2003

A bloom-filter strategy for response time reduction in distributed query processing.

Wanxin Gao
University of Windsor

A Bloom-filter Strategy for Response Time Reduction in Distributed Query Processing

by

Wanxin Gao

A Thesis
Submitted to the Faculty 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
Univeristy of Windsor

Windsor, Ontario, Canada
2003

Your file Votre référence
ISBN: 0-494-00123-2
Our file Notre référence
ISBN: 0-494-00123-2

ABSTRACT

In distributed database systems, query optimization is to find strategies attempt to minimize the amount of data transmitted over the network. Optimization algorithms have an important impact on the performance of distributed query processing. Since optimal query processing in distributed database systems has been shown to be NP-Hard [WC96], heuristics are applied to find a cost-effective and efficient (but suboptimal) processing strategy.

Many query optimization strategies have been proposed to minimize either the total cost or the response time. The approaches in distributed query processing have mainly focused on the use of joins, semijoins, and filters. In this thesis, we propose a new reduction strategy based on bloom-filters to significantly reduce the response time of a distributed query. This algorithm can process general queries consisting of an arbitrary number of relations and join attributes. The performance of the algorithm with respect to response time is compared against the Initial Feasible Solution (IFS). An amount of experimental results has been used to evaluate the performance of our algorithm. Compared to the IFS, our algorithm provides a significantly improved query solution.

To my grandmother, Yu Liu

my mother, Xiaoqing Tong

my father, Jing Gao

my husband, Hongtao Zhang

ACKNOWLEDGMENTS

I would like to express my sincere gratitude and appreciation to Dr. Joan Morrissey, who has been a constant source of inspiration and encouragement. Her support, guidance was invaluable in the successful completion of this thesis. I am thankful to Dr. Xiao Jun Chen for her insightful comments and constant support. I would like to thank Dr. Karen Y. Fung for her comments and suggestions. I also would like to thank Dr. Peter Tsin and my other committee members for being accommodating when I needed it most. I would like to thank my friends Amber and Nelson for all the help and support during the completion of this thesis. Also thanks must go to my husband – Hongtao whose help and company was a constant source of strength, happiness and encouragement throughout the course of this work. Last, but not least, I would like to thank my parents whose love always keep my hope and confidence.

CONTENTS

LIST OF FIGURES

Chapter 1 INTRODUCTION

A Distributed Database System (DDBS) is a kind of decentralized information system. It allows data to be distributed and managed over the network. Distributed Database Systems have certain advantages over traditional Centralized Database Systems in that they achieve the advantages of performance, reliability, availability and modularity that are inherent in distributed systems [Teo92]. However, optimizing queries in DDBS is difficult. In this thesis, a new reduction algorithm is proposed to significantly reduce the response time of queries.

1.1 Distributed Database Systems

1.1.1 What is a Distributed Database System?

A Distributed Database (DDB) is a collection of multiple, logically interelated databases which are dispersed geographically over a computer network and maintained by local computers. What we are interested in is an environment where data is distributed among a number of sites (Figure 1.1).

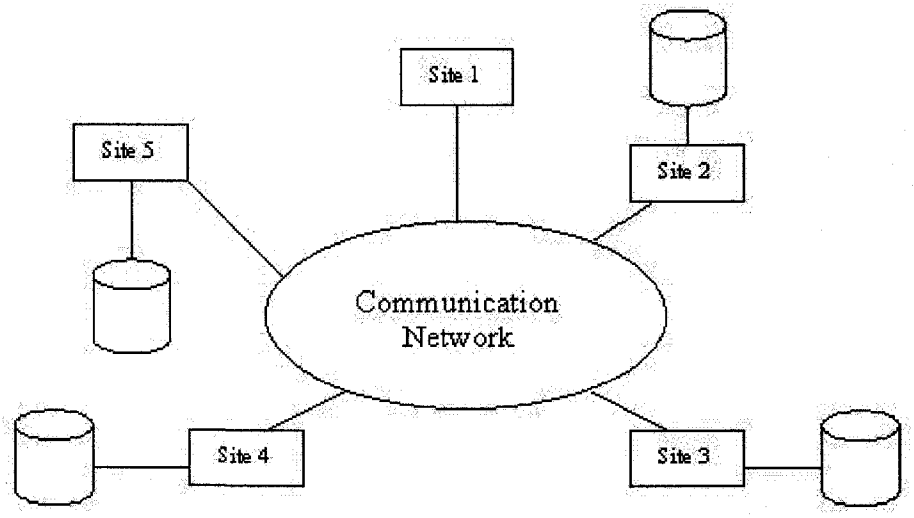

Figure 1.1 DDBS Environment

A Distributed Database Management System (DDBMS) is then defined as the software system that permits the management of the DDBS and makes the distribution transparent to the users. A Distributed Database System (DDBS) refers to the combination of the DDB and DDBMS. To form a DDBS, files should not only be logically related, but there should be structure among the files, and access should be via a common interface.

1.1.2 Potential advantages and problems of DDBS

As presented in Figure 1.1, in a distributed database system, data is distributed among a number of sites. Thus, each of the relations can be fragmented and each partition can be stored at a different site. This is known as fragmentation. Furthermore, it may be preferable to duplicate some of this data at other sites for performance and reliability reasons. This is known as replication. However, some inherent problems give additional consideration complexity and maintenance costs. Distribution causes some problems of control and security. Depend on these characters of the DDBS, there are several advantages and problems as follows [ŐV99]:

Advantages:

i. Since a distributed DBMS fragments the conceptual database, enabling data to be stored in close proximity to its points of use,

- each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for centralized databases.
- localization reduces remote access delays that are usually involved in wide area networks.

ii. With replication, site failure does not mean transaction failure.

iii. The inherent parallelism of distributed systems may be exploited for inter-query and intra-query parallelism.

iv. In a distributed environment, it is much easier to accommodate increasing database sizes.

Problems:

i. Complexity. DDBS problems are inherently more complex than centralized database management ones.

ii. Cost. Distributed systems require additional hardware and some additional more complex software and communication that may be necessary to solve some of the technical problems.

iii. Distribution of Control. Distribution creates problems of synchronization and coordination.

iv. Security. In a distributed database system, there are serious problems in maintaining adequate security over computer network.

v. NP-Hard. For general queries with arbitrary complexity to generate an optimal processing strategy in distributed database systems is called NP-Hard. Therefore, most proposed algorithms for processing distributed queries are heuristic, and focus on producing efficient but suboptimal strategies that minimize some practical cost of the query.

1.1.3 Distributed query processing

Basically, a distributed query processing algorithm is defined to be a set of relational operations and network transmission steps. To process distributed queries, most algorithms process as follows.

i. Initial local processing: all local processing that requires no intersite communication such as selection, projection and local join operations are performed.

ii. Reduction: after the preprocessing by the first step, a sequence of reducers (semijoins and joins) is used to reduce the size of the relations in a cost-effective way and for reducing the total communication cost required.

iii. Final processing: all resulting relations (possibly reduced) are sent to the final (or assembly) site where the final query processing is performed to produce the answer for a given query.

1.2 Query optimization approaches overview

The area of query optimization is very large within the database field. It has been studied in a great variety of contexts and formed many different angles, giving rise to several diverse solutions in each case. Among the algorithms that have been proposed for query optimization, especially distributed query optimization, the research in this area falls into the following categories: Using join based strategies [LMHD$^+$85][BMS96] [CY90][LPP91][CL89][Seg84]; Semijoins based strategies [BGWR$^+$81][AHY83][KR87] [RK91] [MB96][YL89][PC90][CL90][WLC91][WCS92][TC94]; filters based strategies [TC92][CCY92][LR95][MBB95][Ma97][MM98][Kam96][MO98][Osb98][MLO00] [Lia99].

1.3 The thesis statement and topics to be investigated

We will propose a new optimization algorithm for distributed databases – algorithm H, which uses bloom-filters to accomplish the same reduction effects as semijoins, but at a lower cost. The primary goal of our algorithm is to reduce the response time of a distributed query. The secondary goal is to reduce relation sizes while using data transmission as little as possible. This algorithm can process general queries consisting of an arbitrary number of relations and join attributes.

The performance of the algorithm with respect to response time is compared against the Initial Feasible Solution (IFS) that ships all relations directly to the query site and perform joins there. Our algorithm is evaluated to determine how much better it comes to reducing the response time when comparing with IFS.

In this thesis, the following questions will be examined:

- Can improvement be made in bloom-filter based query optimization heuristics?
- When is it profitable to apply this algorithm for a distributed query?
- On average, how much response time is reduced with respect to IFS?

1.4 Definitions and notations

Simple queries: A query that after initial local processing, each relation in the query contains only one common join attribute.

General queries: A query consists of an arbitrary number of relations, each relation has an arbitrary number of join attributes.

Tree queries: A query whose graph is tree or equivalent to a query with tree graph.

Cyclic queries: A query that has cycled query graph.

Acyclic tree query: A tree query that has no cycled query graph.

Heuristic: An algorithm that attempts to generate efficient, but suboptimal query execution strategies by minimizing some cost function.

Schedule: The data transmissions used for reducing a relation and the transmission of the reduced relation to the query computer form a schedule for this relation.

Query graph: Let G = (V, E) be a query graph. $G_B = (V_B, E_B)$ is a connected subgraph of G. Let $R_1, R_2, \ldots, R_n$ be relations corresponding to nodes in V_B and let A, B, ... be the attributes associated with edges in E_B.

Degree: In query graph, the number of edges that incident with relation R_i is the degree of R_i.

1.5 Organization of the thesis

The rest of this thesis is organized as follows. Chapter 2 reviews related background in the area of query optimization. The properties of joins, semijoins, and filters are discussed. Several query processing algorithm are presented. Some assumption about our proposed algorithm is presented in chapter 3. In chapter 4 there is a detailed introduction of our proposed algorithm. An illustrated example of how the algorithm works is also presented. The evaluation framework and experiment result are described in chapter 5. Lastly, in chapter 6, we provide a summary of the conclusions attained from the work that this thesis represents along with some plans for future work.

Chapter 2 BACKGROUND REVIEW

In distributed query processing, many approaches use relational operators such as joins [LMHD+85][BMS96][CY90][LPP91][CL89][Seg84], semijoins [BGWR+81][AHY83] [KR87][RK91][MB96][YL89][PC90][CL90][WLC91][WCS92][TC94], and filters [TC92][CCY92][LR95][MBB95][Ma99][MM98][Kam96][MO98][Osb98][MLO00] [Lia99]. In this chapter, some representative approaches related with this thesis will be presented.

2.1 Cost models

In distributed query processing, the objective is to minimize one of two cost functions: the total cost and the response time. They are two most popular cost models. The ***total cost model*** includes all the costs involved in the transmission of all data. The ***response time model*** calculates the elapsed time between the start of the query and the final results are obtained. As the speed of the network over which the data are transmitted is relatively low, the data transmission cost is the most important factor. Most heuristics assume that the cost involved in transmission data from one site to another is linear and local processing cost is considered to be negligible in most cases.

2.2 JOIN operation and approaches based on joins

2.2.1 JOIN operation

The join operation is one of the fundamental relational database query operations that allows data stored at different sites to be combined together based on some common information in query processing [ME92][YC84].

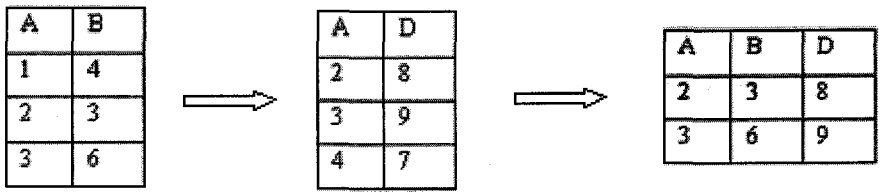

Figure 2.1 Join operation

In figure 2.1, given two relations R_1 and R_2, both relations contain the common join attribute A. Join R_1 with R_2 is performed by concatenating tuples of R_1 and R_2 where the value of attribute A is equal for both relations. In distributed systems, if relations R_1 and R_2 are not in the same location, we have to ship data in one location to another. The sequence of operations could be used to optimise queries. Although the join has the advantage of simplicity, it suffers from some problems. One is that the result relation could be much larger than the relations participating in the join. This increases the data transmission cost. Another is that 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.

2.2.2 Approaches based on joins

In this part, we illustrate two representative algorithms which use joins: R* optimizer [LMHD+85] and two-way join algorithm [CL89][Seg84].

R* optimizer:

In 1979, R* optimizer [LMHD+85] is an experimental adaptation to a distributed environment of System R [ABCE+76] and it is developed at IBM San Jose Research Laboratory. The objective of the R* optimizer is to minimize the total cost of a

distributed join query. The total cost includes local processing, I/O, and communication cost. It is executed at the site where the query is initiated. R* optimizer generates a processing strategy for a query as follows:

1) For each relation in the query, find all possible access paths to access the relation, and choose the one that has the minimum cost as the access path for this relation.
2) For each order of relations in the query, build a strategy to do the joins and calculate the cost of the strategy. Finally, choose the order with the minimum cost.
3) For each site involved in the query, choose a local processing strategy to carry out the local processing.

The purpose of query optimization in R* is to decide on five major aspects of execution: the access method, the join method, the join sites, inner-table transfer strategy, and ordered result delivery to query site. Actually this algorithm significantly reduces the number of alternatives by using dynamic programming and heuristics, but it has a combinatorial complexity in the number of relations involved.

Legato *et al* [LPP91] presents an algorithm for determining a better execution sequence to minimize data transmission costs. This algorithm is to formulate the optimization problem in terms of the construction of a tree structured solution (AQT), and a dominance property has been used to reduce the search space to be explored. Legota's algorithm improved the R^* optimizer, by removing a restriction, which is at least one of the two operands of each binary join is a base relation. As with the R* optimizer, the complexity of this algorithm is exponential.

Two-way join:

The two-way join algorithm is used to solve the problem of single join between two fragmented relations [CL89][Seg84]. Two-way joins are the most commonly encountered operations in relational queries. The optimization of two-way joins serves as a basis for the optimization of multiple joins.

Two general strategies exist for processing the two-way join of relation [AM91]: either union the fragments of each relation, then perform a join between the two relations or join all fragments of the first relation with all fragment of the second relation, then union the results.

Using the second strategy, a four-phase approach has been proposed in [CL89] to optimize two-way joins on fragmented relations in a broadcast local computer network.

Phase one: *Join Graph construction.*

Constructing a join graph for a given join. The nodes in the join graph represent the fragments of joining relations. The edges in the join graph represent the nonempty join of fragments. The qualification of fragment, which refers to the formulation of the properties common to all tuples in a fragment, is used to determine and remove empty fragment joins for the join graph.

Phase two: *Join-Analysis Graph construction.*

Transforming the join graph into a join-analysis graph. The fragment joins that are locally processable are identified on order to minimize the total amount of intersite data transfer.

Phase three: *Determining a Minimum-Weight Vertex Cover.*

Finding Minimum-Weight Vertex Cover (MWVC) for the corresponding join-analysis graph in order to determine which fragment join execution plan has minimum data transmission cost.

Phase four: *Final processing optimization.*

The final optimisation transmits fragment joins to the query site and units item to finish the join.

This algorithm only focuses on minimizing the data transmission cost of shipping fragments and fragment join results when processing a two way join. There have several other optimization approaches for two-way join presented in [AM91][Seg84]. At here, they will not be introduced one by one.

2.3 SEMIJOIN operation and approaches based on semijoins

2.3.1 SEMIJOIN operation

To process a query, the join operator suffers some problems such as the result relation could be much larger than the relations participating in the join, or transferring a large amount of data between different sites over the network, especially if the result of join relation only contains a few tuples which is much smaller than the size of original relation. To alleviate these problems the semijoin operator [BC81][BGWR$^+$81] is introduced as an effective operator to reduce the cost of an expensive join. The semijoin operation is guaranteed to monotonically reduce the size of a relation, with the worst case being no reduction. In addition, the properties of semijoins permit their computation with less intersite data transfers than for joins.

The semijoin is a relational algebraic operation that selects a set of tuples in one relation that match one or more tuples of another relation on the joining domains. Semijoins have been used as a basic ingredient in query processing strategies for a number of hardware and softeware database systems. The semijoin $R_i \rtimes_A R_j$ is computed by the following steps (Figure 2.2):

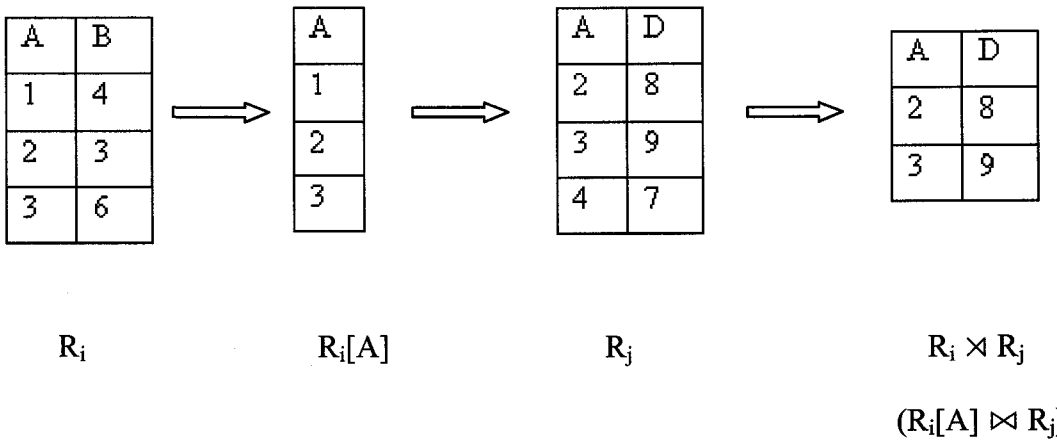

Figure 2.2 Semijoin operation

1) Send the projection $R_i[A]$ from site i to j.

2) Reduce R_j by eliminating tuples whose attribute A are not matching any value in $R_i[A]$.

So semijoin $R_i \rtimes R_j$ equals to join $R_i[A] \bowtie R_j$.

The purpose of the semijoin $R_i \rtimes R_j$ is to reduce the relation R_j before any joining takes place by removing tuples which will not be part of the final result. The semijoin is cost-effective when the benefit is larger than the cost. The semijoin cost is the size of projection $R_i[A]$. The benefit is the amount of size reduction on relation R_j.

The advantages of the semijoins are that the data transmission cost is reduced since only an attribute projection needs to be shipped and never producing a larger relation than those participating in the semijoin. The disadvantage is that it incurs higher local processing costs since a projection must be executed each time.

2.3.2 Approaches based on semijoins

In this part, we illustrate three representative algorithms that use semijoins: SDD-1, AHY, and two-way semijoin.

SDD-1:

The SDD-1 [BGWR+81] (System for Distributed Database) is a distributed database system developed by the Computer Corporation of America. It is the first system to allow a relational database to be distributed on a computer network. Users interact with SDD-1 by submitting queries coded in a high level procedure language called Datalanguage. The SDD-1 algorithm proceeds in four phases as follows:

Initialization. It generates a set of beneficial semijoins - BS, and an execution strategy - ES that includes only local processing.

Selection of beneficial semijoins. The phase selects the beneficial semijoins from BS by iteratively choosing the most beneficial semijoin - SJ_i, and modifying the database statistics (such as the cardinality, size, and selectivity of relations) and BS accordingly. The modification affects the statistics of relation R involved in SJ_i and the remaining semijoins in BS that use relation R. The iterative phase terminates when all semijoins in BS have been considered and appended to the execution strategy - ES.

Assembly site selection. It selects the assembly site by evaluating, for each candidate site, the cost of transferring to it all the required data and taking the one with the least cost.

Postoptimization. It permits the removal from the execution strategy (ES) of those semijoins that affect only relations stored at the assembly site. This phase is necessary because the assembly site is chosen after all the semijoins have been ordered. The SDD-1 optimizer is based on the assumption that relations can be transmitted to another site. This is true for all relations except those stored at the assembly site, which is selected after beneficial semijoins are considered. Therefore, some semijoins may incorrectly be considered beneficial. It is the role of postoptimization to remove them form the execution strategy.

Although SDD-1 algorithm is an improved optimization algorithm, it still has several drawbacks. The SDD-1 algorithm selects local optimal strategies, which means only selecting semijoins that maximize immediate gain. Therefore, it ignores the higher-cost semijoins that would result in increasing the benefits and decreasing the costs of other semijoins at each step of the strategy generation. Thus this algorithm may not be able to select the global minimum cost solution.

AHY:

Apers *et al* [AHY83] present a new algorithm - algorithm AHY (GENERAL) that use semijoin operation to derive processing optimization strategies for arbitrarily complex queries. For a special class of simple queries (see 1.4), Hevner and Yao developed algorithms PARALLEL and SERIAL [HY79] that find strategies with, respectively, minimum response time and total time for simple queries. There are three versions of the

algorithm AHY: one for minimizing response time (RESPONSE) and two for minimizing total time (TOTAL, COLLECTIVE)

Algorithm AHY:

1. Do all initial local processing.

Local processing includes the computation of restrictions, projections, and semi-joins between relations that reside in the same node.

2. Generate candidate relation schedules (see 1.4).

For all three versions, a general query (see 1.4) is decomposed into simple queries by isolating each join attribute. These simple queries are then processed by either algorithm PARALLEL or SERIAL depend on the type of optimisation to generate candidate schedules.

3. Integrate the candidate schedules into a near optimal execution strategy.

After step 2, all candidate schedules are saved. In this step, those candidate schedules are integrated to form a processing schedule for the response time or total tome of each relation R_i according to each strategy of three versions.

4. Remove schedule redundancies.

Some schedules for relations have been transmitted in the schedule of another relation. So the last step is to eliminate these relation schedules.

Algorithm AHY (GENERAL) to be an efficient algorithm of polynomial complexity that derives close to optimal query processing strategies on distributed systems. But the lack of consideration of global conditions and many simplifying assumptions concerning the network may result in suboptimal strategies being generated.

Two - way semijoin:

Traditional semijoin only does the forward size reduction of the joining relations. So, it can be improved for more cost-effective.

Kang and Roussopoulous [KR87][RK91] describe a new relational operator --- 2-way semijoin that enhances the semijoin with backward size reduction capability for more cost-effective query processing. A 2-way semijoin operator not only performs forward reduction as the traditional semijoin operator does, but also performs backward reduction always cost-effective.

A 2-way semijoin $R_i \leftarrow A \rightarrow R_j$ is computed with the following steps (Figure 2.3):

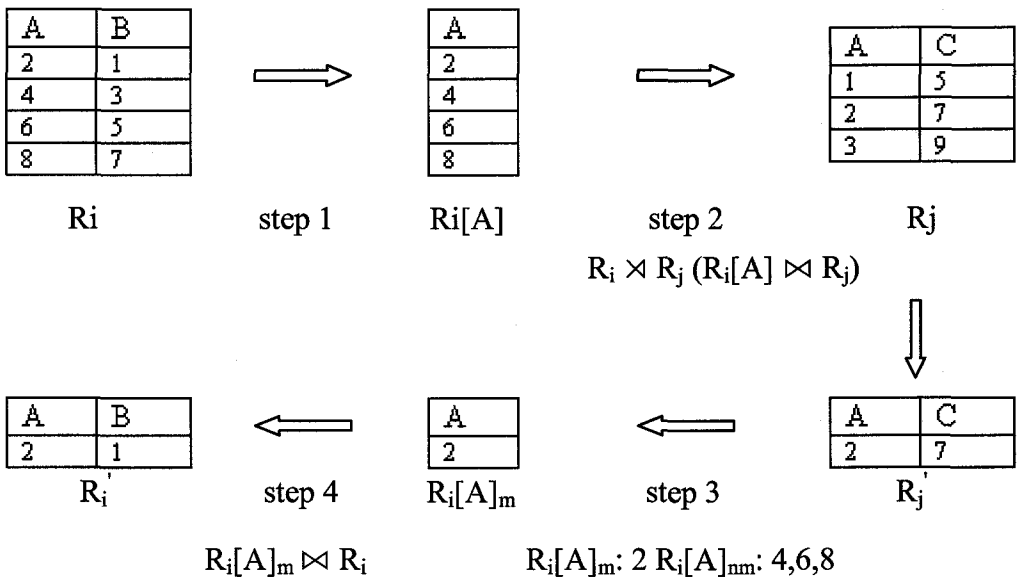

Figure 2.3 2 – way semijoin operation

1. Send $R_i[A]$ from site i to j.

2. Executing $R_i[A] \bowtie R_j$. During the forward reduction of R_j, partition $R_i[A]$ into $R_i[A]_m$ and $R_i[A]_{nm}$ where $R_i[A]_m$ contains the values of attribute A which match one of values in $R_j[A]$, and $R_i[A]_{nm}$ contains the values of $(R_i[A]-R_i[A]_m)$.

3. Send the smaller size one between $R_i[A]_m$ and $R_i[A]_{nm}$ from site j back to i.

4. Perform backward reduction. If $R_i[A]_m$ is used, execute $R_i[A]_m \bowtie R_i$. If $R_i[A]_{nm}$ is used, then tuples whose attribute A are matching values in are eliminated. The figure 2.3 shows an example of 2-way semijoin.

The result of a 2-way semijoin t: $R_i \leftarrow A \rightarrow R_j$ is equal to the result of two separate semijoins s: $R_i \rtimes_A R_j$ and s^r: $R_j \rtimes_A R_i$. Thus, the benefit of t is the sum of the benefits of s and s^r. However, the cost of t is always smaller than the sum of the costs of s and s^r because in step 3 we send the one of $R_i[A]_m$ and $R_i[A]_{nm}$ which is less size from I back to j. Therefore, the extension of the semijoin to the 2-way semijoin is done in a cost-effective way. No matter the cost and the benefit of the forward reduction is, the backward reduction is always performed cost-effective. Besides these, the 2-way semijoin has more powerful propagation effects that have been proved in [KR87].

In [KR87], the authors mention some existing heuristic algorithms based on semijoins can be easily modified by replacing semijoins with 2-way semijoins or combining semijoins and 2-way semijoins together.

The authors also proposed an algorithm [RK91] that attempts to minimize the local processing cost of a query. This algorithm uses the 2-way semijoin framework and pipeline techniques to eliminate the process of creating, storing, and transmitting intermediate results on the local disks of the query site. This gives good I/O savings.

2.4 FILTER and approaches based on filters

2.4.1 FILTER

The difficulty of the optimization of general queries in a distributed database management system is to select the database operations that will process the query and minimize costs. Traditional solutions include the use of heuristic strategies based on semijoin or join operations. Compared with it, constructing a filter is generally smaller than the join attribute projection due to its small size. It is cheaper to transfer a filter over the network than a relation. However because hasing is utilized, a filter suffers from the problem of collisions. Collisions occur as a result of two or more attribute values hashing to the same address in the array. But by choosing the size and the hashing transformations carefully, it is possible to make collisions insignificantly small.

A filter, also called bit vector filter, is an array used to encode the information about the values or other properties contained in an attribute. Bloom filter [Bab79][Mul83] is an important filter that was invented by Vurton Bloom in 1970 [Blo70]. A bloom filter is an array of bit generated by using a hash function on a join attribute. Each bit in the bloom filter functions as compact representation of the join attribute values. The storage structure of a bloom filter consists of an array of bits. Figure 2.4 shows how a filter works.

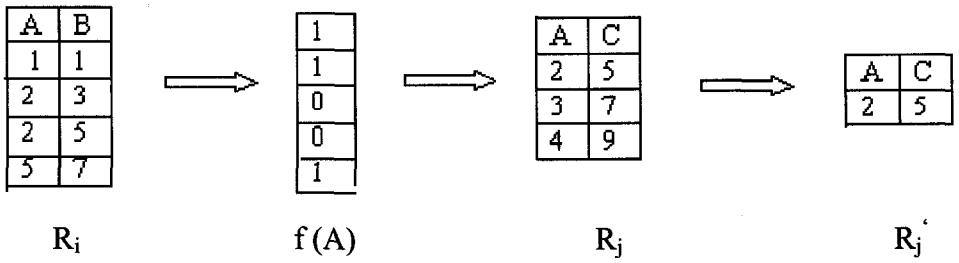

Figure 2.4 Bloom filter operation

1) Create a filter f (A) at site i.

- Create an array to hold bits.
- Let all bits in the array to zero.
- Developed a hash function and use it to produce an address in the array for each joining attribute value.
- For each address produced, set the corresponding bit it 1.

2) Reduce relation size

- Send the filter f (A) to site j.
- Each value of join attribute of R_j, hashes on the value to produce an address.
- For each above address in the filter, evaluate its value.

  If the value is 1, the corresponding tuple is kept for further processing.

  If the value is 0, discard the tuple.

2.4.2 Approaches based on filters

In this part, we illustrate three representative algorithms that use filters: Hash-semijoin, algorithm X, and PERF join.

Hash-semijoin

Tseng and Chen [TC92] propose a new relational operator called a hash-semijoin that is designed based on the concept of search filter to greatly reduce the cost of a semijoin operation by sacrificing some benefits. The hash-semijoin of R_i and R_j is denoted by $R_j \propto R_i$. To reduce the cost of a semijoin operation, they use a search filter that represents the semijoin projection with a small bit array. The approach works as follows:

1. Initialize all bits in the array F by setting to 0.

2. For each value of the join attribute in R_i, use the d hash functions to hash the attribute value into d bit addresses. Then set d bits in the bit array F to 1.

3. Transmit the bit array to the site of R_j.

4. For each tuple of R_j, use the same d hash functions to hash the join attribute value to d bit addresses. If all these d bits are 1 in the bit array, then the tuple in R_j will be selected as a semijoin result.

Compare with traditional semijoin, the hash-semijoin is more cost-effective. However, it can only be used for the tree query (see 1.4). And using search filter to reduce the tuples of the relation may not be a real semijoin result because of the information loss in representing a value with d bits. The case where a value is falsely accepted by the search filter is called a false drop. So after hash-semijoins ($R_j \propto R_i$), the reduced R_j (R_j') is a subset of R_j, but a superset of $R_j \ltimes R_i$ since false drops may occur. That means "hash-semijoins have to sacrifice some benefits". Another drawback is hard to choose the number of hash functions, d. With a smaller d, the cost and the benefit of the hash-semijoin are smaller. On the contrary, with a larger d, the cost and the benefit are larger too. So it is not easy to derive an optimal d that result in an improvement in the potential cost of a semijoin program.

Algorithm X:

Morrissey and Ma [Ma97][MM98] propose a heuristic for processing general queries. Algorithm X uses bloom filters to reduce query response time as well as local processing costs.

The basic idea is to apply all filters to all relations, concurrently. The rationale is that the filter sizes are relatively small and therefore the cost is relatively low. Each relation is at most processed two times, once to construct the filters (which can be done during initial local processing) and once when the reducing filters are applied. Furthermore, the application of all filters concurrently will not increase the response time – and it will certainly decrease the local processing cost.

The detailed description of Algorithm X 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
```

Compared with AHY (response time) [AHY83], algorithm X is more efficient. However, algorithm X is based on the perfect hash function.

PERF join:

Li and Ross [LR95] propose a new search filter called "Positionally Encoded Record Filters"(PERF) and describe their use in a distributed query processing technique called PERF join. A PERF is a novel two-way join reduction implementation primitive which having the same storage and transmission efficiency as a hash filter (e.g., bloom filter), a PERF is based on the tuple scan order instead of hashing. So it doesn't suffer any loss of join information incurred by hash collisions.

The basic idea of PERF join is to minimize the cost of the "backward" reduction. For relation R_i, a bit vector contains one bit of every tuple. When performing forward reduction of semijoin, the PERF for relation R_i encodes with tuples that will be part of the join result. The bit is set to 1 if it is in the projection R_j. Then ship it back to the site of R_i and applied to R_i to filter out any unwanted tuples.

PERF join can reduce the cost of "backward" phase much better than two-way joins and two-way semijons.

Chapter 3 ASSUMPTION

The algorithm we will propose in this thesis is based on the following assumptions.

1) We assume a Distributed Database Management System that contains relational data that is no fragmentation or replication, a point – to – point network.
2) Only Select – Project – Join (SPJ) query is considered. There is no set operations like UNION, INTERSECTION, PRODUCT, DIFFERENCE involved in the research.
3) A query consists of a number of relations, each at different site, and the result made available at the query site. Each relation can have a number of join attributes.
4) We assume the cost model is

   $C(X) = C_0 + X$ Where $C_0 = 0$ for simplicity.

   X is the amount of data transmitted. Here we use unit "word" to represent transmitted data.
5) We assume that we have a perfect hash function so the filter size (number of address) is the same as the domain size. Each attribute value is a 64-bit word.

Chapter 4 ALGORITHM H

In this chapter, we present our algorithm. The idea of this algorithm is that we use bloom-filters to accomplish the same reduction effects as a semijoin strategy, but at a lower cost. We use the parallel concept to achieve the simultaneous data transmission as much as possible and reduce the response time of the query.

4.1 Description of the algorithm H

Before we describe the algorithm, we need to discuss some related concepts and conditions firstly.

1. In this algorithm, we need to compare two filters (such as filter A_1, filter A_2) of the same attribute A.

   - If the bits that have been set to 1 in the two filters are exactly same, then we say that filter A_1 equals filter A_2.
   - If the bits that have been set to 1 in filter A_2 not only include all bits that have been set to 1 in filter A_1 but also include some more bits that have been set to 1, then we say that filter A_1 is smaller than filter A_2 and filter A_2 is larger than filter A_1.
   - If the bits that have been set to 1 in filter A_2 do not include all bits that have been set to 1 in filter A_1 and the bits that have been set to 1 in filter A_1 do not include all bits that have been set to 1 in filter A_2, we say that filter A_1 and A_2 are different.

2. In this algorithm, when we add any element (filter) to the queue or a list, there is a prerequisite – the element is not already on the queue or list.

3. In this algorithm, we need use two filterlists (filterlist1 & filterlist2) and one queue.

   Filterlist1 is used to keep all new/smallest/different filters. When we need decide to keep or discard the created filters in the algorithm, we will compare them with the filters in the filterlist1. If the created filters are new/smaller/different ones, we will keep them. Otherwise we will discard them.

   Filterlist2 is used to keep only the filters that will be used to reduce the other relations in this algorithm. After using them, the filters will be deleted from filterlist2.

   Queue is used to keep the relations in which the filters of some/all join attributes will be used to reduce other relations.

Details of the algorithm:

1. Select the relation(s) with the lowest number of joining attributes. If the number of selected relation(s) is more than one, select the relation with the higher degree (see 1.4) among them. We will denote this relation R_i.
2. Construct the bloom-filters of all join attributes contained in R_i and put all filters in both filterlist1 and filterlist2. Add R_i to queue.
3. Remove the relation from the front of queue. We will denote this relation R_j.
4. Determine if filters for any attributes of R_j exist in filterlist2, and apply them to reduce the relations that can be reduced by these filters.
5. Delete the used filters in the step 4 from filterlist2.
6. For each reduced relation in step 4, denote it R_t, construct the bloom-filters for all joining attributes contained in R_t and check these filters one by one.
   a. If it is a new filter, add it to filterlist1 and filterlist2; add R_t to queue.

b. If it only exists in filterlist1 and is not larger than or equal to any filter for the same joining attribute in filterlist1, compare it with the filters for the same joining attribute in filterlist1 one by one:

- If it is smaller, delete the existing one from filterlist1; add it to filterlist1 and filterlist2; add R_t to queue.
- If it is different from the existing one in fitlerlist1, add it to filterlist1 and filterlist2; add R_t to queue.

c. If it exists in both filterlist1 and filterlist2, compare it with all filters for the same joining attribute in filterlist1 and delete the larger ones from filterlist1.

If it does not equal any filter for the same joining attribute in filterlist1 and is not larger than or equal to any filter for the same joining attribute in filterlist2, compare it with the filters for the same joining attribute in filterlist2 one by one:

- If it is smaller, delete the existing one from filterlist2; add it to filterlist2 and filterlist1; put R_t to the position before the relation that creates the existing filter in filterlist2 on the queue.
- If it is different from the existing one in filterlist2, add it to filterlist2 and filterlist1; add R_t to queue.

7. Repeat step3 to 6 until queue is empty.

4.2 An example of using the algorithm

Following example illustrates how this algorithm works.

In this example, the query has four relations that must be joined. These four relations are shown in Figure 4.1.

R_1	C
	3
	6
	7
	8

R_2	A
	1
	2
	3
	4

R_3	A	C	D
	1	3	4
	4	6	5
	4	9	8

R_4	C	D
	2	2
	3	4
	5	6
	8	7

Figure 4.1 The four relations of the example

Domain size of joining attributes: A 5, C 10, D 10

Domain of an attribute includes all possible values of the attribute. The size of the filter of an attribute is the domain size of the attribute.

Using the example database given in Figure 4.1, the following query graph (see 1.4) is constructed.

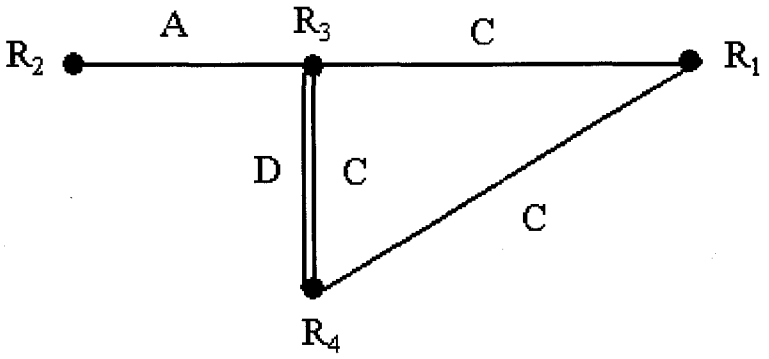

Figure 4.2 Query graph for example database

From this query graph, we find it is a cyclic query (see 1.4). The difficulty of the cyclic query is hard to terminate. But the example shows our proposed algorithm can handle the cyclic queries and easily terminate them.

1. In these four joining relations, R_1 and R_2 both have the lowest number of joining attributes, that is, one. But the degree of R_1 is two which is larger than the degree of R_2, that is, one. R_1 is selected; a filter for attribute C is produced (3,6,7,8); place the filter C_1 on the filterlist1 and filterlist2. Because the filters of joining attributes contained in R_1 will be used to reduce other relations, R_1 is placed on the queue.

   filterlist1 { C_1 } filterlist2 { C_1 } queue { R_1 }

2. Remove R_1 from the queue and use filter C of R_1 (C_1) in filterlist2 to reduce related relations (R_3, R_4); delete the filter C_1 from filterlist2.

   filterlist1 { C_1 } filterlist2 ∅ queue ∅

3. Check the reduced relations R_3 and R_4

   Produce the filter A, C, and D of R_3 and check these three filters.

R_3	A	C	D
	1	3	4
	4	6	5

filters A_3: 1, 4
C_3: 3, 6
D_3: 4, 5

   The filter A is a new filter, add it to filterlist1 and filterlist2; add R_3 to queue.

   The filter C only exists in filterlist1, compare C_3 with C_1 in filterlist1:

   $C_3 < C_1$, delete C_1 from filterlist1; add C_3 to filterlist1 and filterlist2; because R_3 is already on the queue, don't need add R_3 to the queue.

   The filter D is a new filter, add it to filterlist1 and filterlist2; because R_3 is already on the queue, don't need add R_3 to the queue.

   filterlist1 { A_3, C_3, D_3 } filterlist2 { A_3, C_3, D_3 } queue { R_3 }

Produce the filter C and D of R_4 and check these two filters.

R_4	C	D
	3	4
	8	7

filters C_4: 3, 8

D_4: 4, 7

The filter C exists in both filterlist1 and filterlist2, compare C_4 with all filter C in filterlist1; no larger ones can be deleted and no one equals C_4; compare C_4 with C_3 in filterlist2:

C_4 is different from C_3, add C_4 to filterlist2 and filterlist1; add R_4 to the queue.

The filter D exists in both filterlist1 and filterlist2, compare D_4 with all filter D in filterlist1; no larger ones can be deleted and no one equals D_4; compare D_4 with D_3 in filterlist2:

D_4 is different from D_3, add D_4 to filterlist2 and filterlist1; because R_4 is already on the queue, don't need add R_4 to the queue.

filterlist1 $\{ A_3, C_3, D_3, C_4, D_4 \}$ filterlist2 $\{ A_3, C_3, D_3, C_4, D_4 \}$ queue $\{ R_3, R_4 \}$

4. Remove R_3 from the queue and use filter A, C, D of R_3 (A_3, C_3, D_3) in filterlist2 to reduce related relations (R_1, R_2, R_4); delete the filter A_3, C_3, D_3 from filterlist2.

   filterlist1 $\{ A_3, C_3, D_3, C_4, D_4 \}$ filterlist2 $\{ C_4, D_4 \}$ queue $\{ R_4 \}$

5. Check the reduced relations R_1, R_2, and R_4

   Produce the filter C of R_1 and check it.

R_1	C	filter C_1: 3, 6
	3	
	6	

The filter C exists in both filterlist1 and filterlist2, compare C_1 with all filter C in filterlist1; no larger ones can be deleted; but we have $C_3 = C_1$, do not need compare C_1 with existing filter C in filterlist2.

Produce the filter A of R_2 and check this filter.

R_2	A	filter A_2: 1, 4
	1	
	4	

No change about filter A in filterlist1.

Produce the filter C and D of R_4 and check these two filters.

R_4	C	D	filters C_4: 3
	3	4	D_4: 4

The filter C exists in both filterlist1 and filterlist2, compare new C_4 with all filter C in filterlist1 and delete the larger ones (C_3, existing C_4) from filterlist1; no one equals new C_4 in filterlist1; compare new C_4 with the existing C_4 in filterlist2:

new C_4 < existing C_4, delete the existing C_4 from filterlist2; add new C_4 to filterlist2 and filterlist1; because R_4 is already on the queue, don't need add R_4 to the queue.

The filter D exists in both filterlist1 and filterlist2, compare new D_4 with all filter D in filterlist1 and delete the larger ones (D_3, existing D_4) from filterlist1; no one equals new D_4 in filterlist1; compare new D_4 with the existing D_4 in filterlist2:

new D_4 < existing D_4, delete the existing D_4 from filterlist2; add new D_4 to filterlist2 and filterlist1; because R_4 is already on the queue, don't need add R_4 to the queue.

filterlist1 { A_3, C_4, D_4 } filterlist2 { C_4, D_4 } queue { R_4 }

Note: The C_4 and D_4 in both filterlists are new ones now.

6. Remove R_4 from the queue and use filter C and D of R_4 (C_4, D_4) in filterlist2 to reduce related relations (R_1, R_3); delete the filter C_4, D_4 from filterlist2.

filterlist1 { A_3, C_4, D_4 } filterlist2 ∅ queue ∅

7. Check the reduced relations R_1 and R_3

Produce the filter C of R_1 and check it.

R_1	C
	3

filter C_1: 3

No change about filter C in filterlist1.

Produce the filter A of R_3 and check it.

R_3	A	C	D
	1	3	4

filters A_3: 1

C_3: 3

D_3: 4

The filter A only exists in filterlist1, compare new A_3 with the existing A_3 in filterlist1:

New A_3 < existing A_3, delete the existing A_3 from filterlist1; add new A_3 to filterlist1 and filterlist2; add R_3 to queue.

No change about filter C and D in filterlist1.

filterlist1 { A_3, C_4, D_4 } filterlist2 { A_3 } queue { R_3 }

Note: The A_3 in both filterlists is new one now.

8. Remove R_3 from the queue and use filter A of R_3 (A_3) in filterlist2 to reduce related relation (R_2); delete the filter A_3 from filterlist2.

   filterlist1 { A_3, C_4, D_4 } filterlist2 ∅ queue ∅

9. Check the reduced relation R_2

   Produce the filter A of R_2 and check it.

   R_2 A filter A_1: 1

   1

   No change about filter A in filterlist1.

10. The queue is empty now. The algorithm stops.

Now, we will calculate the Response Time (RT) of the example query.

The Response Time of a query is the elapsed time between the start of the query and the final results are obtained.

The RT of this query is as follows:

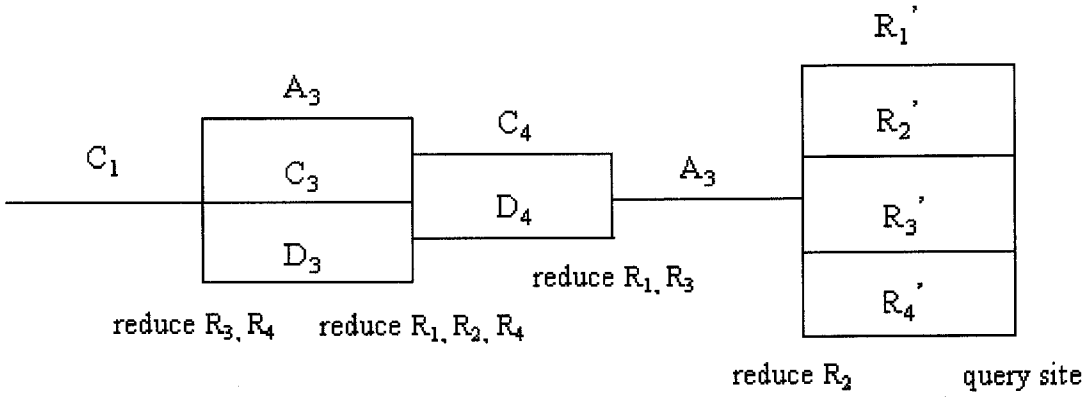

Figure 4.3 The RT of the example query

RT: $C_1 + \text{Max}(A_3, C_3, D_3) + \text{Max}(C_4, D_4) + A_3 + \text{Max}(R_1', R_2', R_3', R_4')$

$= 10/64 + \text{Max}(5/64, 10/64, 10/64) + \text{Max}(10/64, 10/64) + 5/64 + \text{Max}(1, 1, 3, 2)$

$= 0.15625 + 0.15625 + 0.15625 + 0.078125 + 3$

$= 3.546875$

Chapter 5 EXPERIMENTS AND EVALUATION

To study whether the use of the algorithm H leads to better performance, we carried out various experiments based on a large number of queries. In this chapter, we discuss our methodology, present details of our experiments, and finally discuss the significance of these results.

5.1 Methodology

The framework for evaluating the algorithm is based on the following objective:

- To compare the proposed algorithm H against the effects of the Initial Feasible Solution (IFS) that ships all relations directly to the query site and performs joins there. Our algorithm is evaluated to determine how much better it comes to reducing the response time when comparing with IFS. This comparison is done under the assumption of a perfect hash function.

5.1.1 Experimental rationale

With few exceptions, previously proposed algorithms have not been objectively evaluated. Heuristic have been evaluated for performance by comparison with another heuristic or not evaluated for performance at all. But since the mostly compared heuristic algorithms have different assumptions and restrictions, it is hard to determine the performance of which heuristic is better. The Initial Feasible Solution is an algorithm that ships all relations directly to the query site and performs joins there. Since it is a very simple heuristic that has no assumptions or restrictions, then comparing against IFS, the performance of our proposed algorithm is being evaluated objectively.

5.1.2 Experimental system

The experimental system includes the query-relation generator, algorithm H, IFS, and Analysis program.

The query-relation generator that is described in 5.1.3 was programmed by the Database Research Group of the University of Windsor.

The algorithm H has been described in Chapter 4 and implemented in C++.

IFS that is introduced in 5.1.4 is an algorithm that ships all relations directly to the query site and performs joins there.

The analysis program that is introduced in 5.1.5 is to evaluate the performance of our algorithm.

In experiments we constructed 3600 different queries as input. The queries and relations vary in the number of relations, the number of joining attributes, the relation size, the domain size, and the selectivity.

5.1.3 Test queries (query-relation generator)

The proposed algorithm is evaluated using Select – Project – Join (SPJ) queries. A variety of test queries contain the following characteristics:

- Number of relations and attributes

  Each query consisted of between 3 and 6 relations and the number of joining attributes varied between 2 and 4. Overall, this gave us 12 different types of test queries (from 3 relations – 2 attributes to 6 relations – 4 attributes).

- Relation cardinality – the number of tuples or records in a relation.

  Each relation in the query has between 150 and 2000 tuples.

- Attribute domain sizes – the total number of distinct attribute values an attribute can contain.

  The size of each join-attribute domain varies between 100 and 600.

- Selectivity – the ratio of distinct attribute values over the attribute domain size. Intuitively, the selectivity of an attribute is an estimate of the ability of the attribute to reduce the size of the relations. For clarification, a joining attribute has high selectivity if the ratio is low, and low selectivity if the ratio is high. For example, a selectivity of 0.1 is considered high while a selectivity of 0.9 is considered low.

  The selectivities in the query are in the range between 0.1 and 0.9.

- Connectivity – 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 total number of possible joining attributes is a product of the number of relations and the number of common joining attributes.

  For our evaluation we consider minimum connectivity is 50%.

The actual query and relation construction is handled by the C program create_query.c and relbuilder.c that are described in [Bea95] respectively. We describe these two C programs below for completeness:

- *create_query.c*: This program generates a query. The input includes the number of relations, the number of common join attribute, and the range of attribute selectivities. The output consists of the file 'dbstats', file 'domains', and several 'Rel' files.

The file 'dbstats' which is the database statistics contains the number of relations, the number of common joining attributes, the relation cardinalities, and each attribute cardinality and selectivity.

The file 'domains' that is the domain size consists of the domain size for each common joining attribute.

For each relation specified in the query, a file 'Rel' is generated, which consists of the relation cardinality, the number of joining attributes, and for each joining attribute, the attribute label, the size of the attribute and its domain.

The example, given in Figure 5.1, shows these files for a query.

dbstats

3 2

258 104 0.7 76 0.6

392 0 0.0 114 0.9

532 101 0.68 0 0.0

domains

148

127

Rel0

258 2 0 104 148 1 76 127

Figure 5.1 Example files for a query

Consider the 'dbstats' file in Figure 5.1. Line 1 contains the number of relations (3) and the number of common joining attributes (2). Lines 2, 3, and 4 contain the

statistics for each relation specified in the query. For relation R_0, represented by line 2 of 'dbstats', the cardinality is 258. The following numbers in line 2 are the data about two attributes of R_0.

The first attribute is attribute 0, with a cardinality of 104 and a selectivity of 0.7. The second attribute is attribute 1, with a cardinality of 76 and a selectivity of 0.6. The same data about R_1 and R_2 are represented in line 3 and line 4 of 'dbstats' respectively.

The 'domains' file in Figure 5.1 contains a domain size of 148 for common joining attribute 0 and a domain size of 127 for common joining attribute 1.

The 'Rel0' file in Figure 5.1 is generated for relation R_0. The first number (258) is the cardinality of relation R_0. The second number (2) is the number of joining attributes in the query. The third number (0) represents attribute 0 whose cardinality and domain size are the forth and fifth number (104, 148). The sixth number (1) represents attribute 1 whose cardinality and domain size are the seventh and eighth number (76, 127). Similar 'Rel' files are created for relation R_1 and R_2.

- *relbuilder.c*: This program generates a relation based on the statistics generated in *create_query.c*. The input of *relbuilder* is a number indicating the relation to generate. *rebuilder* uses this number to access the corresponding 'Rel' file, which is generated by *create_query.c*. The output is a 'R' file, which contains the required number of tuples of the corresponding relation and the necessary header information, including the number of joining attributes and the joining attribute labels.

5.1.4 Initial Feasible Solution (IFS)

To evaluate the algorithm H that is described in chapter 4 objectively, the performance of algorithm H is compared against IFS. For this reason, we have developed an IFS program. The Initial Feasible Solution is an algorithm that ships all relations directly to the query site and performs joins there. We explain how the IFS works with the same example we used to describe algorithm H in chapter 4.

Four relations are shown in Figure 5.2.

R_1	C
	3
	6
	7
	8

R_2	A
	1
	2
	3
	4

R_3	A	C	D
	1	3	4
	4	6	5
	4	9	8

R_4	C	D
	2	2
	3	4
	5	6
	8	7

Figure 5.2 The four relations of the example

The sizes of R_1, R_2, R_3, and R_4 are 4, 4, 9, and 8 respectively.

Depend on the definition of the Response Time which is the elapsed time between the start of the query and the final results are obtained, the RT of this example should be the maximum one among the sizes of four relations.

RT: Max (R_1, R_2, R_3, R_4)

= Max (4, 4, 9, 8)

= 9

5.1.5 Analysis program

To evaluate the performance of our proposed algorithm H precisely, we have produced the analysis program to collect and compare results generated by algorithm H and IFS. The analysis program calculates the reduced RT (Response Time) that is based on the following formula.

$$\text{reduced RT (\%)} = [\ (\text{RT by IFS} - \text{RT by H}) / \text{RT by IFS}\ ] * 100$$

We still use the same example that is described in chapter 4 and in 5.1.4 to explain this formula.

RT by IFS: 9

RT by H: 3.546875

$$\text{reduced RT (\%)} =[\ (9 - 3.546875) / 9\ ] * 100 = 60.59$$

The analysis program still calculates the average RT achieved by algorithm H and by IFS and the average reduced RT when we compare algorithm H against IFS.

5.2 Experiments and results

In the experiments, each relation in the query consists of (150 - 2000) tuples. The attribute domain contains (100 - 600) distinct values. The experiments carried out are divided into three parts based on the selectivities of all joining attributes in the test queries: high selectivity (0.1 - 0.4), middle selectivity (0.4 – 0.7), and low selectivity (0.7 – 0.9). For each test query type, 100 queries were constructed and executed using algorithm H and IFS, recording the costs incurred. So, each data in the following result tables is the average result of 100 queries. Overall, a total of 3600 queries were used to evaluate the performance of the proposed algorithm H.

Type	RT by IFS	RT by H	Reduced RT
3-2	3042.86	261.06	90.49
3-3	4367.14	154.49	95.56
3-4	4991.43	147.77	96.71
4-2	3018.57	242.25	91.67
4-3	4158.57	217.1	93.81
4-4	5848.57	292.42	94.43
5-2	3240	248.39	91.9
5-3	4805.71	283.15	93.78
5-4	5275.71	243.98	94.23
6-2	3594.29	242	93.14
6-3	4461.43	301.76	92.74
6-4	6242.86	271.28	95.44
Average of the Column	4420.6	242.14	93.66

Figure 5.3 Results of reduced RT at selectivity 0.1-0.4

- The results of the comparison in Figure 5.3 show that, the RT by using the algorithm H equals less than 10% of the RT by using IFS. On average, algorithm H outperforms IFS by 93.66% when the selectivity is between 0.1 and 0.4.
- For test query type 3-2, 4-2, and 5-2, the average reductions of RT are lower than those of the remaining test query types. But the algorithm H still achieves 90+% reduction of RT.
- For test query type 3-3, 3-4, and 6-4, the average reductions of RT are higher than those of the remaining test query types. The algorithm H achieves 95+% reduction of RT.

Type	RT of IFS	RT of H	Reduced RT
3-2	3282.86	530.93	83.96
3-3	4642.86	599.13	85.5
3-4	4577.14	421.61	89.92
4-2	3444.29	465.62	85.85
4-3	4565.71	649.34	85.12
4-4	5192.86	517.36	89.31
5-2	3281.43	355.28	88.15
5-3	4415.71	432.29	88.66
5-4	5722.86	477.26	90.76
6-2	3367.14	351.36	89.23
6-3	4918.57	424.26	89.93
6-4	6028.57	559.28	90.32
Average of the Column	4453.33	481.98	88.06

Figure 5.4 Results of reduced RT at selectivity 0.4-0.7

- The results of the comparison in Figure 5.4 show that, the RT by using the algorithm H equals less than 20% of the RT by using IFS. On average, algorithm H outperforms IFS by 88.06% when the selectivity is between 0.4 and 0.7.
- This average reduction (88.06%) is lower than the one at selectivity 0.1-0.4 (93.66%) about 5%.
- For test query type 3-2, 3-3, 4-2, and 4-3, the average reductions of RT are lower than those of the remaining test query types. But the algorithm H still achieves 83+% reduction of RT.
- For test query type 5-4 and 6-4, the average reductions of RT are higher than those of the remaining test query types. The algorithm H achieves 90+% reduction of RT.

Type	RT of IFS	RT of H	Reduced RT
3-2	3125.71	1559.5	50.29
3-3	4861.43	2390.94	50.94
3-4	5982.86	2742.33	53.88
4-2	3594.29	1606.83	55.66
4-3	4518.57	2146.83	52.37
4-4	6382.86	2741.08	55.54
5-2	3761.43	1669.87	56.45
5-3	4555.71	1868.65	57.84
5-4	6175.71	2609.94	56.21
6-2	3434.29	1405.93	58.72
6-3	5118.57	1887.53	62.8
6-4	6272.86	2114.59	64.83
Average of the Column	4815.36	2070.25	56.29

Figure 5.5 Results of reduced RT at selectivity 0.7-0.9

- The results of the comparison in Figure 5.5 show that, the RT by using the algorithm H equals less than 50% of the RT by using IFS. On average, algorithm H outperforms IFS by 56.29% when the selectivity is between 0.7 and 0.9.
- This average reduction (56.29%) is lower than the one at selectivity 0.1-0.4 (93.66%) about 37% and lower than the one at selectivity 0.4-0.7 (88.06%) about 32%.
- For test query type 3-2, 3-3, 3-4, and 4-3, the average reduction of RT is lower than those of the remaining test query types. But the algorithm H still achieves 50+% reduction of RT.
- For test query type 6-3 and 6-4, the average reduction of RT is higher than those of the remaining test query types. The algorithm H achieves 60+% reduction of RT.

5.3 Discussion

The performance evaluation shows that, on average, the proposed algorithm H gives good improvement, even when the selectivity is low. Actually, in some cases, the proposed

algorithm H has some kind of redundancy problem. This is illustrated with a simple example that is changed a little from the example described in chapter 4.

R_1	C
	3
	6
	7
	8

R_2	A
	1
	2
	3

R_3	A	C	D
	1	3	4
	4	6	5
	4	9	8

R_4	C	D
	2	2
	3	4
	5	6
	8	7

Figure 5.6 The four relations of the example

In this example, we just remove the last value of joining attribute A of R_2. We don't explain every step in detail because it has already been done in the description of the similar example in chapter 4. We directly give the result of each step as follows. The redundancy occurs in the step 8.

1. R_1 is selected; a filter for attribute C is produced (3,6,7,8); place the filter C on the filterlist1 and filterlist2. R_1 is placed on the queue.

   filterlist1 $\{ C_1 \}$ filterlist2 $\{ C_1 \}$ queue $\{ R_1 \}$

2. Remove R_1 from the queue and use filter C of R_1 (C_1) in filterlist2 to reduce related relations (R_3, R_4); delete the filter C_1 from filterlist2.

   filterlist1 $\{ C_1 \}$ filterlist2 $\varnothing$ queue $\varnothing$

3. Check the reduced relations R_3 and R_4

Produce the filter A, C, and D of R_3 and check these three filters.

R_3	A	C	D
	1	3	4
	4	6	5

filters A_3: 1, 4

C_3: 3, 6

D_3: 4, 5

filterlist1 { A_3, C_3, D_3 } filterlist2 { A_3, C_3, D_3 } queue { R_3 }

Produce the filter C and D of R_4 and check these two filters.

R_4	C	D
	3	4
	8	7

filters C_4: 3, 8

D_4: 4, 7

filterlist1 { A_3, C_3, D_3, C_4, D_4 } filterlist2 { A_3, C_3, D_3, C_4, D_4 } queue { R_3, R_4 }

4. Remove R_3 from the queue and use filter A, C, D of R_3 (A_3, C_3, D_3) in filterlist2 to reduce related relations (R_1, R_2, R_4); delete the filter A_3, C_3, D_3 from filterlist2.

   filterlist1 { A_3, C_3, D_3, C_4, D_4 } filterlist2 { C_4, D_4 } queue { R_4 }

5. Check the reduced relations R_1, R_2, and R_4

   Produce the filter C of R_1 and check it.

R_1	C
	3
	6

filter C_1: 3, 6

No change about filter C.

Produce the filter A of R_2 and check this filter.

R_2 A filter A_2: 1

1

filterlist1 { C_3, D_3, C_4, D_4, A_2 } filterlist2 { C_4, D_4, A_2 } queue { R_4, R_2 }

Produce the filter C and D of R_4 and check these two filters.

R_4 C D filters C_4: 3

3 4 D_4: 4

filterlist1 { C_4, D_4, A_2 } filterlist2 { C_4, D_4, A_2 } queue { R_4, R_2 }

6. Remove R_4 from the queue and use filter C and D of R_4 (C_4, D_4) in filterlist2 to reduce related relations (R_1, R_3); delete the filter C_4, D_4 from filterlist2.

filterlist1 { C_4, D_4, A_2 } filterlist2 { A_2 } queue { R_2 }

7. Check the reduced relations R_1 and R_3

Produce the filter C of R_1 and check it.

R_1 C filter C_1: 3

3

No change about filter C.

Produce the filter A of R_3 and check it.

R_3 A C D filters A_3: 1

1 3 4 C_3: 3

D_3: 4

No change about filter A, C, and D

filterlist1 { C_4, D_4, A_2 } filterlist2 { A_2 } queue { R_2 }

8. Remove R_2 from the queue and use filter A of R_2 (A_2) in filterlist2 to reduce related relation (R_3); delete the filter A_2 from filterlist2.

   filterlist1 $\{ C_4, D_4, A_2 \}$ filterlist2 $\varnothing$ queue $\varnothing$

9. Check the reduced relation R_3 (*it's same as in step 7. Step 8 is redundant*)

   Produce the filter A of R_3 and check it.

R_3	A	C	D
	1	3	4

filters A_3: 1

C_3: 3

D_3: 4

   No change about filter A, C, and D.

10. The queue is empty now. The algorithm stops.

Actually, in the first 7 steps, the all relations have already been reduced fully. But because there still has the filter A of R_2 (A_2) in the filterlist2 and R_2 on the queue, we have to do the step 8 in which the proposed algorithm H can not further reduce any relation in the query. The step 8 is useless. It is redundant.

The reason why the algorithm H has the redundancy is the following. The relation R_3 supposed to be further reduced using A_2 in step 8 has already been reduced fully when the algorithm uses C_4, D_4 to reduce the relation R_3 in step 7. Although the relation R_3 has been reduced fully in step 7, the algorithm still have to try to reduce it again in step 8 since there still has the filter A of R_2 (A_2) in the filterlist2 and the R_2 on the queue. The algorithm did not stop until the queue is empty.

Because every relation has at least one joining attribute, if the relation has more than one joining attributes, it is possible for this kind of redundancy to occur.

Although the proposed algorithm H has the redundancy, we still find the performance of our proposed algorithm H is significantly better from the evaluation of this algorithm.

Chapter 6 CONCLUSIONS AND FUTURE WORK

In this thesis, we propose a new filter-based algorithm that uses bloom-filters to process general queries. The primary goal of our algorithm is to reduce the response time of a distributed query. The secondary goal is to reduce relation sizes while using data transmission as little as possible.

This algorithm can process general queries consisting of an arbitrary number of relations and joining attributes. Most heuristic algorithms in distributed database systems can only be used for tree queries (see 1.4) or simple queries (see 1.4). Even if some heuristics can process general queries, it is difficult for them to be efficient for cyclic queries. The difficulty of cyclic queries is hard to terminate. But our proposed algorithm can handle the cyclic queries.

The performance of our proposed algorithm with respect to response time is compared against the Initial Feasible Solution (IFS) to determine how much better it reduces the response time. We perform some experiments to evaluate the proposed algorithm. The test data used to evaluate the algorithm consists of many Select- Project-Join (SPJ) queries, which vary in selectivity, number of relations, and number of joining attributes. Analyzing the results of the evaluation, we get the following figure 6.1.

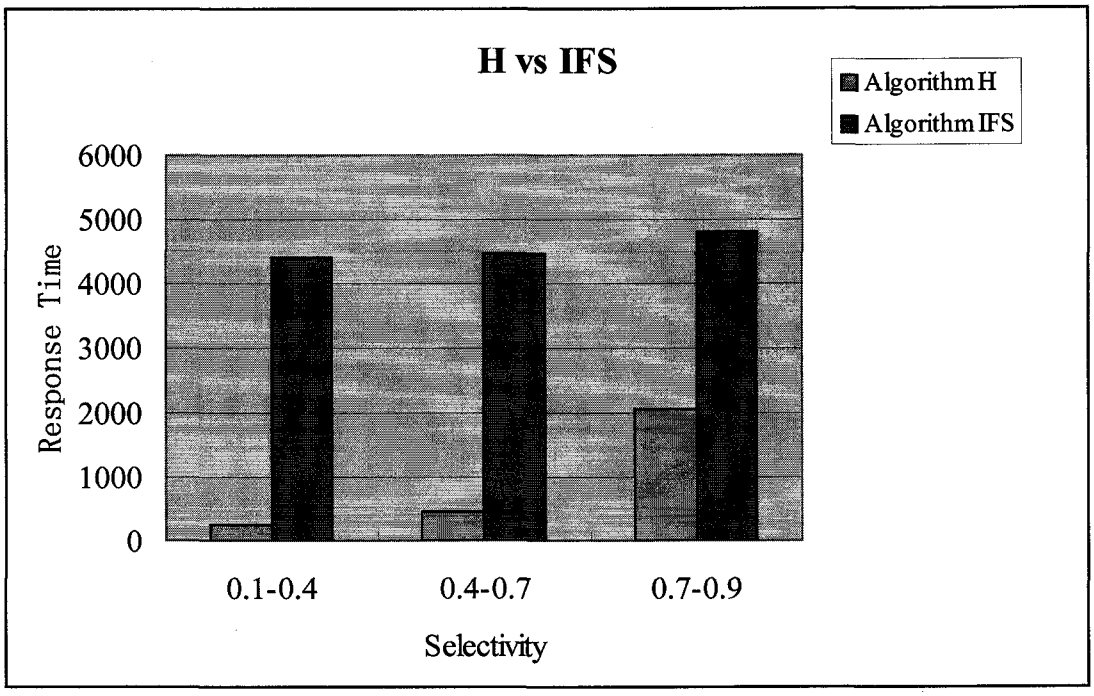

Figure 6.1 H – IFS cost comparisons

- The improvement of algorithm H decreases as the selectivity decreases. In other words, the Response Time of algorithm H increases as the selectivity decreases.
- Algorithm H clearly outperforms IFS (as illustrated in Figure 6.1). On average, algorithm H outperforms IFS by approximately 79.34% in all the cases. The greatest difference in performance is found in those queries whose selectivities of joining attributes are between 0.1 and 0.4.
- The Response Times of the queries whose selectivities of joining attributes are between 0.7 and 0.9 by using algorithm H are much higher than the remaining queries.
- We found the number of relations is an important factor to affect the performance of our proposed algorithm H. The experiment results in chapter 5 indicate that the more number of relations, the better the effect of the reduction of RT, especially in the cases in which the selectivity is 0.4-0.7 or 0.7-0.9.

- For the number of joining attributes, we found that for the queries containing the same number of relations, the more number of joining attributes, the larger reduction of RT.

In conclusion, if the high selectivity (0.1-0.4) is used in our algorithm H, then our algorithm performs excellent. Otherwise, in order to obtain better reduction performance, we suggest to adopt those queries with more number of relations and more number of attributes.

6.1 Future work

We assume that we have a perfect hash function and apply it to our proposed algorithm H. That means no collision problems in our algorithm. In reality, collision problems always exist in filter-based algorithms. So one key area of continued research is to test the proposed algorithm under the situation with collisions and try to use multiple bloom-filters for each common joining attribute to attempt to minimize the collision problem.

Another key area of continued research is to test the effects of our proposed algorithm about cyclic queries by comparing against another algorithm that can handle cyclic queries too.

BIBLIOGRAPHY

[AHY83] Peter M.G.Apers, Alan R.Hevner, and Bing Yao. "Optimization algorithms for distributed queries". IEEE Transactions on Software Engineering, vol.9(1), pp.57-68, 1983.

[AM91] J.K.Ahn and S.C.Moon. "Optimization joins between two fragmented relations on a broadcast local network". Info. Sys, vol.16(2), pp.185-198, 1991.

[ABCE+76] M.M.Astrahan et al. "System R: A relational approach to data management". ACM Transactions on Database Systems, vol.1(2), pp.97-137, June 1976.

[Bab79] E.Babb. "Implementing a relational database by means of specialized hardware". ACM Transactions on Database Systems, vol.4(1), pp.1-29, 1979.

[Bea95] William T. Bealor. "Semi-join strategies for total cost minimization in distributed query processing". Master thesis, University of Windsor, 1995.

[BC81] P.Bernstein and D.Chiu. "Using semi-join to solve relational queries". Associaton for Computing Machinery Journal, vol.28, pp.25-40, Jan 1981.

[BGWR+81] P.Bernstein, N.Goodman, E.Wong, C.Reeve, and J.Rothnie. "Query processing in a system for distributed databases (SDD-1)". ACM Transactions on Database Systems, vol.6(4), pp.602-625, 1981.

[BDT83] Dian Bitton, David J.Dewitt and Carolyn Turbyfill. "Benchmarking Database Systems - a Systematic Approach". Technique report, Computer Science, University of Wisconsin-Madison, 1983.

[Blo70] B.H.Bloom. "Space/time trade-offs in hash coding with allowable errors". Communication of the ACM, vol.13(7), pp.422-426, 1970.

[BMS96] S.Bsnfyopsfhysy, J.M.Morrissey, and A.Senpupta, "A query optimization strategy for distributed databases on all0-optical networks". In Proceedings of the Canadian Conferencd on Electrical and Computer Engineering, pp.245-248, 1996.

[BR88] P. Bodorik and J.S.Riordon. "A threshold mechanism for distributed query processing". In Proc. Of the ACM Computer Science Conference, pp.616-621, 1988.

[Bra84] Kjell Bratbergsengen. "Hashing methods and relational algebra opteations ". In Proc. Of the 10th International Conference on VLBD, pp 323-333, 1984.

[CCY92] T.S.Chen, A.L.P.Chen, and W.P.Yang. "Hash-semijoin: A new technique for minimizing distributed query time". In Proc. Of the 3rd Workshop on Future Trends of Distributed Computing Systems, pp 325-330, 1992.

[CL89] J.S.J.Chen and V.O.L.Li. "Optimizing joins in fragmented database systems on a broadcast local network". IEEE Transactions of Software Engineering, vol.15(1), pp.26-38, 1989.

[CL90] J.S.J.Chen and V.O.K.Li. "Domain-specific semijoin: a new operation for distributed query procesing". Information Sciences, vol.52, pp.165-183, 1990.

[CP84] S.Ceri and G.Pelagetti. "Distributed Databases: Principles and Systems".McGarw-Hill, 1984.

[CY90] M.S.Chen and P.S.Yu. "Using join operations as reducers in distributed query processing". In Proc. Of the 2nd Internatio Symposium on Database in Parallel and Distributed Systems, pp.116-123, 1990.

[Hen80] A.r. Henver. "The optimization of query processing in distributed database systems".PhD thesis, Purdue University, 1980.

[GS86] B.Gacish and A.Segev. "Set query optimization in distributed database systems". ACM Transactions on Database Systems, vol.11(3), pp.265-293, 1986.

[GLPK94] C.Galindo-Legaria, A.Pellenkoft, and M.Kersten. "Fast, randomized join-order selection - why use transformations?". In Proc. 20th Int. VLDB Conference, pp.85-95, Santiago, Chile, September 1994. (Also available as CWI Tech. Report CS-R9416.).

[Kam96] S.Kamat. "Dynamic strategy and bloom filters in distributed query optimization". Master thesis, University of Windsor, 1996.

[KR87] H.Kang and N.Roussopoulos. "Using 2-way semijoins in distributed query processing". In Proc. of The 3rd International Conference on Data Engineering, pp.664-651, 1987.

[Lia99] Y.Liang. "Reduction of collisions in bloom filters during distributed query optimization". Master thesis, University of Windsor, 1999.

[LMHD+85] G.M.Lohman et al. "Query processing in R*". Query processing in database system, Springer, New York, 1985.

[LPP91] P.Legato, G.Paletta and L.Palopoli. "Optimization of join strategies in distributed database". Information Systems, vol.16(4), pp.363-374, 1991.

[LR95] Z.Li and K.A.Ross. "PERF join: an alternative to two-way semijoin and bloomjoin". In Proc. of CIKM'95, pp.137-144, 1995.

[Ma97] X.Ma. "The use of bloom filters to minimize response time in distributed query optimization". Master thesis, University of Windsor, 1997.

[MB96] J.M.Morrissey and W.T.Bealor. "Minimizing data transfers in distributed query optimization: A comparative study and evaluation". The Computer Journal, vol.39, no.8, pp.676-687, Dec 1996.

[MBB95] 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.

[ME92] P.Mishra and M.Eich. "Join processing in relational databases". ACM Computing Surveys, vol.24(1), pp.63-113, March 1992.

[MOL00] J.M.Morrissey, W.K.Osborn, and Y.Liang. "Collisions and reduction filters in distributed query processing". In Proc. of the 2000 IEEE Conference on Electrical and Computer Engineering, vol.1, pp.240-244, 2000.

[MM98] J.M.Morrissey and X.Ma. "Investigating response time minimization in distributed query optimizaiton". Present at ICCI'98, pp.124-138, 1998.

[MO98] J.M.Morrissey and W.K.Osborn. "Distributed query optimization using reduction filters". In Proc. of the 1998 IEEE Conference on Electrical and Comuter Engineering, vol.2, pp.707-710, 1998.

[Mor96] J.M.Morrissey. "Reduction filters for minimizing data transfers in distributed query optimization". In Proc. of the 1996 Canadian Conference on Electrical and Computer Engineering, vol.1, pp.198-201, 1996.

[Mul83] James K.Mullin. "A second look at bloom filters". Communication of the ACM, vol.26(8), pp.570-571, August 1983.

[Mul90] James K.Mullin. "Optimal semijoins for distributed database systems". IEEE Transactions on software engineering, vol.16(5), pp.558-560, May 1990.

[Mul93] James K.Mullin. "Estimating the size of a relational join". Information Systems, vol.18(3), pp.189-196, 1993.

[Osb98] Wendy.K.Osborn. "The use of reduction filters in distributed query optimization". Master thesis, university of windsor, 1998.

[ŐV99] M.T, Őzsu and P.Valdurize. "Principles of distributed database systems". Second 2nd ed., Upper Saddle River, NJ: Prentice-Hall, 1999.

[PC90] W.Perrizo and C.S.Chen. "Composite semijoin in distributed query processing". Information Sciences, vol.50, pp.197-218, 1990.

[RK91] N.Roussopoulos and H.Kang. "A pipeline n-way join algorithm based on the 2-way semijoin program". IEEE Transactions on Knowledge and Data Engineering, vol.3(4), pp.486-495, 1991.

[Seg84] A.Segev. "Optimizing fragmented 2-way joins". In Proc. Of the 4th International Conference on Distributed Computing Systems, pp.378-388, 1984.

[TC92] J.C.R. Tseng and A.L.P.Chen. "Improving distributed query processing by hash-semijoins". Journal of Information Science and Engineering, vol.8, pp525-540, 1992.

[TC94] P.S.M.Tsai and A.L.P.Chen. "Optimizing entity join queries by extended semijoins in a wide area multidatabase environment". In Proc. 1994 International Conference on Parallel and Distributed Systems, pp.676-681, 1994.

[Teo92] T.J.Teorey. "On dependability in distribute databases". Technical report, Center for Information Technology Integration, University of Mochigan, September 1992.

[VG84] P. Valduriez and G. Gardarin. "Join and semijoin algorithms for a multiprocessor database machine". ACM Transactions on Database Systems, vol.9(1), pp 133-161, 1984.

[WC96] C.Wang and M.-S.Chen. "On the complexity of distributed query optimization". IEEE Transactions on Knowledge and Data Engineering, vol. 8(4), pp.650-662, 1996.

[WCS92] C.Wang, A.L.P.Chen, and S.-C.Shyu. "A parallel execution method for minimizing distributed query response time". IEEE Transactions on Parallel and Distributed Systems, vol.3(3), pp.325-332, 1992.

[WLC91] C.Wang, V.O.K.Li, and A.L.P.Chen. "Distributed query optimization by one-shot fixed precision semijoin execution". In Proc. 7th International Conference on Data Engineering, pp.756-763, April 1991.

[YC84] C.Yu and C.Chang. "Distributed query processing". ACM Computing Surveys, vol.16(4), pp.399-433, December 1984.

[YL89] H.Yoo and S.Lafortune. "An intelligent search method for query optimization by semijoins". IEEE Transaction on Knowledge and Data Engineering, vol.1(2), pp.226-237, June 1989.

[YLGT+86] C.Yu, L.Lilien, K.Guh, M.Templeton, D.Brill, and A.Chen. "Sdaptive techniques for distributed query optimization". In Proc. Of the 2nd International Conference on Data Engineering, pp 86-93, 1986.

Vita Auctoris

Wanxin Gao was born in Hefei, China. She graduated from Univeristy of Anhui obtaining a Bachelor's Degree in Computer Science in 1994. From there she joined the Anhui International Trust & Investment Corporation, China as Network & Database administrator. She is currently a candidate for a Master's degree in Computer Science at the University of Windsor and will graduate in the Winter of 2003.