The selectivity and connectivity have been defined in section 4.1.2 Queries and Relations).

4.3 Experiments with Collisions Using a Single Set of Filters

Because the Bloom filters are encoded using hash functions, collisions occur. A collision happens when two or more attribute values hash to the same address.

The purpose of the experiments with collisions is to determine how the collisions affect the performance of the algorithm.

4.3.1 Simulating Collisions

In the experiments with collisions, the algorithm is evaluated at specific percentages of collisions: 1%, 5%, 10%, 20%, 30%, 40%, 50%, and 60%. The percentages of collisions are usually under 60% in the real world, so we simulate collision range from 1% to 60%.

To simulate the specific percentage of collisions, we use the following method. Given a relation which has a common joining attribute B, its active domain is determined. The active domain of common joining attribute B is the set of values taken from domain of B. We obtain the active domain of joining attribute B by projecting the relation over attribute B. If we simulate $c\%$ of collisions in active domain, we randomly select $c\%$ values in attribute B. These values will result in collisions in the filter. We call these values collision values. For each collision, its collision address is determined. The collision addresses are taken from set

$$\{ \text{active domain (B) - collisions} \}$$

To produce the $c\%$ collisions for all joining attributes related to the query, we apply this method to all joining attributes.

The following is an example of simulating the collisions. This example is illustrated in figure 4-4. Given a common joining attribute B, we have the active domain which has 10 values { 5, 3, 4, 7, 2, 1, 6, 9, 10, 8 }. If we simulate 30% collisions, we randomly select three collisions out of the ten values in the active domain. Let us say the values 3, 1, and 8 are chosen. The remaining values of active domain are { 5, 4, 7, 2, 6, 9, 10 }. Then, the collision addresses are selected from the remaining values of the active domain. We set the bits corresponding to the remaining values to 1 in the Bloom filter. To guarantee the collisions occur, we hash the three collisions 3, 1, and 8 to the addresses of the remaining values. Then we get the collision addresses from the remaining values. Let us choose 4, 7,

and 9 as collision addresses. Therefore, the collision 3 will hash to the collision address 4 in the Bloom filter, the collision 1 will hash to the collision address 7, and the collision 8 will hash to the collision address 9. The rest of the attribute values will hash to the addresses represented by the values.

active domain	h(value) →	address	Bloom filter	collisions
5		1	0	
③		2	1	
4		3	0	
7		4	1	← 3
2		5	1	
①		6	1	
6		7	1	← 1
9		8	0	
10		9	1	← 8
⑧		10	1	

Figure 4-4. Simulating collisions

4.3.2 Problem Caused by Collisions

The major disadvantage of using Bloom filters is the collisions. The consequence of collisions is that tuples that should be eliminated from the relation are not and hence we get larger relations and higher costs.

The following example illustrates how the collisions affect the reduction of a relation using a Bloom filter. We have three relations R1, R2, and R3 in figure 4-5. They have the common joining attribute B. We will join these three relations by performing $R1 \bowtie R2 \bowtie R3$.

R1	A	B	R2	B	C	R3	B	D
	1	2		1	2		1	3
	2	4		3	4		2	5
	3	5		4	5		5	8
	6	8		6	7		6	2
	8	9		8	9		7	4

Figure 4-5. Relations for illustration the effect of collisions

To simulate 40% collisions, we randomly choose 40% values of attribute B in R1. Let us say the values 2 and 8 are chosen as the collisions. The remaining values of the attribute B of R1 are 4, 5 and 9, so the fourth, fifth, and ninth positions of filter B are set to 1. The filter B and the collisions are shown in figure 4-6.

R1	A	B	h(value) → address	filter B	collisions
			1	0	2
1		2	2	0	8
2		4	3	0	
3		5	4	1	
6		8	5	1	
8		9	6	0	
			7	0	
			8	0	
			9	1	
			10	0	

Figure 4-6. Collisions in Bloom filter B

We use filter B to reduce R2. We have value 4 in attribute B of R2 and the fourth position of the filter B is 1, we have value 8 in attribute B of R2 and 8 is one of the collisions, so we keep these two tuples in R2. We do not have values 5 and 9 in attribute B of R2, so we update fifth and ninth positions of filter B to 0. The reduced R2 and updated filter B are shown in figure 4-7.

Reduced R2	B	C	updated	address	filter B	collisions
			→	1	0	2
	4	5		2	0	8
	8	9		3	0	
				4	1	
				5	0	
				6	0	
				7	0	
				8	0	
				9	0	
				10	0	

Figure 4-7. Reduced R2 and updated filter B

Next, we use filter B to reduce R3. In attribute B of R3, we have value 2. In figure 4-7, the value 2 is a collision value, so we keep this tuple. The tuple in R3 of figure 4-8 should be eliminated if there is no collision since the attribute B of R2 does not have value 2. Therefore, we need ship the reduced R3 over the network to complete the query because of the collisions. This is the consequence of collisions which affect the performance of the algorithm in reduction of relation. The attribute B in R3 does not have value 4, so we update the fourth position of filter B to 0. The reduced R3 and updated filter B are shown in figure 4-8.

Reduced R3	B D	updated	address	filter B	collisions
		→	1	0	2
	2 5		2	0	8
			3	0	
			4	0	
			5	0	
			6	0	
			7	0	
			8	0	
			9	0	
			10	0	

Figure 4-8. Reduced R3 and updated filter B

4.4 Experiments with Collisions Using Two Sets of Filters

The experiments in this section are designed for evaluating the performance of our algorithm under a specific percentage collisions using two sets of filters. The experimental assumptions such as collision percentages, query types, and relation size are the same as in the previous sections. We introduce the use of two sets of filters in this section.

In our experiments, we notice that the performance of the Bloom filters on the queries with two joining attributes is worse than the others with more joining attributes. When a query contains few joining attributes, the filtering effect of the Bloom filters is hindered. This is illustrated by the following example.

In figure 4-9, given two relations R1 and R2, both contain joining attributes A and B. We will show the results produced by a full reducer and our algorithm.

R1	A	B	R2	A	B
	1	5		1	4
	3	4		3	5
	5	2		5	6
	6	6		6	2
	2	1		2	1

Figure 4-9. Example relations

By using full reducer, we first join these two relations and obtain the following result:

A	B
2	1

The fully reduced R1 and R2 are:

R1	A	B	R2	A	B
	2	1		2	1

Figure 4-10. Fully reduced relations

Suppose the algorithm processes R1 first. The filters for attributes A and B in R1 are created: A: 1, 2, 3, 5, 6 and B: 1, 2, 4, 5, 6. When the filters for A and B are applied to R2, we have the following result:

R2	A	B
	1	4
	3	5
	5	6
	6	2
	2	1

Figure 4-11. Non reduction of R2

No reduction has occurred in R2 as we see in figure 4-11. The reason why no reduction occurs in R2 is the following. For each tuple in R2, the attribute values for A and B are being hashed and set in each filter by two different tuples in R1. For example, the first tuple in R2 has A equal to 1 and B equal to 4. Although no matching tuple exists in R1, the first tuple in R1 contains A equal to 1 while the second tuple in R1 contains B equal to 4. Both of these values will be hashed and set to 1 in their respective filters. When the first tuple of R2 is tested, it passes the filters because both of its attribute values hash to a 1 bit in their respective filters.

Although this hindrance in reduction affects queries with few joining attributes and relations, it does not appear to be a problem with the queries containing a higher number of joining attributes and relations. These results show a higher number of joining attributes decreases the chance of a tuple false passing all necessary filters.

We will do some experiments on the query types 3-2, 4-2, 5-2, and 6-2.

4.4.1 Two Sets of Filters

From the previous section, we notice that collisions affect the performance of the algorithm. The collisions are the essential problem in using Bloom filters. Our goal is to reduce the collisions and have smaller relations to ship over the network to complete the given query. In this section we investigate how to reduce the collisions by using two sets of filters.

We use two sets of filters, that is, two filters for each joining attribute. We need two hash functions for the two filters. Note that two hash functions would give two sets of collisions. We simulate the two sets of collisions based on specific percentage of collisions. In this way, we can analyze the results under specific percentage of collisions. We utilize the two sets of filters to filtrate the collisions. The two sets of collisions are randomly selected from active domain based on specific percentage of collisions. The two sets of collisions may have overlaps.

The two sets of filters filtrate the collisions. We have 'rules for two sets of filters' as follows:

The tuple should be kept if the value in this tuple is one of the following three cases:

- a. The value is in both sets of collisions.
- b. The value is in one set of collisions and in one filter.
- c. The value is not in any set of collisions, but in one filter.

In the following example, we explain how to build two sets of filters and how two sets of filters reduce the collisions. We use the same relations R1, R2, and R3 in figure 4-5 of section 4.3.2 (Problem Caused by Collisions). These three relations have common joining attribute B.

To simulate 40% collisions for two sets of filters, we randomly choose 40% values of attribute B in R1 as our first set of collisions. Let us say the values 2 and 8 are chosen. The first filter is constructed by setting fourth, fifth, and ninth positions to 1. Similarly, we build the second set of collisions by selecting 40% values of attribute B in R1. Let us say the values 4 and 9 are chosen. The second filter is constructed by setting second, fifth, and eighth positions to 1. This is shown in figure 4-12.

R1						
A	B	address	first filter	first collisions	second filter	second collisions
		1	0	2	0	4
1	2	2	0	8	1	9
2	4	3	0		0	
3	5	4	1		0	
6	8	5	1		1	
8	9	6	0		0	
		7	0		0	
		8	0		1	
		9	1		0	
		10	0		0	

Figure 4-12 Two sets of filters

We reduce R2 using two sets of filters and the 'rules for two sets of filters'. We have value 4 in attribute B of R2 and the fourth position of the first filter is 1 so we keep this tuple. We have value 8 in attribute B of R2. In figure 4-12, the value 8 is one of the first collisions and the eighth position of second filter is 1, so we keep the tuple according to our 'rules for two sets of filters'. We do not have value 5 and 9 in attribute B of R2, so we change fifth and ninth positions of the first filter from 1 to 0. We do not have value 2 and 5 in attribute B of R2, so we update second and fifth positions of the second filter from 1 to 0. The reduced R2 and updated two sets of filters are shown in figure 4-13.

Reduced R2

B	C	address	first filter	first collisions	second filter	second collisions
		1	0	2	0	4
4	5	2	0	8	0	9
8	9	3	0		0	
		4	1		0	
		5	0		0	
		6	0		0	
		7	0		0	
		8	0		1	
		9	0		0	
		10	0		0	

Figure 4-13 Reduced R2 and updated filters

Next, we reduce R3 with the updated filters. In attribute B of R3, we have value 2. In figure 4-13, value 2 is a value of the first collisions, but the second positions of first filter and second filter are 0. By the 'rules of two sets of filters', the tuple with value 2 in attribute B of R3 is reduced. The other tuples in R3 are also reduced because they are not in the filters and collisions. Therefore, we do not need to ship the R3 over the network to complete the query. In this way, we improve the performance of our algorithm using two sets of filters to reduce the collisions.

Chapter 5 Results

In this chapter, we present the results which are divided into three groups. The result group one is for the results without collisions. The result group two is for the results with collisions using a single set of filters. The result group three is for the results with collisions using two sets of filters. We compare the results of these three groups to show that two sets of filters can reduce the number of collisions and improve the performance of the algorithm.

5.1 Results without Collisions

The main purpose of the experiments without collisions is to determine how well the algorithm performs under the assumption of a perfect hash function.

The figure 5-1 presents the results without collision.

Query Type	Average Reduction (%)	Full Reduction (%)
3 - 2	93.44	50
4 - 2	97.12	68
5 - 2	99.29	88
6 - 2	99.79	90
Average of the Column	97.41	74

Figure 5-1. Results without collisions

In figure 5-1, the definition of query type is given in section 4.1.2 (Queries and Relations), the Average Reduction (%) and the Full Reduction (%) are described in section 4.1.4 (Analysis Program), the results are based on the experimental assumption described in section 4.2 (Experiments without Collisions) and each data is the average result of 100 queries.

The results without collision show that the algorithm achieves substantial reductions in the sizes of the relations. This is shown in the second column of the table in

figure 5-1. On average for this column, approximately 97.41% of all tuples not required for the final answer of the query are eliminated from the relations.

In the third column of the table in figure 5-1, the results of full reduction of query types 5-2 and 6-2 are good. They are 88% and 90% respectively.

Queries of types 3-2 and 4-2, in many cases, have substantially lower amounts of data reduction than queries of other types. Also, queries of types 3-2 and 4-2 are the least likely to be fully reduced.

5.2 Results with Collisions Using a Single Set of Filters

The results without collisions show that, on average, the algorithm achieves significant reduction of unneeded data in query relations, and achieves an acceptable percentage of fully reduced queries. However, the experiments without collisions do not consider the effect of collisions on the performance of the algorithm, and the collisions are the major problem when using Bloom filters.

In this section, we present the results of the experiments under the specific percentages of collisions using a single set of filters. We compare the results in this section to the results without collisions in order to show the fact that the collisions affect the performance of the algorithm.

The results of the experiments are based on the assumptions described in section 4.3 (Experiments with Collisions Using a Single Set of Filters). We evaluate the algorithm under the assumption of collisions. We simulate the specific percentages of collisions at 1%, 5%, 10%, 20%, 30%, 40%, 50% and 60%.

In figure 5-2, we present the results of average reduction (%) with collisions. Each data is the average result of 100 queries. The first row represents the specific percentages of collisions. We do the experiments on the query types 3-2, 4-2, 5-2, and 6-2. The last column is the average data of the different query types. The last row represents the average data of different percentages of collisions. The data at right bottom corner is the average result of all data.

collision %	1	5	10	20	30	40	50	60	average
3 - 2	89.27	86.65	87.56	87.10	84.44	78.52	80.39	70.12	83.59
4 - 2	94.73	94.29	94.63	92.88	92.43	90.49	87.88	86.35	91.92
5 - 2	97.65	97.22	98.58	97.80	96.53	96.06	93.57	92.23	96.42
6 - 2	99.14	98.97	99.43	98.57	98.58	97.86	97.73	94.23	98.17
average	95.20	94.28	95.05	94.09	93.00	90.73	89.89	85.73	92.53

Figure 5-2. Results of average reduction (%) with collisions using a single set of filters

We analyze results of average reduction (%) with collisions in figure 5-2 as follows.

On average, 92.53% of all unneeded tuples are eliminated by the algorithm when the collision rate is in the range 1% to 60%. Therefore, on average, the algorithm gives substantial reductions in relation size.

For query types 3-2 and 4-2, the algorithm gives average percentages of reduction which are substantially lower than those of the remaining query types.

In figure 5-3, we show the results of full reduction with collisions using a single set of filters. The results are based on the experimental assumptions described in section 4.3 (Experiments with Collisions) and each data is the results of 100 queries. The first row represents the specific percentages of collisions. We also do the experiments on query types 3-2, 4-2, 5-2, and 6-2. The last column shows the average data of different query type. The last row represents the average data of different percentage of collisions. The data at right bottom corner is the average result of all data.

collision %	1	5	10	20	30	40	50	60	average
3 - 2	26	19	19	18	19	17	21	13	21
4 - 2	50	41	40	34	39	29	29	14	37
5 - 2	70	66	58	47	43	44	40	30	54
6 - 2	80	66	52	46	41	36	32	25	52
average	57	48	42	36	35	32	31	21	41

Figure 5-3. Results of average percentages of full reduction with collisions using a single set of filters

The results in figure 5-3 show that the percentage of full reduction declines substantially when the percentage of collisions increases. On average, 41% of all queries are fully reduced by the algorithm when the collision rate is from 1% to 60%. In figure 5-1, the average full reduction is 74% under the assumption of perfect hash function. The full reduction decreases more than 30%. Therefore, the amount of collisions substantially affects the number of queries achieving full reduction.

In figure 5-3, we notice that query types 3-2 and 4-2 have percentages of full reduction that are significantly lower than other query types.

5.3 Results with Collisions Using Two Sets of Filters

We have improved the algorithm by utilizing two sets of filters. The assumptions of the experiments are similar to the assumptions of the experiments with collisions. The assumptions are described in chapter 4 (Experiments).

In figure 5-4, we present the results of evaluation of the algorithm using two sets of filters under the specific percentages of collisions. The results show the average reduction (%) with collisions. Each data in the table is the average results of 100 queries. The first row represents the specific percentages of collisions. We also do the experiments on query types 3-2, 4-2, 5-2, and 6-2. The last column contains the average data of different query types. The last row represents the average data of the different percentage of collisions. The data at right bottom corner is the average result of all data.

collision %	1	5	10	20	30	40	50	60	average
3 - 2	89.08	89.17	85.53	89.14	88.63	88.91	87.53	89.39	88.29
4 - 2	94.79	94.57	93.46	96.88	95.03	95.06	93.12	94.95	94.79
5 - 2	98.24	97.66	97.76	98.09	97.30	97.41	97.39	98.52	97.85
6 - 2	99.23	99.61	99.37	99.30	99.41	99.02	99.01	99.46	99.25
average	95.38	95.25	94.03	95.85	95.09	95.11	94.26	95.58	95.05

Figure 5-4. Results of average percentage reduction (%) with collisions for the two sets of filters

The results in figure 5-4 show us the following facts:

- ~ The different percentages of collisions do not affect the average reduction of relations using two sets of filters described in the previous section. We obtain this conclusion by comparing the data in each row.
- ~ The performance of the algorithm using two sets of filters is better than the performance of the algorithm using a single set of filters under the specific percentage of collisions. The average of all data is improved from 92.53% to 95.05%. We obtain this conclusion by comparison the results in figure 5-4 to the results in figure 5-2. The results in figure 5-2 are the evaluation of the algorithm using a single set of filters
- ~ By using two sets of filters, for the query type 3-2, the average result of various percentage of collisions is improved from 83.50% to 88.29% and for the query type 4-2, the average result of various percentage of collisions is improved from 91.92% to 94.79%.
- ~ The average result of all data in figure 5-4 is 95.05%, only 2.36% lower than the results without collision in figure 5-1, the average reduction of different query types is 97.41% in that table. This means that the performance of our algorithm using two sets of filters is good under the assumption of collisions.

In figure 5-5, we show the results of full reduction for evaluation of the algorithm using two sets of filters under the specific percentage of collisions.

collision %	1	5	10	20	30	40	50	60	average
3 - 2	47	41	31	46	42	34	34	30	38
4 - 2	59	54	54	61	57	55	45	53	55
5 - 2	82	74	72	71	70	72	69	77	74
6 - 2	87	90	85	90	91	79	84	82	86
average	69	65	61	67	65	60	58	61	63

Figure 5-5. Results of full reduction (%) with collisions for the two sets of filters

The results in figure 5-5 tell us that

- ~ The different percentages of collisions do not affect the full reduction of relations much using two sets of filters. We obtain this conclusion by comparing the data in each row.
- ~ The performance of the algorithm using two sets filters is better than the performance of the algorithm using a single set of filters under the specific percentage of collisions. The average of all data is improved by 22%, from 41% to 63%. We obtain this conclusion by comparison the results in figure 5-5 to the results in figure 5-3. The results in figure 5-3 are the evaluation of the algorithm using a single set of filters
- ~ For the query type 3-2, the average result of various percentage of collisions is improved from 21% to 38%. For the query type 4-2, the average result of various percentage of collisions is improved from 37% to 55%.

Chapter 6 Conclusions

The optimization of general queries in a DDBMS is an important research area. The problem is the selection of the best sequence of database operations to process the query and keep costs to a minimum. As finding the optimal solution is NP-hard, the approach is to develop heuristic algorithms which can produce near-optimal solutions.

The main approaches in the literature have been joins [5, 31, 32, 33], semijoins [7, 8, 9, 10, 11, 35] and filters [16, 17, 18, 19, 20, 21, 24, 26]. In general, semijoin-based algorithms perform better than join-based algorithms. But the problem is that semijoins are still expensive. Filters have been proposed as a cheap way to implement semijoins. However, since Bloom filters are constructed using hash functions, collisions are a problem. Specifically, because of collisions some relations in a query may not be reduced to the full extent possible. This means that data transmission costs are higher than they need to be.

In this thesis we investigate the effect of collisions on a filter-based algorithm. Our hypothesis is that the use of two sets of filters, each with a different set of collisions, can reduce the effect of the collisions and improve the performance of the algorithm.

To investigate the effect of the collisions and test our hypothesis we conducted three sets of experiments:

1. We simulated a perfect hash function to evaluate the performance of the algorithm without collisions.
2. We simulated various percentages of collisions to see the effect of these collisions on the performance of the algorithm. For these experiments we used a single set of filters for each joining attribute.
3. We noted that certain types of queries were very badly affected by the collisions. So in our last set of experiments we simulated two separate hash functions, with possibly different sets of collisions, to investigate the effect of two sets of filters on the performance of our algorithm.

From our experiments we conclude the following:

1. Using a perfect hash function, the algorithm achieves substantial reduction in the relations.
 - ~ On average, approximately 97.41% of all tuples not required for the final answer of the query are eliminated from the relations.
 - ~ The average full reduction of all data is 74%.
 - ~ We get better reduction when the query requires more relations.
2. With a single set of filters, collisions effect the performance of the algorithm.
 - ~ The full reduction of the algorithm declines substantially when the percentage of collisions increases. On average, only 41% of all queries are fully reduced.
 - ~ The query types 3-2 and 4-2 have low percentages of full reduction, 21% and 37% respectively.
3. Using two sets of filters, the performance of the algorithm is much better than using a single set of filters under the assumption of collisions.
 - ~ The average reduction of all data is improved from 92.53% to 95.05%.
 - ~ The average reduction of all data is 95.05% when collisions occur. This is only 2.36% lower than the result under perfect hash function.
 - ~ The average full reduction of all data is improved by 22%, from 41% to 63%.
 - ~ The different percentages of collisions do not affect the reduction.

Bibliography

- [1] M.T. Ozsü and P. Valduries, "Principles of Distributed Database System", Printice Hall International, 1991.
- [2] C. Wang and M. S. Chen, "On the Complexity of Distributed query Optimization", IEEE Transactions on Knowledge and Data Engineering, 8(4), Pages 650-662, 1996.
- [3] Xiaobo Ma, "The Use of Bloom Filters to Minimize Response Time in Distributed Query Processing", Master's thesis, University of Windsor, 1997.
- [4] W. Perrizo and C. S. Chen. "Composite semijoins in distributed query processing", Information Science, 50: 197-218, 1990.
- [5] S. Bandyopadhyay, J. M. Morrissey, and A. Senpüta, "A Query Optimization Strategy for Distributed Databases on All-optical Networks", In Proceedings of the Canadian Conference on Electrical and Computer Engineering, pages 245-248, 1996.
- [6] J.M. Morrissey, W.T. Bealor, and S. Kamat, "A comparative evaluation of dynamic heuristics for cost minimization", In Proceedings of the 8th International Conference on Computing and Information (ICCI'96), 1996.
- [7] B.C. Ooi and B. Srinivasan, "On the identification of semijoin sequences for distributed query processing", In Processings of the IFIP WG 10.3 Working Conference on Distributed Processing, pages 551-561, 1987.
- [8] N. Roussopoulos and H. Kang, "A pipelined n-way join algorithm based on the 2-way semijoin program". IEEE Transactions on Knowledge and Data Engineering, 3(4), pages 486-495, 1991
- [9] M.S. Chen and P.S. Yu, "Combining join and semijoin operations for distributed query processing", IEEE Transactions on Knowledge and Data Engineering, Pages 534-542, 1993
- [10] P.S.M. Tsai and A.L.P. Chen, "Optimizing entity join queries by extended semijoins in a wide area multidatabase environment", In Proceedings 1994 International Conference on Parallel and Distributed Systems, Pages 676-681, 1994.
- [11] X. Lin, M.E. Orlowska, and X. Zhou, "Using parallel semi-join reduction to minimize distributed query response time", In Proceedings of the First International

Conference on Algorithms and Architectures for Parallel Processing, pages 517-526, 1995.

[12] A.Y. Lu and P.C.Y. Sheu. "Processing of multiple queries in distributed databases". In Proceedings of the Seventh International Conference on Data Engineering, Pages 42-49, 1991.

[13] M.S. Chen and P.S. Yu. "Combining Join and Semi-join Operations for Distributed Query Processing". IEEE Transactions on Knowledge and Data Engineering. vol. 5, no. 3 June 1993.

[14] P. Valduriez and G. Gardarin. "Join and semijoin algorithms for a multiprocessor database machine". ACM Transactions on Database Systems, vol9, no. 1, March 1984, Pages 133-161.

[15] T.S. Chen, A.L.P. Chen, and W.P. Yang. "Hash- semijoin: A new technique for minimizing distributed query time". In Proceedings of the 3rd Workshop on Future Trends of Distributed Computing Systems, pages 325-330, 1992.

[16] S. Kamat. "Dynamic strategy and Bloom filters in distributed query optimization". Master's thesis, University of Windsor, 1996.

[17] Z. Li and K.A. Ross. "Perf join: an alternative to two-way semijoin and bloomjoin". In Proceedings of CIKM'95, pages 137-144, 1995.

[18] J.M. Morrissey and X. Ma. "Investigating response time minimization in distributed query optimization", Presented at ICCI'98, 1998.

[19] J.M. Morrissey and W.K. Osborn. "Experiments with the use of reduction filters in distributed query optimization". In Processings of the Ninthe IASTED International Conference on Parallel and Distributed Computing and Systems, 1997.

[20] J.M. Morrissey. "Reduction filters for minimizing data transfers in distributed query optimization". In Proceedings of the 1996 Canadian Conference on Electrical and Computer Engineering, 1996.

[21] J.K.Mullin. "A second look at Bloom filters". Communications of the ACM, vol. 26, no.8, 1983

[22] J. K. Mullin. "Optimal semijoins for distributed database systems". IEEE Transactions on Software Engineering, vol. 16, no. 5, pages 558-560, 1990.

- [23] J.K. Mullin. "Estimating the size of a relational join". *Information System*, vol. 18, no. 3, pages 189-196, 1993
- [24] W.K. Osborn. "Distributed query optimization using bloom filters". Report, University of Windsor, 1996.
- [25] J.C.R. Tseng and A.L.P. Chen. "Improving distributed query processing by hash-semijoins". *Journal of Information Science and Engineering*, vol. 8, pages 525-540, 1992.
- [26] C.Y. Wang, W.P. Yang, J.C.R. Tsing and M. Hsu. "Random filter and its analysis". In *Proceedings of the 23rd Asilomar Conference on Signals, Systems and Computers*, pages 1031-1035, 1989.
- [27] W. K. Osborn. "The use of reduction filters in distributed query optimization". Master thesis, University of Windsor, 1998.
- [28] Jarke, M and Koch, J, "Query Optimization in database system", *ACM Compu. Survey*, vol.16, no. 2, June 1984, pages 111-152.
- [29] Harris, EP and Ramamohanarao, K, "Join Algorithm Cost Revisited", *The VLDB Journal*, vol. 5, 1994, pages 64-84.
- [30] Lu, AY and Sheu, PC-Y, "Processing multiple queries in distributed databases", *Proc. of IEEE*, 1991, Pages 42-49.
- [31] Willia Perrizo, Prabhu Ram and David Wenberg, "Distributed Join Processing Performance Evaluation", *IEEE 1994, International Conference on System Sciences*.
- [32] G.M. Lohman, C.Mohan, L.M. Haas, D. Daniels, B.G. Lindsay, P.G. Selinger, and P.F. Wilms. "Query processing in R*". In *Query Processing in Database Systems*. Springer, New York, 1985.
- [33] J.K. Ahn and S.C. Moon. "Optimizing joins between two fragmented relations on a broadcast local network". *Information Systems*, vol. 16, no. 2, pages 185-198,1991.
- [34] J.S.J. Chen and V.O.K. Li. "Optimizing joins in fragmented database systems on a broadcast local network". *IEEE Transactions on Software Engineering*, vol. 15, no. 1, pages 26-38, 1989.
- [35] Apers, PMG, Hevner, A, and Yao, SB, "Optimization algorithm for distributed queries", *IEEE Trans. Software Eng.*, Vol.9, No.1, Jan. 1983, Pages 57-68.

- [36] Bernstein, PA, Goodman, N, Wong, E, Reeve, C, and Rothnie, J, "Query processing in a system for distributed database (SDD-1)", *ACM Trans. Syst.*, Vol.6, Dec. 1981, pages 602-625.
- [37] Ceri, S, Gottlob, G, Tanca, L, and Wiederhold, G, "Magic semijoins", *Information Processing Letters*, Vol. 33, North-Holland, 1989.
- [38] Chen, M-S and Yu, PS, "Interleaving a join sequence with semijoins in distributed query processing", *IEEE Trans. Parallel and Distributed Systems*, 1992, pages 611-621.
- [39] Hevner, AR and Yao, SB, "Query processing in distributed database system", *IEEE Trans. Software Eng.*, vol. SE-5, no. 5, May 1979, pages 177-187.
- [40] Pramanik, S and Vineyard, D, "Optimizing join queries in distributed databases", *IEEE Trans. Software Eng.*, vol.14, no.9, Sept.1988, pages1319-1326.
- [41] Yu, CT and Chang, CC, "On the design of a of a query processing strategy in a distributed database environment", *Proc. ACM SIGMOD Intl. Conf. Management of Data*, 1983, pages 30-39.
- [42] W. T. Bealor. "Semijoin strategies for total cost minimization in distributed query processing". Master's thesis. University of Windsor, 1995.
- [43] J.M. Morrissey, S. Bandyopadhyay, and W.T. Bealor. "A comparison of static and dynamic strategies for query optimization". In proceedings of the 7th IASTED/ISM International Conference on Parallel and Distributed Computing Systems, 1995.
- [44] J.M. Morrissey, S. Bandyopadhyay, and W.T. Bealor. "A heuristic for minimizing total cost in distributed query processing". In proceedings of the 7th International Conference on Computing and Information - ICCI'95, 1995.
- [45] J.M. Morrissey and W.T. Bealor. "Minimizing data transfers in distributed query optimization: A comparative study and evaluation", *Computer Journal*, vol39, no. 8, 1997.
- [46] H. Kang and N. Roussopoulos. "Using 2-way semijoins in distributed query processing". In Proceedings of the 3rd International Conference on Data Engineering, pages 644-651, 1987.
- [47] C. Wang, V.O.K. Li and A.L.P Chen. "Distributed query optimization by one-shot fixed-precision semijoin execution". In Proceedings of the 7th International Conference on Data Engineering, pages 756-763, 1991.

- [48] C. Wang, V.O.K. Li and A.L.P. Chen. "One-shot semi-join execution strategies for processing distributed join query". *Computer Systems Science and Engineering*, vol.8, no. 4, pages 245-253, 1993.
- [49] J.S.J. Chen and V.O.K. Li. "Domain-specific semijoin: a new operation for distributed query processing". *Information Sciences*, 52, 1990.
- [50] F. Barlos and O. Frieder. "On the development of a site selection optimizer for distributed and parallel database systems". In *CIKM'93. Proceedings of the Second International Conference on Information and Knowledge Management*, pages 423-432, 1993.
- [51] B.H. Bloom, "Space/time tradeoffs in hashing coding with allowable errors", *Commun. ACM*, vol.13, no.7, July 1970, pages 422-426.

Vita Auctoris

NAME	Yan Liang
PLACE OF BIRTH	Chengdu, China
YEAR OF BIRTH	1961
EDUCATION	M. Sc., Computer Science, University of Windsor, Windsor, Ontario, Canada 1997 ~ 1999 B.Sc., Mechanical Engineering, Beijing Institute of Technology, Beijing, China 1978 ~ 1982